SYNTHESIS OF DISTRIBUTED PROTOCOLS FROM SCENARIOS AND SPECIFICATIONS

Abhishek Udupa

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2016

Rajeev Alur, Zisman Family Professor of Computer and Information Science
Supervisor of Dissertation

Lyle Ungar, Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee:
Chaired by Steve Zdancewic, Professor of Computer and Information Science
Joseph Devietti, Assistant Professor of Computer and Information Science
Oleg Sokolsky, Research Associate Professor of Computer and Information Science
Stavros Tripakis, Associate Professor, Aalto University, Finland

SYNTHESIS OF DISTRIBUTED PROTOCOLS

FROM SCENARIOS AND SPECIFICATIONS

To my parents

Acknowledgments

I would like to thank my advisors Rajeev and Milo for their continual support and mentoring over the past five years. Rajeev provided me with a great deal of freedom to build tools and pursue my own ideas, while also gently nudging me in the right direction whenever I drifted too far off course. His problem solving techniques have shaped my own research abilities, and will continue to shape the way I approach problems in the years to come. Milo was always available to provide me with solid advice, whether it be on research, or other professional and personal matters. It is no understatement when I say that this dissertation would not have been possible without their mentoring.

I thank my dissertation committee chaired by Steve Zdancewic, and with Oleg Sokolsky, Joseph Devietti and Stavros Tripakis as members, for their comments and feedback that have served to improve the overall quality of this dissertation. I am also grateful to them for being extremely flexible with respect to the scheduling of the dissertation proposal and defense.

I am grateful to my parents, who encouraged my "scientific" curiosity at an early age, even if meant that I would take apart things and need a lot of assistance in putting them back together, assuming that I had not destroyed it. On a more serious note, they have provided me with every opportunity that paved the path to this dissertation, and tolerated all my off-kilter views on a variety of topics, and I thank them for that, and for not asking how long until I graduate too many times.

The work described in this dissertation was completed in collaboration with a great set of collaborators. Arun Raghavan, Santosh Nagarakatte and Jyotirmoy Deshmukh helped me find my feet in my early graduate school days. Stavros Tripakis, Christos Stergiou, Arjun Radhakrishna and Mukund Raghothaman have been a pleasure to work with. I thank them for being such awesome collaborators.

I thank Sudipto Guha, Rajeev Alur, Milo Martin, Ben Taskar, Val Tannen and Benjamin Pierce for being great instructors and putting the effort into teaching the courses at Penn that I have benefitted immensely from.

My stay at Penn was enriched by the company of great friends like Mukund Raghothaman, Christos Stergiou, Arjun Radhakrishna, Arun Raghavan, Jyotirmoy Deshmukh, Salar Moarref, Christian Delozier, Arjun Narayan and Katherine Gibson. I hope that these friendships will continue to grow, even after I graduate.

Outside of Penn, my friends from college, Aaron, Dianne, Aswin, Raksha, Chengappa, Nishi and Alden have proved to be gracious hosts on my various visits to and vacations in their respective cities, as well as objective, non-judgmental sounding boards in getting my thoughts straight at various points.

I would also like to thank my Master's thesis advisors, R. Govindarajan and Matthew J. Thazhuthaveetil, at the Indian Institute of Science, Bangalore, who encouraged, supported and mentored my very first research projects. Thanks are also due to Sriram Rajamani, Aditya Nori, Bill Thies and Kaushik Rajan, who have all mentored me during my various stints at Microsoft Research India, as well as Murali Talupur, who was my mentor during an internship at Intel Corporation. The mentoring I received from all of these people played a large role in my decision to pursue, and continue with a doctoral degree.

The research described in this dissertation was partially supported by NSF award CCF 0905464 and the NSF Expeditions in Computing grant CCF 1138996.

Abstract

Distributed protocols, typically expressed as stateful agents communicating asynchronously over buffered communication channels, are difficult to design correctly. This difficulty has spurred decades of research in the area of automated model-checking algorithms. In turn, practical implementations of model-checking algorithms have enabled protocol developers to prove the correctness of such distributed protocols. However, model-checking techniques are only marginally useful during the actual development of such protocols; typically as a debugging aid once a reasonably complete version of the protocol has already been developed. The actual development process itself is often tedious and requires the designer to reason about complex interactions arising out of concurrency and asynchrony inherent to such protocols. In this dissertation we describe program synthesis techniques which can be applied as an enabling technology to ease the task of developing such protocols. Specifically, the programmer provides a natural, but incomplete description of the protocol in an intuitive representation — such as scenarios or an incomplete protocol. This description specifies the behavior of the protocol in the common cases. The programmer also specifies a set of high-level formal requirements that a correct protocol is expected to satisfy. These requirements can include safety requirements as well as liveness requirements in the form of Linear Temporal Logic (LTL) formulas. We describe techniques to synthesize a correct protocol which is consistent with the common-case behavior specified by the programmer and also satisfies the high-level safety and liveness requirements set forth by the programmer. We also describe techniques for program synthesis in general, which serve to enable the solutions to distributed protocol synthesis that this dissertation explores.

Contents

List of Tables

List of Figures

List of Algorithms

1

Introduction

Protocols for coordination among concurrent processes are an essential component of modern multiprocessor and distributed systems. The multitude of behaviors arising due to asynchrony and concurrency makes the design of such protocols difficult. Consequently, analyzing such protocols has been a central theme of research in formal verification for decades. Now that verification tools have matured to a point where they can be applied to find bugs in real-world protocols, a promising research direction is to develop and leverage *program synthesis* techniques as an enabling technology to simplify the design process of such protocols via more intuitive programming abstractions for specifying the desired behavior.

Traditionally, a distributed protocol has been modeled as a set of communicating processes, where each process is described as an extended state machine which has a finite number of *control states* or *locations*, along with a finite number of typed *state variables*. The correctness of a protocol is specified by both safety and liveness requirements. Model-checking techniques are then used to check that the protocol satisfies the safety and liveness requirements. In many cases, the model-checking algorithms are completely automatic. It is thus natural to ask if we can derive a correct protocol implementation starting from a set of safety and liveness requirements. And indeed, in *reactive synthesis* [RW89, PR89, BJP$^+$12], the goal is to automatically derive a non-distributed protocol, or a single reactive module, from its correctness requirements specified in temporal logic. However, if we require the implementation to be distributed, then reactive synthesis is undecidable [PR90, LT00, Tri04, FS05]. Furthermore, it is not clear that precisely codifying the behavior of the entire protocol using an intricate and complex formula in some temporal logic of choice is necessarily an easier or simpler task than specifying an operational description or an executable model of the protocol.

This dissertation proposes an alternative, and potentially more feasible approach inspired by *program sketching* [SLRBE05]. Our approach asks the programmer to specify the common case behavior of the protocol as a set of *incomplete* communicating processes, which may include some *unknown* functions. These unknown functions could be used in the *guards* for transitions — which describe the condition under which the transition in question can be executed — and the *update* functions to state variables on transitions — which describe how the state variables of the state machine evolve upon execution of the transition. The programmer could also provide some information on the missing behavior. This information might be in the form of input-output examples describing the behavior of the unknown functions, or could be information about exactly what behavior is left unspecified, for example, information about what kinds of messages need to be handled at a given point. The programmer would also have to state the *high-level* correctness requirements for the protocol in a temporal logic of choice.[1] The role of the synthesizer is to *complete* the incomplete protocol provided by the programmer, such that the completed protocol (a) satisfies the high-level correctness requirements set forth by the programmer, and (b) is consistent with the information provided by the programmer about the unspecified behavior. This methodology for protocol specification can be viewed as a fruitful collaboration between the designer and the synthesis tool: the programmer has to describe the structure of the desired protocol, but some details that the programmer is unsure about, for instance, regarding corner cases and handling of unexpected messages, are filled in automatically by the tool.

In our formalization of the synthesis problem, processes communicate using input/output channels that carry typed messages. Each process is described by a state machine with a set of typed state variables. Transitions consist of guards — that test some condition over the state variables — and updates to state variables and fields of messages to be sent. Such guards and updates can involve *unknown* (typed) functions to be filled in by the synthesizer. In many distributed protocols, such as cache coherence protocols, processes are expected to behave in a symmetric manner. Thus, we allow variables to have *symmetric types* that restrict the read/write accesses to obey symmetry constraints. To specify safety and liveness requirements, we allow the use of safety and liveness (or Büchi) monitors respectively. Finally, fairness assumptions are utilized to restrict incorrect executions to those that are *fair*. It is worth noting that in

[1]Note that these correctness requirements are typically much less detailed and simpler than the temporal logic formulae expected as input to typical reactive synthesis algorithms which attempt to synthesize a protocol purely from a specification in temporal logic.

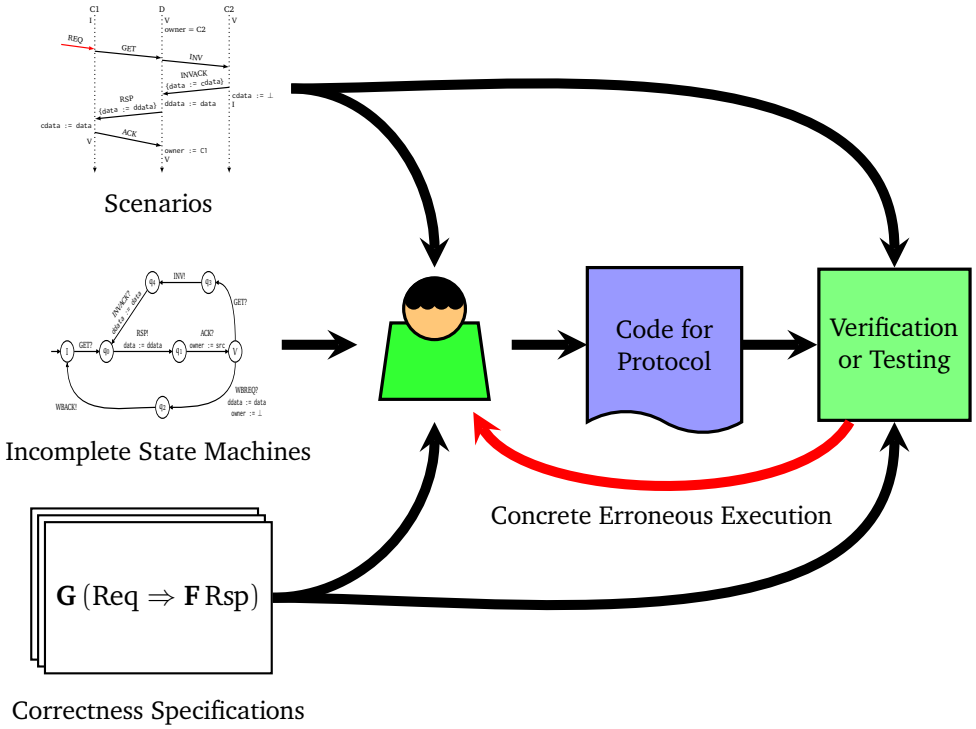


Figure 1.1: The traditional methodology for designing distributed protocols

verification one can get useful analysis results by focusing solely on safety requirements. In synthesis, however, ignoring liveness requirements and fairness assumptions, typically results in trivial solutions. The protocol completion problem, then, is, given a set of extended state machines with unknown guards and update functions, to find expressions for the unknown functions so that the composition of the resulting machines does not have an accepting fair execution.

In the rest of this chapter, we describe the design methodology we propose as part of this dissertation, as well as compare and contrast it with the traditional design methodology, by means of illustrative examples.

1.1 The Traditional Design Methodology

Figure 1.1 describes how distributed protocols are typically constructed. The programmer starts off with a variety of artifacts which describe the desired protocol. These can be in the form of *scenarios*, which describe the behavior of the protocol under specific use-cases, or as incomplete state machines which describe the *common-case* behavior of the protocol. In

addition, the programmer usually also has in mind some *high-level* correctness requirements that the protocol is expected to satisfy. These requirements could include *safety* requirements — which ensure that the protocol *never* does something "bad" — as well as *liveness* requirements — which ensure that the protocol *eventually* does something "good". The programmer then manually constructs an executable model (or implementation) of the protocol. This executable model can be described in various languages and formalisms, such as the Promela modeling language [Hol97], or the Murφ modeling language [ID96, Dil96], for example.

The resulting implementation is then checked for correctness using some combination of verification and testing techniques. For example, testing techniques could be used to check for correct behavior with respect to different scenarios specified by the programmer, whereas verification techniques could check that the candidate protocol satisfies the high-level safety and liveness specifications set forth by the programmer. In the event that an error is found during this check for correctness, the verification or testing framework provides the programmer with a concrete execution of the candidate protocol that demonstrates the error. The programmer then uses this information to *refine* or *correct* the behavior of the candidate protocol in the context of the specific counterexample currently under consideration. This process often requires the programmer to reason *globally* about the protocol to avoid introducing new errors as in the corrected version of the protocol. This tedious process of discovering bugs and correcting the protocol is iterated until a correct protocol is constructed. We now illustrate this process with a concrete example of how a simple cache coherence protocol might be constructed using this methodology.

1.1.1 The VI Cache Coherence Protocol

A cache coherence protocol ensures that all the processors in a multiprocessor system see a consistent view of data, despite the possibility that data values might be cached — and modified — by other processors in their local caches. Any coherence protocol essentially needs to ensure that the coherence property holds: The value *read* by any processor from a memory location must be the *most recent* value *written* to that memory location by any other processor in the system. Directory-based coherence protocols ensure that this property is maintained by using a centralized directory which is responsible for granting permissions to processors to read and write to memory locations. The processors and the directory then coordinate by exchanging messages with each other to acquire read and write permissions for memory locations in a

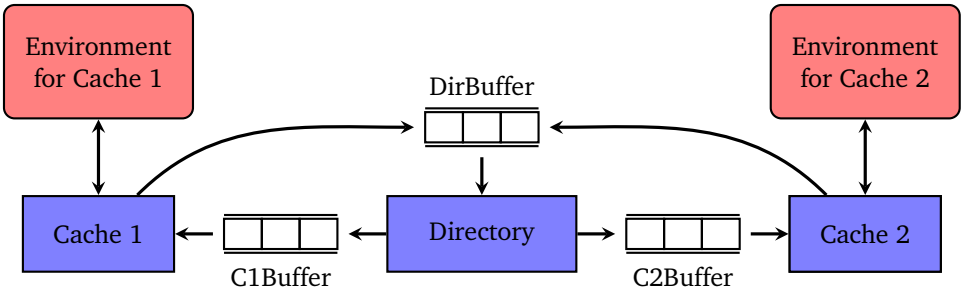
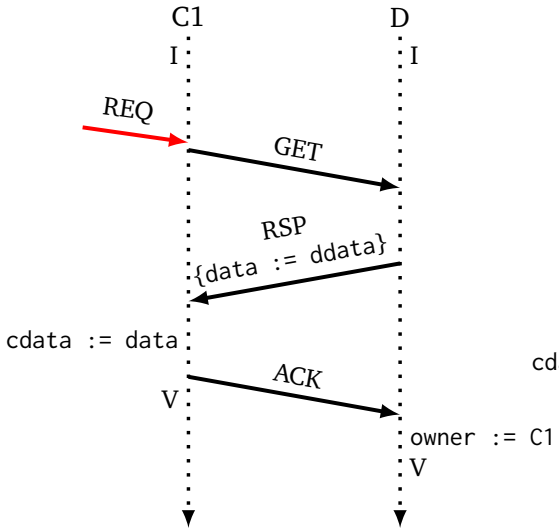


Figure 1.2: Communication Architecture of the VI Cache Coherence Protocol

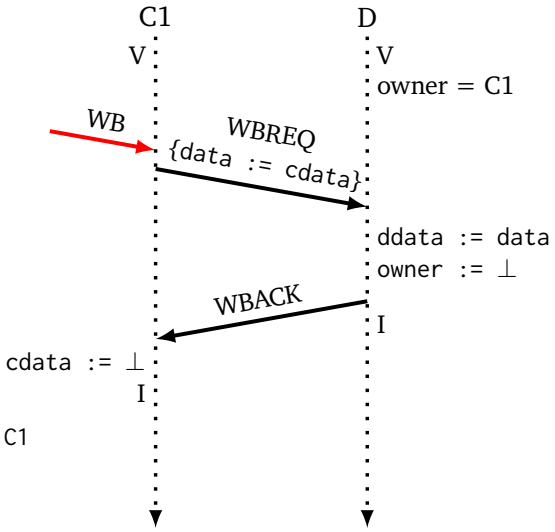
manner that does not violate the coherence property. For the purpose of illustrating how the traditional design methodology might be applied, we consider a simple cache coherence protocol called the Valid-Invalid (VI) protocol.

Figure 1.2 depicts the communication architecture of a variant of the VI cache coherence protocol, shown here with two cache state machines for clarity. The same architecture generalizes to an arbitrary number of cache state machines. The protocol consists of a state machine called the "Directory", which maintains the knowledge of which cache currently holds a cached copy of each data address. The caches communicate requests for access to a data block through the reliable, but unordered buffer named "DirBuffer". All communication in the protocol is buffered and asynchronous. The only exception is that communication between the caches and their respective environments occurs synchronously. The buffers have a finite size, but are sized to be large enough to ensure that no state machine ever blocks on a full buffer. The directory processes requests in the "DirBuffer" in an arbitrary order, and communicates commands to the caches by the buffers "C1Buffer" and "C2Buffer". The caches on their part, again process commands in an arbitrary order. Finally, we note that a state machine need not necessarily respond to all commands and requests at all points in time. In other words, a state machine is allowed to *defer* the processing of some command or request, which is in a buffer, until some condition has been enabled.

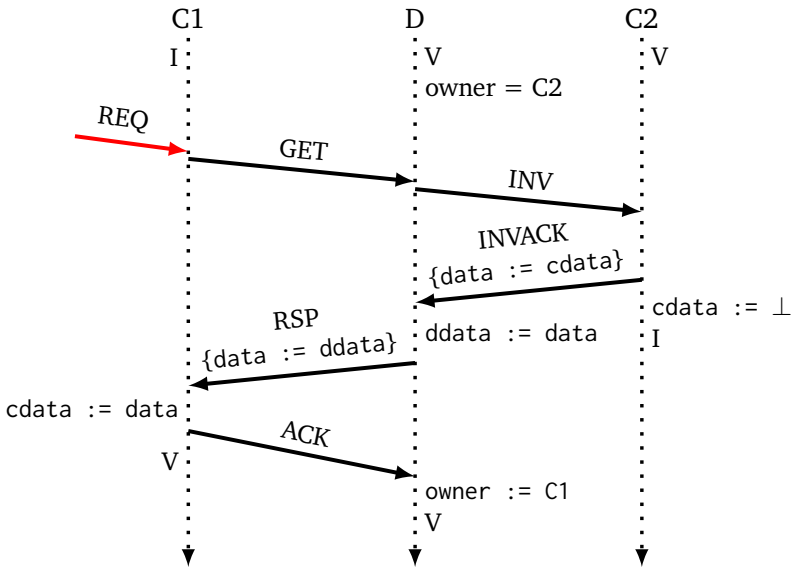
The working of the VI coherence protocol is perhaps best explained using the scenarios shown in Figure 1.3. The protocol consists of two classes of state machines: The cache controller state machines, whose behaviors are symmetric, denoted by `C1`, `C2`, ..., `Cn` in Figure 1.3, and a singular directory state machine, denoted by `D` in Figure 1.3. Each cache machine has a state variable named `cdata` which contains the cached value of data at any point, and can be undefined if the cache does not have an up-do-date cached copy of the data. The directory



(a) The first scenario for the VI protocol



(b) The second scenario for the VI protocol



(c) The third scenario for the VI protocol

Figure 1.3: The scenarios for the VI protocol

machine has two state variables: one named `ddata` which represents the most up-to-date value of the data when no cache in the system has a valid cached copy of the data; the other variable named `owner` denotes which cache (if any) currently contains the most up-to-date value of the data. The description of the VI coherence protocol is presented here in a slightly abstracted fashion for ease of understanding. In an actual implementation, the directory state machine would need a few more state variables to track the cache whose request is currently being

serviced. Note that the inputs from the environments for the caches are denoted by red arrows in Figure 1.3.

The first scenario shown in Figure 1.3(a) describes the how the protocol behaves when a cache requests ownership, *i.e.*, read and write permissions, and no other cache currently has ownership of the data block in question. In this situation, all the caches as well as the directory are in the *Invalid* state, denoted by I. In this situation, the directory itself is assumed to possess the most up-to-date copy of the data value. The cache requests the directory for access by sending the directory a GET message and the directory responds immediately, with a response message RSP, which contains the most up-to-date value of the data block, granting ownership to the cache. Following this the cache *unblocks* the directory by sending an acknowledgment message ACK. Upon receipt of the ACK message, the directory notes that the cache `C1` is now the *owner* of the block and transitions to the *Valid* state, denoted by V.

The second scenario shown in Figure 1.3(b) describes how a cache can *relinquish* ownership on a given data block. In this situation, the cache under question must own the data block, and thus it, along with the directory, must be in the *Valid* state denoted by V. The cache sends a WBREQ message containing the most up-to-date value of the data block to the directory. The directory updates its data block with the value received from the cache, and also notes that no cache currently owns the data block in question, by setting its `owner` state variable to be undefined. Following this, it responds with a WBACK message to the cache and transitions to the *Invalid* state, denoted by I. The cache, upon receipt of the WBACK message invalidates its local copy of the data, and also transitions to the *Invalid* state, denoted by I.

The third scenario, shown in Figure 1.3(c) describes how the protocol works when a cache requests ownership of a data block, but another cache already has ownership of the block. This situation is represented by the cache `C1` being in the *Invalid* state, and the directory as well as the cache `C2` being in the *Valid* state. From the perspective of cache `C1`, this scenario is the same as the one shown in Figure 1.3(a). However, the directory, upon receipt of the GET message sends an invalidation message INV to the cache `C2` which currently owns the data block. Upon receipt of the INV message, cache `C2` responds by sending an acknowledgment of invalidation, INVACK, which also contains the most up-to-date value of the data block, to the directory and transitions to the *Invalid* state. Its permissions have now been stripped by the directory. Upon receipt of the INVACK message from `C2`, the directory updates its local copy of the data. From this point on, the scenario proceeds in a manner similar to the scenario shown in Figure 1.3(a).

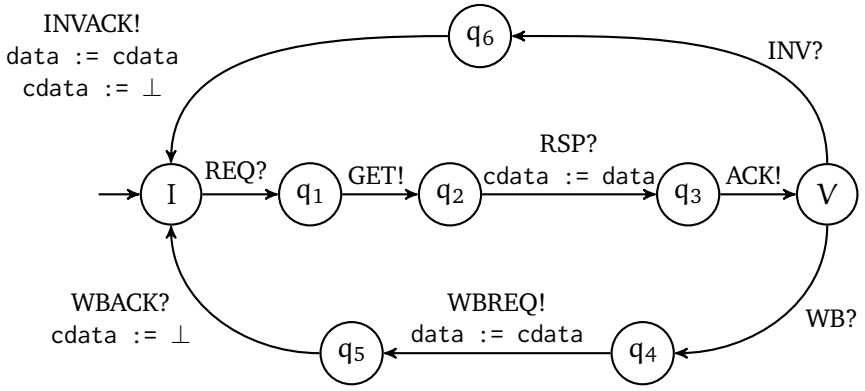


Figure 1.4: The incomplete state machine for the cache(s) in the VI protocol

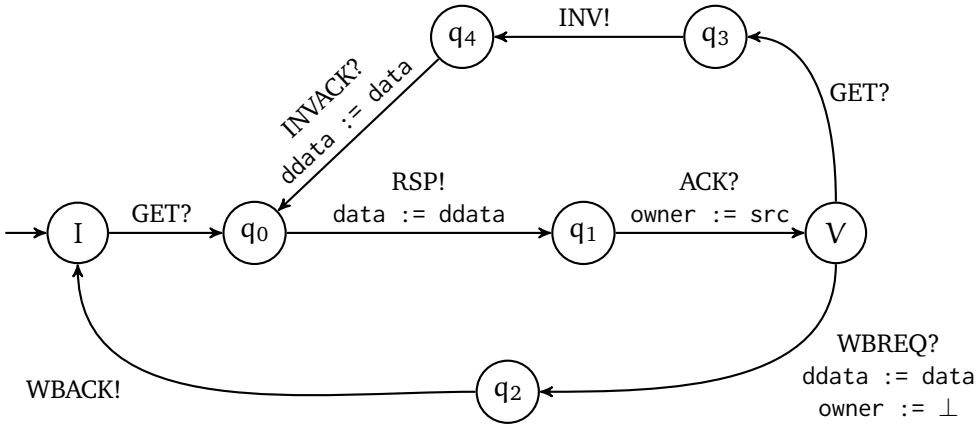


Figure 1.5: The incomplete state machine for the directory in the VI protocol

In addition to the coherence property — which all cache coherence protocols ought to satisfy — it is desirable that each cache coherence protocol satisfies a set of *liveness* requirements, to ensure progress. In the case of the VI cache coherence protocol, the intuitive liveness requirement is that a GET request from every cache eventually results in the receipt of an RSP message with the most up-to-date value of the data by the cache that has issued a GET request. Additionally, a similar liveness requirement is also desirable with respect to the WBREQ request, which must eventually result in the receipt of a message that results in the cache transitioning to the *Invalid* or I state.

1.1.2 Designing Distributed Protocols: The Easy Parts

Based on the three common-case scenarios shown in Figure 1.3, the programmer constructs the state machines for the caches and the directory as shown in Figures 1.4 and 1.5 respectively.

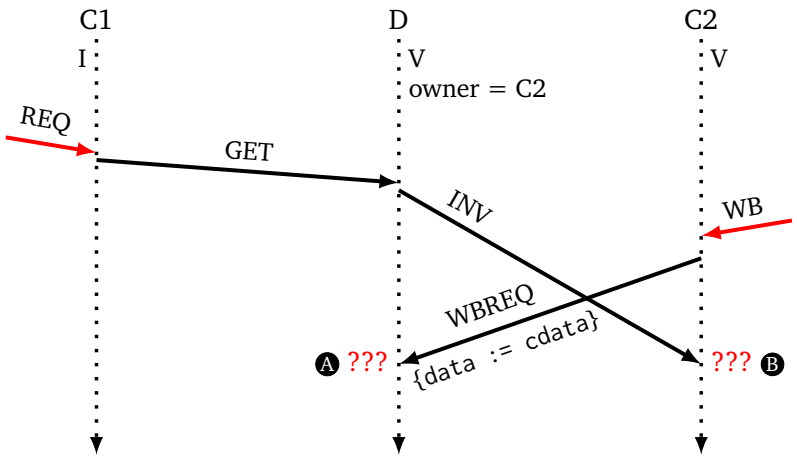


Figure 1.6: A scenario implied by the scenarios shown in Figure 1.3 in the VI protocol

This translation is rather straight-forward and can even be automated in some cases, as we shall discuss in Chapter 7. Upon attempting to verify the correctness of the protocol described by the state machines in Figures 1.4 and 1.5, a verification tool presents the execution shown in Figure 1.6 as a counterexample which results in a deadlock.

The execution shown in Figure 1.6 occurs as a result of the interleaving of the scenarios shown in Figure 1.3(b) and Figure 1.3(c). Specifically, the state machines `C1` and the directory are proceeding according to the scenario shown in Figure 1.3(c). The cache `C2` does not have any knowledge about the state of other state machines in the system and proceeds according to the scenario shown in Figure 1.3(b) upon receiving a WB command from its environment. This results in a deadlock. the directory state machine is expecting an INVACK message, but instead receives a WBREQ message. The state machine for the cache `C2` is expecting a WBACK message, but instead receives an INV message. Neither the cache nor the directory state machines have a transition which describes what needs to happen in this circumstance, thus resulting in a deadlocked protocol. Note that it is impossible to avoid this situation, owing to the distributed nature of the protocol. This naturally leads us to a discussion about the difficult parts of designing a protocol using the traditional methodology.

1.1.3 Designing Distributed Protocols: The Difficult Parts

To specify the correct behavior in the specific scenario shown in Figure 1.6, the programmer needs to handle the behaviors in the state machines shown using dashed transitions to an unknown target state in Figures 1.7 and 1.8. Note that the locations labeled Ⓐ and Ⓑ in

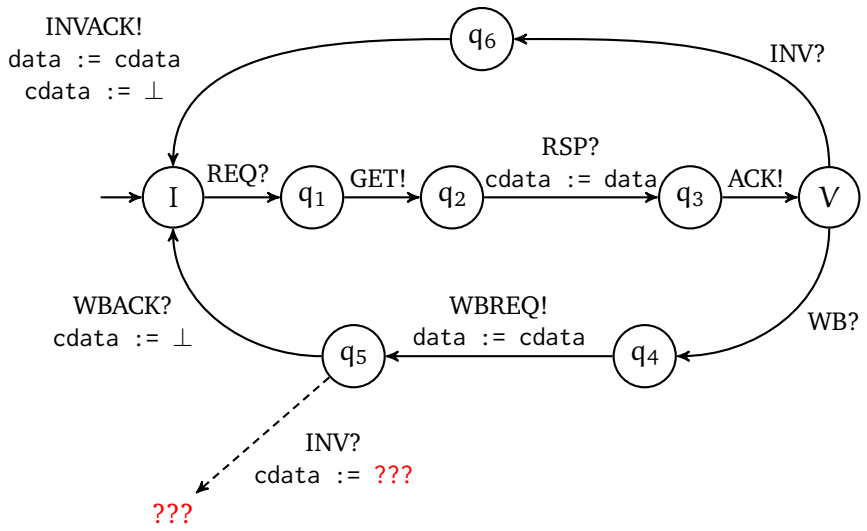


Figure 1.7: Unhandled behavior in the cache state machine for the VI protocol

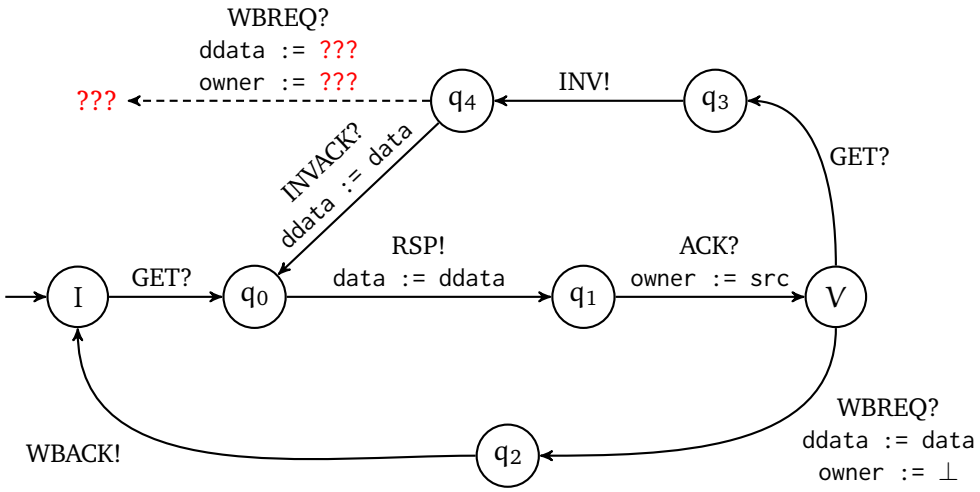


Figure 1.8: Unhandled behavior in the directory state machine for the VI protocol

Figure 1.6 correspond to the state q_4 in the directory state machine and the state labeled q_5 in the cache state machine respectively. The transitions represented by the dashed arrows thus represent the new transitions that must be added to eliminate the deadlock in the execution shown in Figure 1.6.

To eliminate the deadlocking execution, the programmer now needs to answer the following correlated questions:

1. Which state must the cache state machine transition to upon receipt of an INV message in the state q_5?

2. How must the `cdata` state variable of the cache state machine be updated along this transition?
3. Which state must the directory state machine transition to upon receipt of a WBREQ message in the state q_4, given the choice made earlier for the transition of the cache machine upon receipt of the INV message in state q_5?
4. How must the `ddata` and `owner` state variables of the directory state machine be updated along this transition, again taking into consideration all the choices made so far in the process of correcting the protocol.

Clearly, the right answers to these questions are correlated. Thus the programmer is forced to perform some form of global reasoning about the protocol to describe the correct behavior. Further, the programmer may need to perform this kind of reasoning multiple times as additional erroneous executions are discovered. We argue that this process is rather tedious and contributes significantly to the difficulty of designing correct implementations of distributed protocols. We now present an alternative methodology which makes use of program synthesis techniques to make the process of designing distributed protocols easier.

1.2 An Alternative Approach to Protocol Design

Given the undecidability of the problem of synthesizing distributed protocols purely from temporal logic specifications, we view the synthesis problem as one of *completion* in this dissertation. This section provides an intuitive description of how this view can help in making the difficult parts of protocol design easier.

1.2.1 Automating the Difficult Parts of Protocol Design

Figure 1.9 provides a high-level overview of the approach we propose in this dissertation. The programmer specifies the behavior of the protocol using a combination of common-case scenarios (which can be easily translated to incomplete state machines, either automatically, or manually) and incomplete state machines constructed from the well-understood common-case behavior of the protocol in question. Our approach also requires that the programmer formally specifies the high-level safety and liveness requirements that the protocol is expected to satisfy. We then leverage synthesis techniques to *complete* this incomplete protocol, or add behaviors to this incomplete protocol provided by the programmer to obtain an implementation which is *correct by construction*, *i.e.*, the completed protocol admits *at least* all the behaviors admitted

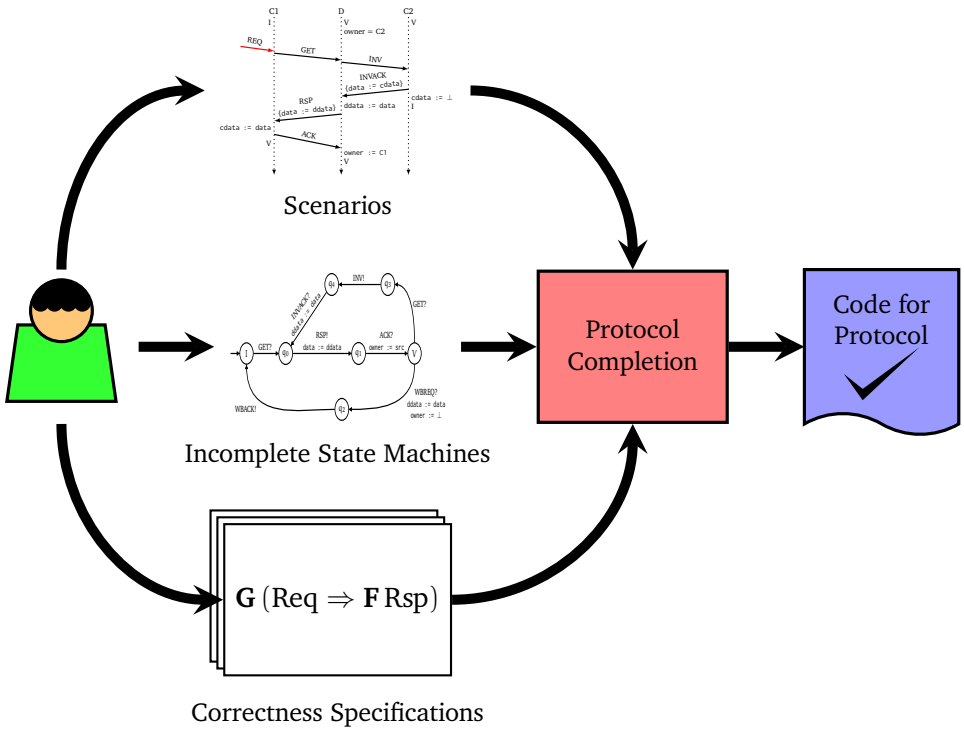


Figure 1.9: The methodology proposed in this dissertation for designing distributed protocols

by the incomplete protocol specified by the programmer, and satisfies all the high-level safety and liveness requirements.

This view of synthesis as a completion problem yields two advantages. First, the programmer is freed from the tedium and complexity of the iterative debugging process, and has only to specify an incomplete protocol, which is relatively easy. Second, we side-step the undecidability of distributed protocol synthesis. The completion problem is itself decidable, provided that the domains of all the state variables are finite. The decidability results from the fact that the completion process does not attempt to add new control states or variables. Obviously, this decidability comes at a cost: to be useful, the programmer needs to provide a "reasonably complete" version of the protocol, *i.e.*, it must be possible to obtain a correct protocol from the incomplete protocol provided by the programmer, without the addition of new control locations or state variables to the state machines in the protocol.

The upshot is that viewing the problem as one of *completion*, rather than synthesis, allows us to leverage the fact that the easy bits of distributed protocol design can be done by the programmer, to develop effective and useful algorithms that alleviate the difficulty of designing

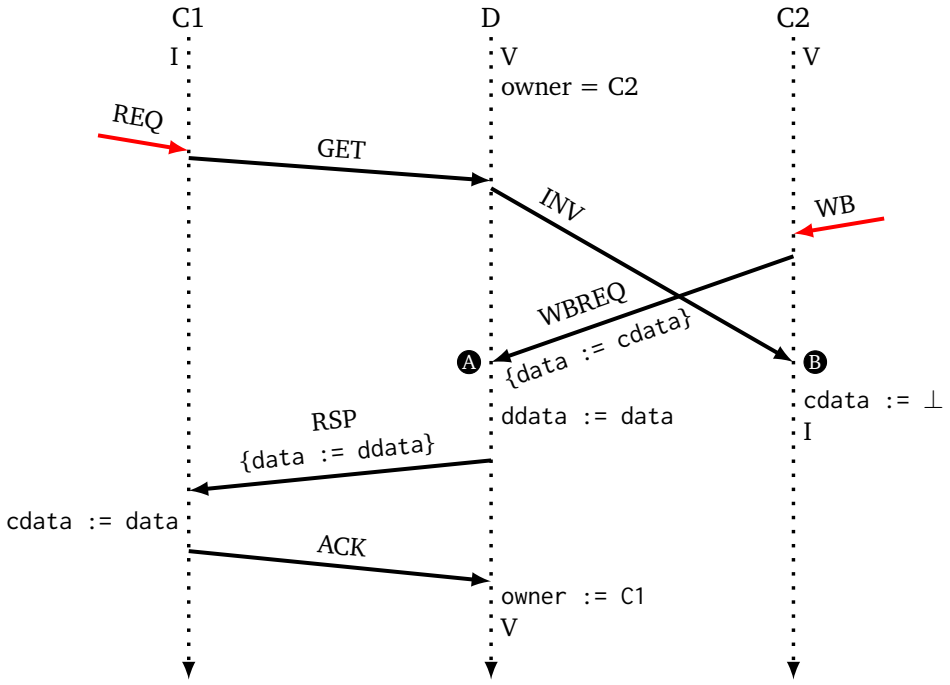


Figure 1.10: A possible completion of the implied scenario in the VI protocol

distributed protocols. This trade-off between programmer involvement and effectiveness of automated algorithms for completion is a theme that we will explore throughout the subsequent chapters in this dissertation.

Turning our attention back to the example of the VI cache coherence protocol, Figure 1.10 shows one possible way in which the implied scenario shown in Figure 1.6 can be extended, such that all the correctness properties are satisfied. Essentially, the directory, treats the WBREQ message in the same manner as it would treat an INVACK message from the cache, and updates its local copy of the data with the value contained in the WBREQ message. The rest of the scenario plays out between the directory and cache `C1` as shown in Figure 1.3(c). The cache `C2`, on its part, treats the INV message in the same manner as a WBACK message, invalidating its local copy of the data and transitioning to the *Invalid* or I state.

Figures 1.11 and 1.12 show the cache and directory state machines, respectively, completed according Figure 1.10. Viewed as a completion problem, the algorithms described in subsequent chapters of this dissertation are able to synthesize the state machines shown in Figures 1.11 and 1.12, starting from the incomplete state machines shown in Figures 1.4 and 1.5, along with a set of high-level safety and liveness requirements.

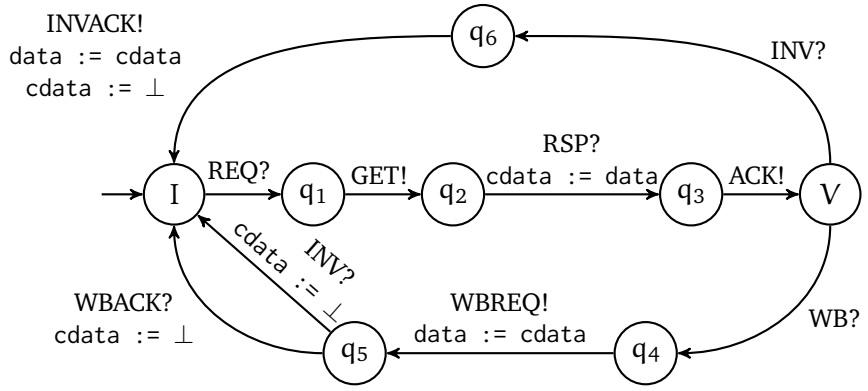


Figure 1.11: The completed state machine for the cache in the VI protocol

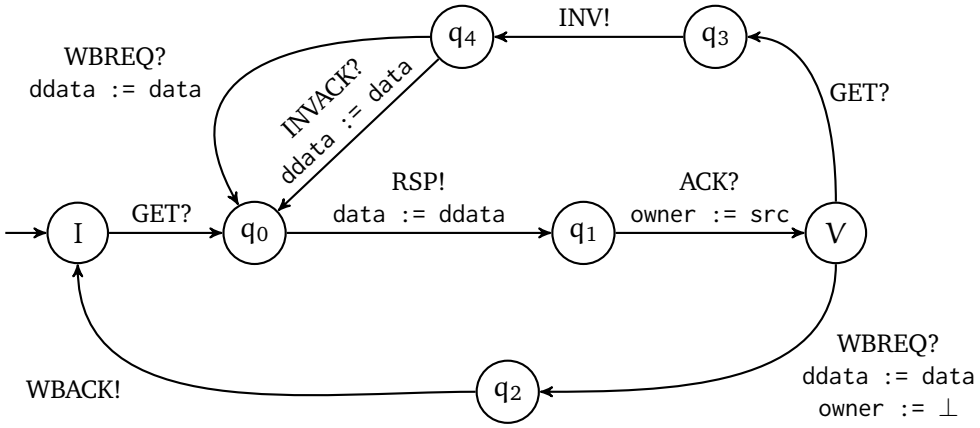


Figure 1.12: The completed state machine for the directory in the VI protocol

Note that although the state machines for the VI protocol did not have any *guards* in their transitions, this may not be the case in general. Some protocols might consist of state machines which require transitions to be executed *conditionally*, based on some predicate on the state variables of the machine in question. In the case of the VI protocol, these predicates can be viewed as being universally `true`. In general, a completion algorithm would need to determine these predicates, in addition to determining the target state and the updates to state variables along a transition.

1.2.2 Feasibility and Effectiveness of Protocol Completion

The utility of the design methodology for distributed protocols that we have just introduced, depends heavily on whether it is feasible to build tools that support the methodology and on the effectiveness of such tools. Objectively, the proposed design methodology for distributed

protocols can be considered useful, provided that the answers to the following two questions can be proven to be in the affirmative:

- Is it possible to develop effective algorithms to solve the distributed protocol completion problem? A related question is whether these algorithms can be useful in assisting a protocol designer in developing correct versions of real world protocols which are beyond the capabilities of traditional approaches to reactive synthesis.
- Is it easier for a protocol designer to specify the behavior of the protocol using a combination of scenarios and incomplete state machines, along with a set of high-level formal requirements in temporal logic?

Detailed experimental evaluations in the subsequent chapters of dissertation demonstrate that the answer to the first question is indeed in the affirmative. The second question, on the other hand, is rather subjective. While a large scale user study is beyond the scope of this dissertation, we hope that the examples provided in this chapter, as well as in subsequent chapters, serve to convince the reader that it is indeed easier to specify distributed protocols using the approach that we propose.

To demonstrate affirmative answers to the questions just raised, we built and evaluated several prototype tools. We now present a brief summary of the capabilities of each tool. A more thorough exposition will be provided in subsequent chapters.

The first tool we built, dubbed TRANSIT, required the protocol designer or programmer to be a part of the synthesis loop. The programmer was expected to provide *local* remedies to specific, concrete erroneous executions uncovered during the verification process. Our case studies demonstrate that this was useful in reducing the tedium of the debugging phase of protocol design. In each case, the programmer was able to obtain a correct protocol, with only a few rounds of interaction with the completion tool. The approach was found to be very scalable, and was successfully used to specify the industrial SGI Origin cache coherence protocol [LL97]. This is a large, scalable, real life protocol, with millions of reachable states, and has been deployed in high-end systems from SGI.

Encouraged by the success of TRANSIT, we now sought to automate the process, and free the programmer from being part of the synthesis loop. The second tool which we developed only handles protocols where the state machines do not have any state variables. In this specific setting, the completion problem can be viewed as a minimal Boolean satisfiability problem, which in turn can be solved effectively by Integer Linear Program (ILP) solvers. Another feature

of this tool was it accepted inputs in the form of scenarios, which are as shown in Figure 1.3, rather than as incomplete state machines. This tool was able to automatically complete several text-book protocols, such as the alternating-bit protocol, protocols for consensus and even a simple cache coherence protocol.

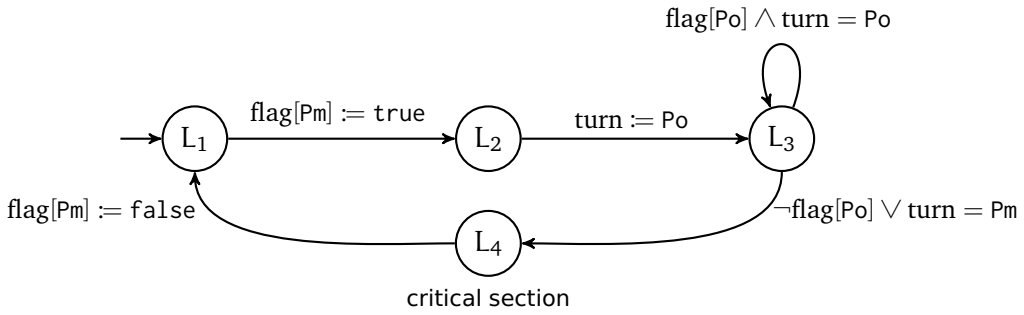
The limitation of this scenario based completion tool was not scalability: it synthesized everything we threw at it with ease. However, it was rather difficult to specify larger protocols with the restriction that state machines not have any variables. The third tool which we built addresses this limitation, and also allows the programmer to specify symmetry constraints that a completed protocol ought to satisfy. Using this tool, we were able to automatically synthesize protocols for mutual exclusion, a moderately sized self-stabilization protocol, as well as the modestly complex German/MSI cache coherence protocol.

While the automated algorithms are not as scalable as the algorithms which require the programmer to be a part of the synthesis loop, they can still be useful in developing protocols of moderate complexity. The experimental evaluations in subsequent chapters of this dissertation will explore this trade-off between programmer involvement and scalability in greater depth.

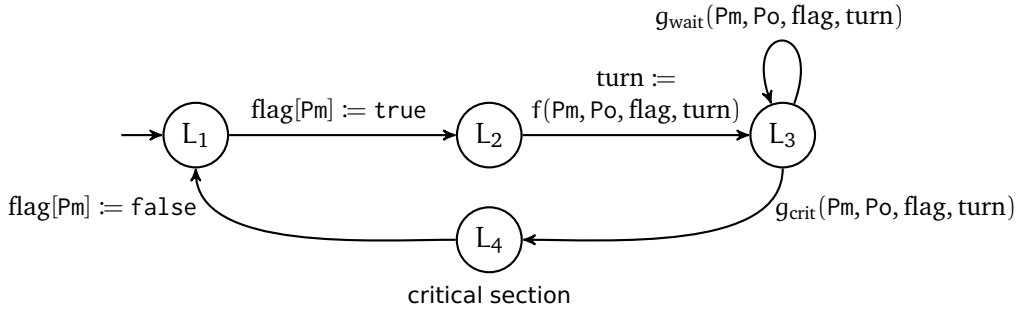
1.2.3 Protocol Completion as Synthesis of Interpretations

Throughout this dissertation, we will take the view that the protocol completion is tantamount to the problem of synthesizing *interpretations* for multiple, possibly correlated, unknown functions. To make this view apparent, observe that the *guard* for each new transition to be added can be viewed as a Boolean valued function over the state variables of the state machine in question. Similarly, the *updates* to each state variable along a transition to be added can be viewed as a function of the appropriate type over the state variables. Lastly, the target control state to transition to can also be viewed as an update to a distinguished state variable — say a variable named "location" — of a suitable enumerated type.

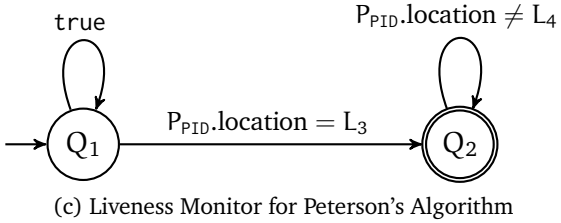
To illustrate this view, as well as to describe the subtleties of symmetry and fairness in some detail, we consider another example: the Peterson's mutual exclusion protocol. Figure 1.13(a), describes this protocol, which manages two symmetric processes contending for access to a critical section, labeled as the state L_4 in Figure 1.13. Each process is parameterized by two parameter variables `Pm` and `Po` (for "my" process id and "other" process id respectively), such that `Pm` $\neq$ `Po`. Both the parameters `Pm` and `Po` are of type `processid`, which is a *symmetric type*, and they are allowed to take on values `P0` and `P1`. We therefore have two instances of the



(a) Parameterized Symmetric Process



(b) Incomplete Process Sketch



(c) Liveness Monitor for Peterson's Algorithm

Figure 1.13: Peterson's Mutual Exclusion Protocol

symmetric process shown in Figure 1.13(a): P_0, where $(\texttt{Pm} = \texttt{P0}, \texttt{Po} = \texttt{P1})$, and P_1, where $(\texttt{Pm} = \texttt{P1}, \texttt{Po} = \texttt{P0})$. P_0 and P_1 communicate through the shared variables *turn* and *flag*. The variable *turn* has type processid. The *flag* variable is an array of Boolean values, indexed by values of the type processid. The main objective of the protocol is to control access to the critical section, represented by location L_4, and ensure that both of the processes P_0 and P_1 are never simultaneously in the critical section, *i.e.*, it is a safety violation for both P_0 and P_1 to be in state L_4 at the same time. For clarity, the assignments to *flag* and *turn* are shown as simple assignments in Figure 1.13, but in a faithful model of the Peterson's algorithm, these would be involve exchange of messages, with the shared variables *flag* and *turn* represented as state machines for atomic registers.

The liveness monitor shown in Figure 1.13(c) captures the requirement that a process not wait indefinitely to enter the critical section. The liveness monitor is itself parameterized by the parameters Pm and Po in a manner similar to the processes, with each instance encoding the liveness requirement for the appropriate process. The monitor accepts all undesirable runs where a process has requested access to the critical section (*i.e.*, the process in question is in state L_3), but never reaches state L_4 — which corresponds to entering the critical section — after reaching state L_3. In other words, the monitor accepts an infinite execution where one of the processes P_0 or P_1 is stuck in state L_3 forever. Note that a run accepted by the monitor may be *unfair* with respect to some processes. For instance, if process P_0 is in state L_3 and could possibly transition to state L_4, but the scheduler never schedules process P_0, and process P_0 therefore never enters state L_4, then this execution is unfair with respect to process P_0. Enforcing *weak* process fairness on P_0 and P_1, — *i.e.*, if a process is enabled at every point in an infinite execution, then it must be executed at some point in that execution — is sufficient to rule out unfair executions, but not necessary. Enforcing weak fairness on the single transition between (L_3, L_4) suffices to rule out all unfair executions which could possibly be accepted by the monitor shown in Figure 1.13(c).

Now, to view the protocol completion problem as one of synthesizing interpretations, consider the *incomplete* version of Peterson's mutual exclusion protocol as shown in Figure 1.13(b). Here, the condition under which a process is allowed to enter the critical section, the condition under which a process must wait in location L_3, and the update to the *turn* variable along the edge from L_2 to L_3 have been replaced by the unknown functions g_{wait}, g_{crit} and f respectively. The target control locations along these transitions could also be unknown, but are retained here for clarity.[2]

The functions g_{wait} and g_{crit} represent unknown Boolean valued functions over the state variables and the parameters of the process under consideration. The function f represents the unknown update to the *turn* variable. Including the parameter variables Pm and Po as part of the domain of g_{wait}, g_{crit} and f is necessary to ensure that the completions synthesized by a tool for processes P_0 and P_1 are symmetric. We defer a formal definition of what it means for protocols and interpretations to be symmetric until Chapter 2. Now, given a set of fairness assumptions, the protocol completion problem reduces to automatically discovering interpretations for these unknown functions, such that the completed protocol satisfies the necessary mutual exclusion

[2]It would be rather confusing to see transition arrows all shooting up into nowhere.

property, and that every *fair* execution of the completed protocol is not an accepting run of the liveness monitor shown in Figure 1.13(c).

Thus the protocol completion problem can indeed be viewed as a problem of synthesizing interpretations for a set of correlated unknown functions. This provides an excellent segue for the next section in this chapter, which motivates the development of a general framework for describing such problems, independent of distributed protocols.

1.3 A Framework for Function Synthesis

Although synthesis of distributed protocols is the primary focus of this dissertation, during course of research on this topic, we also had to develop scalable program synthesis techniques to enable the synthesis of distributed protocols. As we have just explained, synthesizing *guard* and *update* functions that constitute the descriptions of state machines in a distributed protocol is tantamount to the synthesis of multiple unknown functions, the constraints on the behavior of which can possibly be correlated. We observed that a lot of recent work on program synthesis for different domains were essentially solving this very same problem — *i.e.*, that of synthesizing an unknown function (or a set of unknown functions) such that the synthesized function satisfies some constraints — with a variety of independently developed (and possibly domain-specific) algorithms. Unfortunately, there was no uniform way to compare the strengths and weaknesses of each of these algorithms due to subtle differences in the way each of them required the constraints over the unknown functions to be expressed, as well as the search space for the interpretations or bodies of these unknown functions.

This led us to formulate the Syntax-Guided Synthesis (SyGuS) problem, which is intended to be a general framework — along with a specification language — for expressing program synthesis problems. The motivation for this was two-fold:

- Provide a common input language for specifying the constraints on, and the search space for candidate interpretations or bodies of unknown functions to be synthesized. This format, called SyGuS-IF, can then serve as a common input language for describing benchmarks to evaluate tools implementing different program synthesis techniques.
- Spur research in program synthesis by organizing annual SyGuS competitions, in the same manner that the SMTLIB and SMTCOMP initiatives have helped encourage research in satisfiability modulo theory (SMT) solvers and theorem provers.

The intention is for SyGuS to be to program synthesis, what SMTLIB is to program verification.

This dissertation includes a description of the SyGuS problem as well as a description and evaluation of two SyGuS solvers which implement algorithms based on enumerative strategies — *i.e.*, algorithms which systematically, and intelligently, enumerate function interpretations or bodies from the search space until a solution is found — to solve instances of the SyGuS problem. The SyGuS solvers may be considered as technologies which enable the construction of higher-level and domain specific synthesis algorithms — such as algorithms for distributed protocol completion and synthesis.

1.4 Contributions of this Dissertation

Having introduced the specific problems that the research to be described in this dissertation was intended to tackle, we now provide a short summary of contributions made by this dissertation:

- We demonstrate that although the problem of full distributed reactive synthesis from temporal logic specifications is hard, useful assistance can still be provided to the designer/developer of such protocols by viewing the synthesis problem as one of *completion*. In this world-view, the developer assists the synthesis algorithm by providing the information which is natural and easy for a developer to provide. The tool in turn provides as much automation as possible to the developer in automatically discovering the parts of the protocol which the developer finds difficult to reason about.
- We describe and evaluate three tools which we have developed, which aim to make the process of developing distributed protocols easier. Each of these tools differ in the level of automation provided — and thus in their performance and scalability — and in the restrictions they impose on the kinds of protocols they can handle.
- In evaluating and comparing the abilities of these tools, we explore the three-way trade-off between the level of automation provided by the tools versus the amount of developer involvement in the process of developing these protocols versus the scalability of the tools.
- We describe the SyGuS framework for specifying program synthesis problems in a general manner, which is intended to be an enabling technology for higher-level synthesis techniques to build upon. We also describe and evaluate algorithmic strategies based on systematic and intelligent enumerative search to solve instances of the SyGuS problem.

The subsequent chapters of this dissertation are organized as follows. Chapter 2 introduces some notation and definitions and provides a formal definition of the protocol completion

problem. Chapter 3 describes an elegant symbolic solution strategy for the protocol completion problem, and explains why the strategy is not effective in a practical setting. Chapter 3 also describes the insights gained from implementing and experimenting with the symbolic algorithm, which motivated the choices made with respect to the rest of the work described in this dissertation. Chapters 4, 7 and 8 describe the solution approaches to the protocol completion problem that we have implemented and evaluated. Chapter 5 describes a general-purpose framework for program synthesis, called SyGuS, that arose from the work described in Chapter 4, and Chapter 6 discusses some enumerative strategies for solving instances of the SyGuS problem. Chapter 9 provides an overview of related work in the area and discusses how the work described in this manuscript differs from earlier work. Chapter 10 summarizes the contributions of this dissertation and discusses the avenues along which the work may be extended in the future, and concludes with some reflections on the problems addressed in this dissertation.

2

The Protocol Completion Problem

Having informally described and motivated the protocol completion problem in the previous chapter, we now present a rigorous definition of the problem and set up the prerequisite definitions and notation which will be used in the rest of this dissertation.

2.1 Objective

Our primary objective is to ease the task of developing correct distributed protocols. To accomplish this, we leverage the fact that it is often easy to specify the behavior of a distributed protocol in the common cases. We allow the developer to specify a *skeleton* or *sketch* of the protocol that defines (i) the set of communicating processes that make up the protocol (ii) the *state variables* of each communicating process that is part of the protocol, (iii) the *communication architecture* of the protocol, which defines which processes can communicate, and along which direction, as well as describes the properties of communication links between processes, (iv) the behavior of the protocol in the common case scenarios, (v) the correctness properties, in the form of *invariants* and *liveness monitors*, and (vi) a set of *fairness* requirements under which the liveness properties ought to hold. Collectively, these artifacts describe an *incomplete* protocol. We will assume that the incomplete protocol, by itself, does not satisfy the desired correctness properties. The goal then, is to complete this incomplete protocol, by adding transitions where required, such that the completion satisfies the desired correctness properties.

2.2 Formalization and Notation

We now formally define our notion of a state machine, executions of state machines, composition of state machines and other related notions in this section. Our formalism draws on the notion of

input-output automata (I/O automata) described in the text-book by Lynch [Lyn96]. However, we do not assume that the state machines (or I/O automata) are *input-complete*, *i.e.*, are required to handle *any* input at *any* point in their execution.

2.2.1 Types

Let $\mathcal{B}$ be a set of *base types*, where each type $\mathsf{T} \in \mathcal{B}$ has finite cardinality, and is either (1) the Boolean type, (2) an enumerated type, (3) a fixed range integer type, or (4) a *symmetric* type. Symmetric types are similar to enumerated types, but the behavior of the system is considered to be invariant under permutations of the symmetric type. The notion of a symmetric type will be described in more detail in Section 2.2.7. Given a type $\mathsf{T}_1 \in \mathcal{B}$, and a type T_2, the composite type $\mathsf{array}(\mathsf{T}_1, \mathsf{T}_2)$ contains all mappings from values of type T_1 to values of type T_2. Given types $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_n$, the composite type $\mathsf{record}(\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_n)$ denotes a type whose values range over $\mathsf{T}_1 \times \mathsf{T}_2 \times \cdots \times \mathsf{T}_n$. Given a fixed set of base types $\mathcal{B}$, we define $\mathcal{T}_\mathcal{B}$ to be the smallest set of types such that (1) $\mathcal{T}_\mathcal{B} \supseteq \mathcal{B}$ and, (2) $\mathcal{T}_\mathcal{B}$ is closed under composition using the array and record operators, *i.e.*, if $\mathsf{T}_1 \in \mathcal{B}$, and $\mathsf{T}_2 \in \mathcal{T}_\mathcal{B}$, then $\mathsf{array}(\mathsf{T}_1, \mathsf{T}_2) \in \mathcal{T}_\mathcal{B}$, and if $\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_n \in \mathcal{T}_\mathcal{B}$, then $\mathsf{record}(\mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_n) \in \mathcal{T}_\mathcal{B}$. We drop the subscript and use $\mathcal{T}$ to refer to $\mathcal{T}_\mathcal{B}$ whenever the context is clear. Note that every type $\mathsf{T} \in \mathcal{T}_\mathcal{B}$ has finite cardinality.

2.2.2 Function Symbols

Given a set of types $\mathcal{T}$, we fix a set of function symbols $\mathfrak{F}$. Each function symbol $f \in \mathfrak{F}$ has a signature denoted $d_1 \times d_2 \times \cdots d_n \to r$, where $d_1, d_2, \ldots, d_n \in \mathcal{T}$ represent the domain of the function and $r \in \mathcal{T}$ represents the range of the function. A function symbol may have a fixed interpretation, e.g., the symbol '+' might denote integer addition, or the interpretation may be unknown. We denote by $\mathfrak{U} \subseteq \mathfrak{F}$, the subset of the function symbols in $\mathfrak{F}$ whose interpretations are unknown. We define an *expression* to be a well-typed composition of function symbols applied to values or variables of the appropriate types. Further, we assume that *exactly one* state machine uses any given unknown function in its description. Thus for each unknown function $f_u \in \mathfrak{U}$, we can speak about *the* state machine that uses f_u in its description.

2.2.3 Messages

We define Σ to be a message alphabet and $\mathsf{mtype} : \Sigma \to \mathcal{T}$ to be a function that maps each message $m \in \Sigma$ to the type of its *payload*. Further, we define Σ_P to be a set of *parame-*

terized messages. A parameterized message has the form $m\langle p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n\rangle$, where $T_1, T_2, \ldots, T_n \in \mathcal{T}$ are symmetric types, and $p_1, p_2, \ldots, p_n$ are parameter variables whose values can range over $T_1, T_2, \ldots T_n$ respectively. For every parameterized message m of the form $m\langle p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n\rangle \in \Sigma_P$, the corresponding *instances* of the message m are in Σ. *i.e.*, $m\langle p_1 \mapsto v_1, p_2 \mapsto v_2, \ldots, p_n \mapsto v_n\rangle \in \Sigma$, for all values $v_1, v_2, \ldots, v_n$, where $v_1 \in T_1, v_2 \in T_2, \ldots, v_n \in T_n$. Further, any two instances of a parametric message have the same payload type. Specifically, if m is a parameterized message as described above, then, we have that $\mathsf{mtype}(m\langle p_1 \mapsto u_1, p_2 \mapsto u_2, \ldots, p_n \mapsto u_n\rangle) \equiv \mathsf{mtype}(m\langle p_1 \mapsto v_1, p_2 \mapsto v_2, \ldots, p_n \mapsto v_n\rangle)$ for all $\{u_i\}$ and $\{v_i\}$. Parameterized messages themselves cannot be used as inputs to, or outputs of state machines, but instances of parameterized messages can. Instances of parameterized messages have restrictions on being inputs and outputs of state machines as we will describe shortly.

2.2.4 Extended State Machines

An extended state machine (ESM) A is a tuple $A \triangleq \langle L, l_0, I, O, V, \sigma_0, R, \mathcal{F}_s, \mathcal{F}_w\rangle$, where:

- L is a finite set of *locations*.
- l_0 is the *initial* location at which every execution of the ESM begins.
- $I \subseteq \Sigma$ is a set of *input* messages.
- $O \subseteq \Sigma$ is a set of *output* messages, such that $I \cap O = \emptyset$.
- V is a finite set of typed state variables. For notational convenience, we define the typing function $\mathsf{typeof} : V \to \mathcal{T}$ which maps each variable to its type.
- σ_0 is an *initialization* function that maps each variable $v \in V$ to an initial value $s \in \mathsf{typeof}(v)$.
- $R \triangleq R_i \cup R_o \cup R_\epsilon$ is a set of transitions, partitioned into a set of *input* transitions R_i, a set of *output* transitions R_o and a set of *internal* transitions R_ϵ. Each transition $r \in R$ is of the form $r \triangleq \langle l, m, \mathsf{guard}, \mathsf{updates}, l'\rangle$, where $l, l' \in L$ are the initial and final locations for the transition; $m \in I$ for input transitions, $m \in O$ for output transitions, and $m = \epsilon$ for internal transitions; guard is a Boolean valued expression over the state variables V, and updates maps each lvalue under consideration to an update expression of the appropriate type. If $r \in R_i$, then updates maps each lvalue $v \in V$ to an update expression which may only refer to the variables in $V \cup \{m_p\}$, where $m_p \notin V$ refers to the payload of the incoming message m. If $r \in R_o$, then updates maps each lvalue $v \in V \cup \{m_p\}$ to an update expression, where $m_p \notin V$ is the payload of the outgoing message, and the update expressions may refer only

to variables in V. Finally, if $r \in R_\epsilon$, then updates maps each lvalue $v \in V$, to an update expression which may only refer to variables in V.

- $\mathcal{F}_s, \mathcal{F}_w \subseteq 2^{R_o \cup R_\epsilon}$ are sets of subsets of the transitions which characterize strongly and weakly *fair executions* of the state machine A respectively.

Note that the guard and update expressions in the description of an ESM might involve functions whose interpretations are unknown. If the description of an ESM contains at least one occurrence of a function symbol $f_u \in \mathfrak{U}$ in any of its guards or in any of its update functions, then we call such a description an ESM sketch (ESM-SK).

2.2.5 Executions

We define executions of an ESM or ESM-SK A by first choosing an *interpretation* $\mathcal{I}$. An interpretation $\mathcal{I}$ must satisfy the following: (1) $\mathcal{I}$ maps each function $f \in \mathfrak{F} \setminus \mathfrak{U}$ to its predefined or nominal interpretation, and (2) $\mathcal{I}$ maps each function $f_u \in \mathfrak{U}$ to some valid interpretation. Given the set of state variables V of A, a valuation σ maps each each variable $v \in V$ to a value of the appropriate type, $\sigma(v)$. Let $\mathcal{S}_V$ be the set of all such valuations, given a set of variables V. Given a valuation $\sigma \in \mathcal{S}_V$, a variable $x \notin V$ and a value $v_x \in \mathsf{typeof}(x)$, we write $\sigma[x \mapsto v_x] \in \mathcal{S}_{V \cup \{x\}}$ to denote the valuation that maps all variables $y \neq x$ to $\sigma(y)$ and maps x to v_x.

A *state* of an ESM or ESM-SK A is defined as a pair (l, σ), where $l \in L$ and $\sigma \in \mathcal{S}_V$. Given a transition $r \in R$, of A, of the form $r \triangleq \langle l, m, \mathsf{guard}, \mathsf{updates}, l' \rangle$, and an interpretation $\mathcal{I}$, we say that r is *enabled* with respect to $\mathcal{I}$ at a state (p, σ), if and only if (1) substituting each variable $v \in V$ with $\sigma(v)$ in the expression for guard results in the guard being equivalent to true, and (2) $p = l$. Note that given an interpretation $\mathcal{I}$, the expression guard defines a set of valuations where the transition r is enabled. We write $[\![\mathsf{guard}, \mathcal{I}]\!]$ to denote this set. Similarly, given the interpretation $\mathcal{I}$, updates defines a function:

- $[\![\mathsf{updates}, \mathcal{I}]\!] : \mathcal{S}_V \to \mathcal{S}_{V \cup \{m_p\}}$, if r is an output transition; here $m_p \notin V$ is a variable that represents the payload of the outgoing message m and has type $\mathsf{mtype}(m)$.
- $[\![\mathsf{updates}, \mathcal{I}]\!] : \mathcal{S}_{V \cup \{m_p\}} \to \mathcal{S}_V$, if r is an input transition; here $m_p \notin V$ is a variable that represents the payload of the incoming message m and has type $\mathsf{mtype}(m)$.
- $[\![\mathsf{updates}, \mathcal{I}]\!] : \mathcal{S}_V \to \mathcal{S}_V$, if r is an internal transition.

We define an *execution* of A, under an interpretation $\mathcal{I}$ by describing the sequence of states of A which result from the execution of successive transitions defined on A. We write:

- $(l, \sigma) \xrightarrow{m?v_m} (l', \sigma')$ if and only if A has an *input* transition $r \in R_i$, which has the form $r \triangleq \langle l, m, \mathsf{guard}, \mathsf{updates}, l' \rangle$, $\sigma \in [\![\mathsf{guard}, \mathcal{I}]\!]$, and $[\![\mathsf{updates}, \mathcal{I}]\!] (\sigma[m_p \mapsto v_m]) \equiv \sigma'$.
- $(l, \sigma) \xrightarrow{m!v_m} (l', \sigma')$ if and only if A has an *output* transition $r \in R_o$, which has the form $r \triangleq \langle l, m, \mathsf{guard}, \mathsf{updates}, l' \rangle$, $\sigma \in [\![\mathsf{guard}, \mathcal{I}]\!]$, and $[\![\mathsf{updates}, \mathcal{I}]\!] (\sigma) \equiv \sigma'[m_p \mapsto v_m]$.
- $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$ if and only if A has an *internal* transition $r \in R_\epsilon$, which has the form $r \triangleq \langle l, m, \mathsf{guard}, \mathsf{updates}, l' \rangle$, $\sigma \in [\![\mathsf{guard}, \mathcal{I}]\!]$, and $[\![\mathsf{updates}, \mathcal{I}]\!] (\sigma) \equiv \sigma'$.

For notational convenience we write $(l, \sigma) \to (l', \sigma')$ if (1) there exist m and v_m such that $(l, \sigma') \xrightarrow{m?v_m} (l', \sigma')$, or (2) there exist m and v_m such that $(l, \sigma) \xrightarrow{m!v_m} (l', \sigma')$, or (3) $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$. Further, given a named transition $t \triangleq \langle l, m, \mathsf{guard}, \mathsf{updates}, l' \rangle$, we also write $(l, \sigma) \xrightarrow{t} (l', \sigma')$, to denote $(l, \sigma) \xrightarrow{m?v_m} (l', \sigma')$, or $(l, \sigma) \xrightarrow{m!v_m} (l', \sigma')$, or $(l, \sigma) \xrightarrow{\epsilon} (l', \sigma')$ if $t \in R_i$, $t \in R_o$, or $t \in R_\epsilon$ respectively.

An execution e of an ESM or ESM-SK A under an interpretation $\mathcal{I}$ is thus a sequence of the following form: $e \triangleq (l_0, \sigma_0) \to (l_1, \sigma_1) \to \cdots \to (l_n, \sigma_n) \to \cdots$, where for every $j \geqslant 0$, (l_j, σ_j) is a state of A, (l_0, σ_0) is an initial state of A, and for every $j \geqslant 0$, $(l_j, \sigma_j) \to (l_{j+1}, \sigma_{j+1})$. An execution may be finite or infinite.

A state (l, σ) of an ESM or ESM-SK A is *reachable* under an interpretation $\mathcal{I}$ if and only if A has a finite execution of the form $(l_0, \sigma_0) \to (l_1, \sigma_1) \to \cdots \to (l, \sigma)$, under $\mathcal{I}$. A state (l, σ) of A is called *deadlocked* under an interpretation $\mathcal{I}$ if and only if there does not exist a state (l', σ') such that $(l, \sigma) \to (l', \sigma')$. In other words, no transitions of A are enabled in a deadlocked state. An ESM or ESM-SK A is called *deterministic* under an interpretation $\mathcal{I}$ if for every state $s = (l, \sigma)$ of A, if it is the case that there are multiple transitions enabled in state s, then each of them is an input transition and each of them corresponds to the receipt of a *distinct* message. Lastly, an infinite execution $e \triangleq (l_0, \sigma_0) \to (l_1, \sigma_1) \to \cdots$ of an ESM or ESM-SK A, under an interpretation $\mathcal{I}$ is called a *fair* execution if and only if both of the following hold:

1. For each $F \in \mathcal{F}_w$, if there exists a k such that for all $i \geqslant k$, some transition $t' \in F$ is enabled at state (l_i, σ_i) in e under $\mathcal{I}$ then there exists $j \geqslant k$, such that $(l_k, \sigma_k) \xrightarrow{t} (l_{k+1}, \sigma_{k+1})$ is a step in e, where $t \in F$. Informally, if some transition in $F \in \mathcal{F}_w$ is enabled at every point in an execution e under an interpretation $\mathcal{I}$ after a finite prefix of e, then some transition in F must be taken in the infinite suffix of the execution e.
2. For each $F \in \mathcal{F}_s$, if there exist infinitely many i in such that some transition $t' \in F$ is enabled at state (l_i, σ_i) in e, under $\mathcal{I}$, then there must also exist infinitely many j such that $(l_j, \sigma_j) \xrightarrow{t} (l_{j+1}, \sigma_{j+1})$ is a step in e, where $t \in F$. Informally, if some transition in

$F \in \mathcal{F}_w$ is enabled infinitely often in an execution e, under an interpretation $\mathcal{I}$, then some transition in F must also be executed infinitely often in the execution e.

2.2.6 Composition of ESMs and ESM-SKs

Let $A_1 \triangleq \langle L_1, l_{01}, I_1, O_1, V_1, \sigma_{01}, R_1, \mathcal{F}_{s1}, \mathcal{F}_{w1} \rangle$ be an ESM or ESM-SK. Given another ESM or ESM-SK $A_2 \triangleq \langle L_2, l_{02}, I_2, O_2, V_2, \sigma_{02}, R_2, \mathcal{F}_{s2}, \mathcal{F}_{w2} \rangle$, the composition of A_1 and A_2, denoted by $A_1 \mid A_2$, is defined only if (1) $O_1 \cap O_2 \equiv \emptyset$, and (2) $V_1 \cap V_2 \equiv \emptyset$. We define the composition $A = A_1 \mid A_2$ as an ESM $A \triangleq \langle L, l_0, I, O, V, \sigma_0, R, \mathcal{F}_s, \mathcal{F}_w \rangle$, where:

- $L = L_1 \times L_2$
- $l_0 = (l_{01}, l_{02})$
- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$
- $O = O_1 \cup O_2$
- $V = V_1 \cup V_2$
- σ_0 is a function that maps each variable $v \in V_1 \cup V_2$ to an initial value $\sigma_0(v) \in \mathsf{typeof}(v)$ and is defined as:

$$\sigma_0(v) \triangleq \begin{cases} \sigma_{01}(v) & \text{if } v \in V_1 \\ \sigma_{02}(v) & \text{otherwise} \end{cases}$$

- For every set $F_i \in \mathcal{F}_{s1}$, we include a set F_{i1} in $\mathcal{F}_s$. Similarly, for every set $F_i \in \mathcal{F}_{s2}$, we include a set $F_{i2} \in \mathcal{F}_s$. For every set $F_i \in \mathcal{F}_{w1}$, we include a set $F_{i1} \in \mathcal{F}_w$, and for every set $F_i \in \mathcal{F}_{w2}$, we include a set $F_{i2} \in \mathcal{F}_w$. The construction of these sets is described when we describe how the transitions R are constructed.
- The set of transitions $R = R_i \cup R_o \cup R_\epsilon$, where R is partitioned into R_i, R_o and R_ϵ, is constructed according to the following rules. Note that the following rules also describe how the fairness sets $\mathcal{F}_s$ and $\mathcal{F}_w$ are constructed:
  - For every message $m \in I$, if $m \notin I_2$, then for every transition of $t_1 \in R_1$, which has the form $t_1 \triangleq \langle l_1, m, \mathsf{guard}, \mathsf{updates}, l_1' \rangle$, and for every $l_2 \in L_2$, we include the transitions, each of which has the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}, \mathsf{updates}, (l_1', l_2) \rangle$ in R_i, and thus in R.
  - For every message $m \in I$, if $m \notin I_1$, then for every transition of $t_2 \in R_2$, which has the form $t_2 \triangleq \langle l_2, m, \mathsf{guard}, \mathsf{updates}, l_2' \rangle$, and for every $l_1 \in L_1$, we include the transitions, each of which has the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}, \mathsf{updates}, (l_1, l_2') \rangle$ in R_i, and thus in R.
  - For every message $m \in I_1 \cap I_2$, and for every pair of transitions (t_1, t_2), such that

$t_1 \in R_1, t_2 \in R_2$, where t_1 has the form $t_1 \triangleq \langle l_1, m, \mathsf{guard}_1, \mathsf{updates}_1, l_1' \rangle$, and t_2 is of the form $t_2 \triangleq \langle l_2, m, \mathsf{guard}_2, \mathsf{updates}_2, l_2' \rangle$, we include a transition t which has the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}_1 \wedge \mathsf{guard}_2, \mathsf{updates}_1; \mathsf{updates}_2, (l_1', l_2') \rangle$ in R_i and thus in R. Note that the operator ";" denotes sequencing of updates.

- For every message $m \in O_1$, if $m \notin I_2$, then for every transition $t_1 \in R_1$ which has the form $t_1 \triangleq \langle l_1, m, \mathsf{guard}, \mathsf{updates}, l_1' \rangle$, and for every $l_2 \in L_2$, we include the transitions, each of the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}, \mathsf{updates}, (l_1', l_2) \rangle$ in R_o and thus in R. Further, for each $F_i \in \mathcal{F}_{s1}$, ($\mathcal{F}_{w1}$) such that $t_1 \in F_i$, for every $l_2 \in L_2$, we include the transition $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}, \mathsf{updates}, (l_1', l_2) \rangle$ in the set $F_{i1} \in \mathcal{F}_s$ (respectively $F_{i1} \in \mathcal{F}_w$).
- Similarly, for every message $m \in O_2$, if $m \notin I_1$, then for every transition $t_2 \in R_2$ which has the form $t_2 \triangleq \langle l_2, m, \mathsf{guard}, \mathsf{updates}, l_2' \rangle$, and for every $l_1 \in L_1$, we include the transitions, each of the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}, \mathsf{updates}, (l_1, l_2') \rangle$ in R_o and thus in R. Further, for each $F_i \in \mathcal{F}_{s2}$, ($\mathcal{F}_{w2}$) such that $t_2 \in F_i$, for every $l_1 \in L_1$, we include the transition $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}, \mathsf{updates}, (l_1, l_2') \rangle$ in the set $F_{i2} \in \mathcal{F}_s$ (respectively $F_{i2} \in \mathcal{F}_w$).
- For every message $m \in O_1$, if $m \in I_2$, then for every pair of transitions (t_1, t_2) such that $t_1 \in R_1$ and $t_2 \in R_2$, where t_1 and t_2 have the form $t_1 \triangleq \langle l_1, m, \mathsf{guard}_1, \mathsf{updates}_1, l_1' \rangle$, $t_2 \triangleq \langle l_2, m, \mathsf{guard}_2, \mathsf{updates}_2, l_2' \rangle$, we include a transition t which has the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}_1 \wedge \mathsf{guard}_2, \mathsf{updates}_1; \mathsf{updates}_2, (l_1', l_2') \rangle$ in R_o, and thus in R. Further, for each $F_i \in \mathcal{F}_{s1}$ ($\mathcal{F}_{w1}$) such that $t_1 \in F_i$, we include the transition t in $F_{i1} \in \mathcal{F}_s$ (respectively $F_{i1} \in \mathcal{F}_w$).
- Similarly, for every message $m \in O_2$, if $m \in I_1$, then for every pair of transitions $(t_1, t_2) \in R_1 \times R_2$, where t_1 and t_2 have the form $t_1 \triangleq \langle l_1, m, \mathsf{guard}_1, \mathsf{updates}_1, l_1' \rangle$, $t_2 \triangleq \langle l_2, m, \mathsf{guard}_2, \mathsf{updates}_2, l_2' \rangle$, we include a transition t which has the form $t \triangleq \langle (l_1, l_2), m, \mathsf{guard}_1 \wedge \mathsf{guard}_2, \mathsf{updates}_2; \mathsf{updates}_1, (l_1', l_2') \rangle$ in R_o, and thus in R. Further, for each $F_i \in \mathcal{F}_{s2}$ ($\mathcal{F}_{w2}$) such that $t_1 \in F_i$, we include the transition t in $F_{i2} \in \mathcal{F}_s$ (respectively $F_{i2} \in \mathcal{F}_w$).
- For every transition $t_1 \in R_1$ which is of the form $t_1 \triangleq \langle l_1, \epsilon, \mathsf{guard}, \mathsf{updates}, l_1' \rangle$, and for every $l_2 \in L_2$, we include in R_ϵ and thus in R, every transition t which is of the form $t \triangleq \langle (l_1, l_2), \epsilon, \mathsf{guard}, \mathsf{updates}, (l_1', l_2) \rangle$. Further, for each $F_i \in \mathcal{F}_{s1}$ ($\mathcal{F}_{w1}$) such that $t_1 \in F_i$, for every $l_2 \in L_2$, we include the transition $t \triangleq \langle (l_1, l_2), \epsilon, \mathsf{guard}, \mathsf{updates}, (l_1', l_2) \rangle$ in $F_{i1} \in \mathcal{F}_s$ (respectively $F_{i1} \in \mathcal{F}_w$).
- Similarly, for every transition $t_2 \in R_2$ of the form $t_2 \triangleq \langle l_2, \epsilon, \mathsf{guard}, \mathsf{updates}, l_2' \rangle$, and

for every $l_1 \in L_1$, we include every transition t, each of which has the form $t \triangleq \langle (l_1, l_2), \epsilon, \mathsf{guard}, \mathsf{updates}, (l_1, l_2') \rangle$ in R_ϵ and thus in R. Further, for each $F_i \in \mathcal{F}_{s2}$ ($\mathcal{F}_{w2}$) such that $t_2 \in F_i$, for every $l_1 \in L_1$, we include the transition t of the form $t \triangleq \langle (l_1, l_2), \epsilon, \mathsf{guard}, \mathsf{updates}, (l_1, l_2') \rangle$ in $F_{i2} \in \mathcal{F}_s$ (respectively $F_{i2} \in \mathcal{F}_w$).

Because the composition of two ESMs or ESM-SKs is again an ESM or ESM-SK, all the earlier definitions regarding reachability, deadlocks, executions and fairness are still valid. Note that the composition operator "|" is associative and commutative.

2.2.7 Symmetry and Symmetric Types

Distributed protocols often exhibit symmetric behavior, e.g., the behavior of the state machines in Peterson's mutual exclusion algorithm described in Section 1.2.3 exhibits symmetry. To allow the programmer to express such symmetric behavior, we use symmetric types, which are similar to the *scalarset* construct used in the Murφ model checker [ID96].

A symmetric type $T \in \mathcal{T}$ is characterized by (1) its name, and (2) its cardinality, $|T|$, which is a finite natural number. The only operations permitted on values of a symmetric type are comparisons for equality and disequality between two values. Given a collection of state machines parameterized by a set of symmetric types, e.g., the state machines P_0 and P_1 in Peterson's algorithm, the behavior of the system is required to be invariant under permutations (*i.e.*, renaming) of the parameter values.

Given a symmetric type T, let $\mathsf{perm}(T)$ be the set of all permutations $\pi_T : T \to T$, over the symmetric type T. For ease of notation, we define $\pi_T(v) = v$ for values $v \notin T$, *i.e.*, values whose type is not T, provided that the type of v is not an array or record type. If the type of v is a record type, then $\pi_T(v)$ is defined as the record value obtained by applying π_T on each field of v. If the type of v is an array type, whose index type is *not* T, then $\pi_T(v)$ is defined as the array value obtained by applying π_T recursively to all the elements of v. If the type of v is an array type whose index type is T, then $\pi_T(v)$ is defined as the value obtained by first recursively applying π_T to all the elements of v and then permuting the array elements themselves according to π_T, *i.e.*, for all $j \in T$, $\pi_T(v)[\pi_T(j)] \equiv \pi_T(v[j])$. Given the collection of symmetric types in the system, $T_1, T_2, \ldots, T_n \in \mathcal{B}$, we define the set of *system wide* permutations, $\mathsf{perm}(\mathcal{T}_\mathcal{B})$, as the composition of the permutations over the individual types, $\pi_{T_1} \circ \pi_{T_2} \circ \cdots \circ \pi_{T_n}$.

ESMs, ESM-SKs and messages may be parameterized by a list of parameter variables, each of a symmetric type. The semantics of such parameterization is that there exists one

instance of the object for every possible value that the parameter variables can take. Consider a parameterized message $m\langle p_1 : T_1, p_2 : T_2, \ldots, p_n : T_n\rangle$. Here $p_1, p_2, \ldots, p_n$ are parameter variables which can take values of types $T_1, T_2, \ldots, T_n$ respectively. Then for every possible list of values $\langle v_1, v_2, \ldots, v_n\rangle$, where $v_i \in T_i$, and $i \in [1, n]$, there exists an instance $m\langle p_1 \mapsto v_1, p_2 \mapsto v_2, \ldots, p_n \mapsto v_n\rangle$ of the parameterized message m. The semantics of a parameterized ESM or ESM-SK are similar, except that the parameter variables p_i are available for use as read-only variables within the guards and updates of the ESM or ESM-SK.

Given the set of types $\mathcal{T}_\mathcal{B}$ an interpretation $\mathcal{I}$ is said to be symmetric with respect to $\mathcal{T}_\mathcal{B}$ if and only if for all $f_u : d_1, d_2, \ldots, d_n \to r \in \mathfrak{U}$, for all $\pi \in \mathsf{perm}(\mathcal{T}_\mathcal{B})$, and for all $e_1 \in d_1, e_2 \in d_2, \ldots, e_n \in d_n$, we have that $\pi(f_u(\pi(e_1), \pi(e_2), \ldots, \pi(e_n))) \equiv f_u(e_1, e_2, \ldots, e_n)$. An ESM or ESM-SK A is said to be symmetric with respect to $\mathcal{T}_\mathcal{B}$, if and only if for any interpretation $\mathcal{I}$ such that $\mathcal{I}$ is symmetric with respect to $\mathcal{T}_\mathcal{B}$, and for all $\pi \in \mathsf{perm}(\mathcal{T}_\mathcal{B})$, every execution of A under $\mathcal{I}$ of the form:

$$e \triangleq (l_0, \sigma_0) \xrightarrow{*_1} (l_1, \sigma_1) \xrightarrow{*_2} \cdots \xrightarrow{*_n} (l_n, \sigma_n) \xrightarrow{*_{n+1}} \cdots$$

where $*_i$ is one of $m_i?v_{m_i}$ or $m_i!v_{m_i}$ or ϵ, implies that the *permuted* execution of the form:

$$\pi(e) \triangleq (\pi(l_1), \pi(\sigma_1)) \xrightarrow{\pi(*_1)} (\pi(l_2), \pi(\sigma_2)) \xrightarrow{\pi(*_2)} \cdots \xrightarrow{\pi(*_n)} (\pi(l_n), \pi(\sigma_n)) \xrightarrow{\pi(*_{n+1})} \cdots$$

is also admitted by A under the same interpretation $\mathcal{I}$. Here $\xrightarrow{\pi(*_i)}$ represents a transition along which the instances of messages parameterized by symmetric types and message payloads are also permuted according to the permutation π. Further, we also require that e is a weakly (respectively, strongly) fair execution of A if and only if $\pi(e)$ is a weakly (respectively, strongly) fair execution of A. In other words, we require the strong and weak fairness assumptions on A to be symmetric as well.

Our framework allows the programmer to describe protocols which are symmetric according to the notion of symmetry just described. We ensure that symmetry breaking constructs are not used by enforcing syntactic restrictions on the description of ESMs and ESM-SKs. This is done in a manner similar to the what has been described in earlier work [ID96]. Further, we also ensure that any interpretations $\mathcal{I}$ that are generated during the process of synthesis are such that they satisfy the symmetry assumptions made on the ESM-SKs that they are a part of, as we shall describe in later sections.

2.2.8 Requirements and Specifications

We now turn our attention to the way in which requirements — *i.e.*, the properties that we expect from a correct protocol — are specified. The techniques proposed in this dissertation support requirements expressed either as Linear Temporal Logic (LTL) formulas, or directly as Büchi monitors.[3] To make the presentation self-contained, we now briefly describe the syntax and semantics of LTL and describe how we use monitors (possibly constructed from the LTL formulas) to characterize the correctness of protocols.

Linear Temporal Logic

Given a set of atomic propositions AP, the syntax of Linear Temporal Logic (LTL) formulas over these atomic propositions is given by the following rules:

- If $p \in \mathsf{AP}$, then p is an LTL formula
- If φ_1 and φ_2 are LTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\mathbf{X}\,\varphi_1$, and $\varphi_1\,\mathbf{U}\,\varphi_2$.

Other commonly used operators and connectives can be defined in terms of these basic operators using the standard equivalences. We list a few of them here:

- $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- $\mathbf{F}\,\varphi_1 \equiv \mathtt{true}\,\mathbf{U}\,\varphi_1$
- $\mathbf{G}\,\varphi_1 \equiv \neg\mathbf{F}\neg\varphi_1$
- $\varphi_1\,\mathbf{R}\,\varphi_2 \equiv \neg\,(\neg\varphi_1\,\mathbf{U}\,\neg\varphi_2)$
- $\varphi_1\,\mathbf{W}\,\varphi_2 \equiv (\varphi_1\,\mathbf{U}\,\varphi_2) \vee \mathbf{G}\,\varphi_1$

We define the semantics of an LTL formula over *executions* of ESMs and ESM-SKs. Given the set of types $\mathcal{T}$, a set of function symbols $\mathfrak{F}$, an ESM or ESM-SK $A = \langle L, l_0, I, O, V, \sigma_0, R, \mathcal{F}_s, \mathcal{F}_w\rangle$, we let the set of atomic propositions AP be the set of all Boolean valued expressions over $V \cup \{\mathsf{loc}\}$ which do not involve Boolean connectives. Here, $\mathsf{loc} \notin V$ is a distinguished variable that tracks the *location* of A, whose values are allowed to range over L, and the only operation allowed on this type is comparison of values of equality. Given a state $s \triangleq (l, \sigma)$, where $l \in L$ and $\sigma \in \mathcal{S}_V$, an interpretation $\mathcal{I}$, and an atomic predicate $p \in \mathsf{AP}$, we say that s *satisfies* p under the interpretation $\mathcal{I}$, written as $s \models_{\mathcal{I}} p$ if and only if p', which is obtained by substituting l for every occurrence of loc in p and $\sigma(v)$ for every occurrence of v in p, for each variable $v \in V$ is equivalent to $\mathtt{true}$. We extend the notion of satisfiability to arbitrary Boolean expressions

[3] Every LTL formula can be translated into a (possibly non-deterministic) Büchi monitor, but there exist Büchi monitors/automata which do not have an equivalent LTL formula.

— which are just atomic propositions composed with Boolean connectives — in the natural manner.

Given an infinite execution $e \triangleq s_0 \to s_1 \to \cdots$ of A, under an interpretation $\mathcal{I}$, where $s_i \triangleq (l_i, \sigma_i)$ for $i \geqslant 0$, the satisfaction semantics of an LTL formula φ over e are inductively defined as follows, where φ_1 and φ_2 are subformulas of φ, and p is an atomic proposition; *i.e.*, $p \in \mathsf{AP}$. Note that we use the notation $e \models_{\mathcal{I}} \varphi$ to denote that the execution e (also under the interpretation $\mathcal{I}$) satisfies the LTL formula φ under the interpretation $\mathcal{I}$.

- If $\varphi \equiv p$, then, $e \models_{\mathcal{I}} p$ if and only if $s_0 \models_{\mathcal{I}} p$.
- If $\varphi \equiv \varphi_1 \wedge \varphi_2$, then $e \models_{\mathcal{I}} \varphi$ if and only if $e \models_{\mathcal{I}} \varphi_1$ and $e \models_{\mathcal{I}} \varphi_2$.
- If $\varphi \equiv \neg\varphi_1$, then $e \models_{\mathcal{I}} \varphi$ if and only if it is not the case that $e \models_{\mathcal{I}} \varphi_1$.
- If $\varphi \equiv \mathbf{X}\,\varphi_1$ then $e \models_{\mathcal{I}} \varphi$ if and only if $s_1 \models_{\mathcal{I}} \varphi_1$.
- If $\varphi \equiv \varphi_1\,\mathbf{U}\,\varphi_2$, then $e \models_{\mathcal{I}} \varphi$ if and only if $\exists j \geqslant 0\,(s_j \models_{\mathcal{I}} \varphi_2 \wedge \forall i < j\,(s_i \models_{\mathcal{I}} \varphi_1))$.

The semantics for the other operators such as **F**, **G**, **R** and **W**, can be deduced by using the equivalences mentioned earlier to express LTL formulas involving these operators in terms of the basic operators **X** and **U**.

We can now define the satisfaction semantics of an ESM or ESM-SK A with respect to an LTL formula φ as follows: A satisfies φ, under an interpretation $\mathcal{I}$, written $A \models_{\mathcal{I}} \varphi$ if and only if for *every* execution e of A (also under the interpretation $\mathcal{I}$), we have that $e \models_{\mathcal{I}} \varphi$.

Algorithmically, this check is usually performed by translating the negation of the LTL formula, *i.e.*, $\neg\varphi$, into a Büchi monitor with accepting states and checking if the synchronous product of the Büchi monitor and A admits a fair accepting cycle. We defer a detailed description of this model-checking algorithm until Section 8.3, but we now provide a brief description of the (Büchi and safety) monitors used in this process.

Monitors

Every LTL formula φ can be translated into a (possibly non-deterministic) Büchi automaton (BA) or Büchi monitor which can then be used to algorithmically check if a given transition system satisfies the LTL formula φ. The translation from LTL to BA takes time exponential in the size of the LTL formula, and has been studied extensively in literature [WVS83, LP85, VW94, DGV99, EH00, SB00, GO01, BKRS12, Dur14], and will not be covered in detail in this dissertation. We will assume that the requirements which are specified using LTL formulas have already been translated into Büchi monitors, whose form we describe in this section. This translation

can be accomplished using widely available tools like LTL2BA [GO01], LTL3BA [BKRS12], or SPOT [Dur14] for instance.

Consider an ESM or ESM-SK $A \triangleq \langle L, l_0, I, O, V, \sigma_0, R, \mathcal{F}_s, \mathcal{F}_w \rangle$. Recall that $\mathcal{S}_V$ is the set of all valuations of the set of variables V. We denote the set of all *states* of A as $\mathfrak{S} \triangleq L \times \mathcal{S}_V$. A *monitor* over A is an automaton $M \triangleq \langle Q, q_0, \Delta \rangle$. Here Q is the set of automaton states, $q_0 \in Q$ is the initial state and $\Delta \subseteq Q \times \mathfrak{S} \times Q$ is a transition relation. We denote the *synchronous* composition of A with M as $A \parallel M$. The semantics of such a synchronous composition are standard: Each time A makes a transition, M makes a transition as well. Suppose the current state of M is q, and the state of A is $s \in \mathfrak{S}$, then M can (non-deterministically) transition to any state q' such that $(q, s, q') \in \Delta$. The notion of an execution is extended in the natural manner to the product $A \parallel M$, by augmenting the state with a component denoting the state $q \in Q$ of the monitor M, and we write $(l, \sigma, q) \to (l', \sigma', q')$, where the locations $l, l' \in L$, the valuations $\sigma, \sigma' \in \mathcal{S}_V$ and $q, q' \in Q$ if and only if $(l, \sigma) \to (l', \sigma')$, and $(q, (l, \sigma), q') \in \Delta$. The notion of reachability also follows naturally from the extended notion of an execution. The composition $A \parallel M$ *inherits* the the fairness assumptions from A, and an execution e of $A \parallel M$ under an interpretation $\mathcal{I}$ is fair if and only if the projection of e onto A is fair, under the interpretation $\mathcal{I}$.[4]

A *safety* monitor is a monitor augmented with a set of *error* states. In other words, a safety monitor $M_s \triangleq \langle Q, q_0, \Delta, Q_{err} \rangle$, where $Q_{err} \subseteq Q$ is a set of error states. An finite execution e of $A \parallel M_s$ is called *erroneous* if the monitor M_s is in a state $q \in Q_{err}$ in the last state of the execution e. Given an interpretation $\mathcal{I}$, A satisfies M_s under the interpretation $\mathcal{I}$, written as $A \models_{\mathcal{I}} M_s$, if and only if $A \parallel M_s$ admits no erroneous executions under the interpretation $\mathcal{I}$, *i.e.*, a state where the monitor component of the state $q \in Q_{err}$ is not reachable.

A *liveness* monitor is a monitor augmented with a set of *accepting* states. In other words, a liveness monitor $M_l \triangleq \langle Q, q_0, \Delta, Q_{acc} \rangle$, where $Q_{acc} \subseteq Q$ is a set of accepting states. An infinite execution e of $A \parallel M_l$ is called an *accepting* execution if the monitor M_l visits an accepting state infinitely many times in e. Given an interpretation $\mathcal{I}$, we say that A satisfies M_l under the interpretation $\mathcal{I}$, written as $A \models_{\mathcal{I}} M_l$ if and only if every fair execution of $A \parallel M_l$, under the interpretation $\mathcal{I}$, is not an accepting execution.

Although we have made a distinction between liveness and safety monitors for ease of exposition, especially in the later chapters of this dissertation, we note that both safety and liveness

[4] An execution e of $A \parallel M$ is projected onto A by simply dropping the component for M in each state and each transition of e.

monitors can be expressed as (possibly non-deterministic) Büchi monitors with accepting states. This is immediately obvious in the case of liveness monitors of the form we have described above: a liveness monitor is itself a Büchi monitor with Q_{acc} as the set of accepting states of the Büchi monitor. One can express a safety monitor as a Büchi monitor by setting $Q_{acc} = Q_{err}$ and adding a self-loop from every state $q \in Q_{err}$, *i.e.*, by adding $(q, (l, \sigma), q)$ for every $q \in Q_{err}$ and for every $(l, \sigma) \in \mathfrak{S}$ to Δ, thereby allowing it to accept infinite executions. Further, we also allow monitors to be *symmetric* in the same manner as for ESMs and ESM-SKs. Finally, we say that an ESM or ESM-SK A satisfies the LTL specification φ under an interpretation $\mathcal{I}$, if and only if $A \models_{\mathcal{I}} M$, where M is the Büchi monitor corresponding to the LTL formula $\neg\varphi$.

2.3 Problem Statement

We are given $A_1, A_2, \ldots, A_n$, where each A_i is either an ESM or an ESM-SK, over a set of types $\mathcal{T}_{\mathcal{B}}$ and a function vocabulary $\mathfrak{F}$. We are also given a set of safety monitors $M_{s1}, M_{s2}, \ldots, M_{sm}$, and a set of liveness monitors $M_{l1}, M_{l2}, \ldots, M_{lk}$. The objective is to find an interpretation $\mathcal{I}$ such that (1) the product $A \triangleq A_1 \mid A_2 \mid \cdots \mid A_n$ is deadlock free under the interpretation $\mathcal{I}$, (2) each ESM-SK $A \in \{A_i\}, i \in [1, n]$ is *deterministic* under the interpretation $\mathcal{I}$, (3) for each $M_{si}, i \in [1, m]$, $A \models_{\mathcal{I}} M_{si}$, (4) for each $M_{lj}, j \in [1, k]$, $A \models_{\mathcal{I}} M_{lj}$, and (5) $\mathcal{I}$ is symmetric with respect to $\mathcal{T}_{\mathcal{B}}$.

Note that we require that the interpretation $\mathcal{I}$ is such that each ESM-SK under $\mathcal{I}$ is deterministic. This is based on our observation that typically, the ESMs which cooperate to achieve the goals of a distributed protocol are deterministic individually. The non-determinism in such protocols arises out of (1) non-determinism in the scheduler, and (2) non-determinism in the environment ESMs. We assume that environment ESMs are completely specified, and we will therefore never be required to complete environment ESM-SKs.

The next chapter presents an elegant symbolic algorithm to obtain *all* interpretations $\mathcal{I}$ which satisfy the requirements set forth in the problem statement. Unfortunately, this algorithm is not effective in practice. We describe the reasons why this is so, and also elaborate on the insights obtained by implementing the algorithm and experimenting with a simple cache coherence protocol. These insights also explain some of the choices made in the rest of the work that this dissertation describes.

3

A Symbolic Strategy via Parametrized Transitions

Given that the problem defined in Chapter 2 only involves finite types, thereby rendering the space of solutions finite, one can immediately imagine an elegant symbolic solution to find *all* interpretations which satisfy the requirements set forth in Section 2.3. Such a solution would have the following high level outline:

1. Translate the descriptions of the ESMs and ESM-SKs into a parameterized symbolic transition system, which could be represented using Reduced Ordered Binary Decision Diagrams (henceforth referred to as ROBDDs or BDDs) [Bry85, Bry86, BRB90]. The values of the parameters determine the interpretation $\mathcal{I}$ which is chosen. The set of all interpretations is finite, given that the domains and ranges of all the function symbols $f \in \mathfrak{F}$ are finite, and thus the introduced parameters can only take on a finite number of values, since every value corresponds to a distinct interpretation. The values for these parameters are encoded symbolically as well, and are initially left unconstrained, *i.e.*, *any* interpretation is allowed.
2. Translate the requirements expressed in LTL into a *tester* transition system as described in the work by Kesten et. al. [KPRS06]
3. Interleave the symbolic model-checking algorithm described in the work by Kesten et. al. [KPRS06] with steps to (symbolically) prune parameter valuations which result in incorrect interpretations being chosen, until the model-checking succeeds. At this point, the parameter valuations that we are left with correspond to *all* interpretations that satisfy the requirements set forth in Section 2.3.

This strategy only handles requirements expressible in LTL, whereas the problem statement outlined in Section 2.3 handles requirements expressed as arbitrary Büchi monitors. Given that there exist Büchi monitors which do not have an equivalent LTL formula, this strategy solves

a restricted version of the problem defined in Section 2.3. The strategy can be extended to handle arbitrary Büchi monitors in a relatively straightforward manner. However, the objective of presenting this solution strategy here is to highlight the complexity of distributed protocol completion and glean insights that lead to developing more effective algorithmic strategies. So, we will focus only on LTL requirements and a slightly simplified version of the original problem in this chapter enabling us to leverage the proofs of correctness from earlier work [KPRS06].

3.1 A Simplified, Finite Version of the Problem

Consider the following simplified version of the problem defined in Section 2.3: Each ESM or ESM-SK has no state variables, *i.e.*, $V = \emptyset$. Further, we assume that messages do not have a payload, *i.e.*, $\mathsf{mtype}(m) = \mathsf{unit}$ for all messages $m \in \Sigma$. Essentially, the state machines are now simply finite-state machines or finite-state machine sketches, and we will refer to them as FSMs or FSM-SKs respectively. Each transition $t \in R$ of such an FSM or FSM-SK will have the form $t \triangleq \langle l, m, \mathsf{guard}, l' \rangle$. With no state variables and message payloads, the updates component of a transition is no longer relevant. The guard in this setting is always the Boolean constant `true` in the case of an FSM, but is allowed to be a propositional variable in the case of an FSM-SK, where setting the propositional variable to `true` indicates that $t \in R$ and setting to `false` indicates that $t \notin R$. An *interpretation* $\mathcal{I}$ in this simplified setting is then simply a valuation for these unknown guards. Note that even in this simplified setting, the rest of the definitions regarding composition, symmetry, executions and fairness remain unchanged. Thus an FSM or FSM-SK A is the tuple $A \triangleq \langle L_A, l_{A0}, I_A, O_A, F_{As}, F_{Aw} \rangle$. We assume that the specification is provided as a single LTL formula φ,[5] over the single distinguished variable loc, which represents the location of the FSM-SK A.

We now briefly outline each of these steps of the symbolic solution strategy for this simplified version of the problem. We then explain why this strategy is not satisfactory, even for this simplified version of the problem, based on empirical observations.

3.2 The Parameterized Symbolic Transition System

Given an FSM or FSM-SK A, which has the form $A \triangleq \langle L, l_0, I, O, R, F_{As}, F_{Aw} \rangle$, which itself could possibly be the composition of two or more FSMs or FSM-SKs, we now outline how

[5] If A is required to satisfy multiple LTL formulas, then these can be expressed as a single LTL formula which is the conjunction of the given formulas.

to represent A as a symbolic transition system. We denote the symbolic transition system corresponding to A as $\widetilde{A} \triangleq \left\langle \widetilde{V}, \widetilde{l}_0, \widetilde{R}, \widetilde{\mathcal{F}}_{As}, \widetilde{\mathcal{F}}_{Aw} \right\rangle$, where:

- $\widetilde{V}$ is the set of variables in the symbolic transition system and we use $\widetilde{V}'$ to represent the *primed* version of the variables in $\widetilde{V}$. The primed version of a variable denotes the value of the variable in the *next* state. We require them to distinguish between the current and next values of variables in a symbolic representation.
- $\widetilde{l}_0$ is the symbolic representation of the set of initial states of the symbolic transition system.
- $\widetilde{R}$ is the symbolic transition relation that relates the values in the *next* state, represented by the primed version of the variables, *i.e.*, $\widetilde{V}'$, to their values in the *current* state, represented by the unprimed version of the variables, *i.e.*, $\widetilde{V}$.
- $\widetilde{\mathcal{F}}_{As}$ is a set of pairs of predicates over $\widetilde{V}$, with one pair encoding each set $F \in \mathcal{F}_{As}$.
- $\widetilde{\mathcal{F}}_{Aw}$ is a set of predicates over $\widetilde{V}$, with one predicate encoding each set $F \in \mathcal{F}_{As}$.

We will use BDDs to represent $\widetilde{l}_0$, $\widetilde{R}$, $\widetilde{\mathcal{F}}_{As}$ and $\widetilde{\mathcal{F}}_{Aw}$ in our presentation. It will be convenient to partition R into the sets R_{fixed} and R_{synth}, where R_{fixed} consists of the transitions with "fixed" interpretations, *i.e.*, where guard is the Boolean constant `true`, and R_{synth} consists of the set of transitions whose interpretations are to be synthesized, *i.e.*, where guard is a Boolean valued variable whose value needs to be determined. The usual determinism constraints outlined in Chapter 2 are implicitly assumed in the rest of this chapter. We note that these can also be specified using suitable constraints on R_{synth} and fit well into the BDD based algorithms described in this chapter.

We define the set of variables of $\widetilde{A}$ as $\widetilde{V} \triangleq \{\mathsf{loc}, \mathsf{lastt}\} \cup \widetilde{G}$, where loc and lastt, represent the location of the FSM A, and the identity of the transition that was taken to arrive at the current state, respectively. The set of variables $\widetilde{G} = \{g_1, g_2, \ldots, g_k\}$, where $k = |R|$, consists of Boolean valued variables, where g_i represents the guard of the transition t_i, for each $t_i \in R$. The lastt variables are necessary because symbolic model checking algorithms are more naturally suited to handle handle *state-based* fairness requirements, rather than *transition-based* fairness requirements. By adding the variable lastt, we are essentially enabling the translation of transition-based fairness into state-based fairness.

We now describe how each of the symbolic representations for $\widetilde{l}_0$, $\widetilde{R}$, $\widetilde{\mathcal{F}}_{Aw}$, and $\widetilde{\mathcal{F}}_{As}$ are constructed, starting from a definition of A. We present the predicates over the set of variables V as well as the primed variables V' that correspond to the definition of each of these. It is straightforward to use BDD operations, using a BDD library like CUDD [SB00], for example, to

construct BDDs corresponding to these predicates. The set of initial states of $\widetilde{A}$, can be encoded as follows:

$$\widetilde{l}_0 \equiv \left((\mathsf{loc} = l_0) \wedge \bigwedge_{t_i \in R_{\text{fixed}}} g_i \right)$$

Each transition $t_i \in R$ of the form $t_i \triangleq \langle l_i, m_i, \mathsf{guard}_i, l_i' \rangle$ is encoded symbolically as:

$$(\mathsf{loc} = l_i) \wedge g_i \wedge \left(\mathsf{loc}' = l_i'\right) \wedge \left(\mathsf{lastt}' = t_i\right) \wedge g_i' = g_i$$

The constraint $g_i' = g_i$ ensures that the *interpretation* remains constant across transitions. The complete symbolic transition relation $\widetilde{R}$ is then simply the disjunction of the above encoding over all the transitions $t \in R$:

$$\widetilde{R} \equiv \bigvee_{t_i \in R} \left((\mathsf{loc} = l_i) \wedge g_i \wedge \left(\mathsf{loc}' = l_i'\right) \wedge \left(\mathsf{lastt}' = t_i\right) \right)$$

Consider a fairness assumption $F_w \in \mathcal{F}_w$, where $F_w = \{t_1, t_2, \ldots, t_n\}$, we symbolically encode F_w, denoted $\widetilde{F}_w$, using a single predicate of the form:

$$\widetilde{F}_w \equiv \neg \mathsf{enabled}(F_w) \vee \mathsf{taken}(F_w)$$

where $\mathsf{enabled}(F_w) \equiv \bigvee_{t_i \in F_w} (\mathsf{loc} = l_i \wedge g_i)$, with l_i and g_i referring to the initial location and the Boolean variable corresponding to the guard of the transition t_i, and the predicate $\mathsf{taken}(F_w) \equiv \bigvee_{t_i \in F_w} \mathsf{lastt} = t_i$. The predicate $\mathsf{enabled}(F_w)$ encodes whether *any* transition in F_w *can* be executed, and $\mathsf{taken}(F_w)$ encodes whether the current state has been reached by executing any transition in the set F_w.

For strong fairness assumptions $F_s \in \mathcal{F}_s$, where $F_s = \{t_1, t_2, \ldots, t_n\}$, we symbolically encode F_s, denoted $\widetilde{F}_s$, using a pair of Boolean valued formulas p and q, where $p \triangleq \mathsf{enabled}(F_s)$ and $q \triangleq \mathsf{taken}(F_s)$, where enabled and taken are as defined above. The predicate encodings of fairness assumptions will be used to characterize if a non-empty terminal strongly connected component is fair. We refer the reader to earlier work [KPRS06] for a more detailed explanation.

3.3 Construction of the LTL Tester

We now describe how to construct a *tester* — which is itself a transition system augmented with a set of weak fairness assumptions — corresponding to an LTL formula φ. We denote the

tester for the LTL formula φ as T_φ. This description has been adapted from earlier work by Kesten et. al. [KPRS06].

The symbolic transition relation of T_φ refers to variables in $\widetilde{V}$ and $\widetilde{V}'$, but does not constrain either. T_φ also refers to variables in the set $X_\varphi \cup X'_\varphi$, where X_φ is a set of Boolean valued variables, and is defined as $X_\varphi \triangleq \{x_p \mid p \text{ is a } \textit{principally temporal} \text{ sub-formula of } \varphi\}$. The primed version of the variables in X_φ is denoted by the set X'_φ. A (sub-)formula is *principally temporal* if its top level operator is one of **U** or **X**. If ψ is a sub-formula of φ (note that we consider φ to be a sub-formula of itself), then we denote it as $\psi \in \varphi$. To define the transition relation of T_φ, we define a mapping χ which maps each sub-formula ψ of φ to an assertion over $\widetilde{V}$, defined as follows:

$$\chi(\psi) = \begin{cases} \psi & \text{if } \psi \text{ is an atomic proposition} \\ \neg\chi(p) & \text{if } \psi \equiv \neg p \\ \chi(p_1) \vee \chi(p_2) & \text{if } \psi \equiv p_1 \vee p_2 \\ x_\psi & \text{if } \psi \text{ is principally temporal} \end{cases} \tag{3.1}$$

The transition relation for T_φ is as shown below:

$$\widetilde{R}_\varphi \equiv \bigwedge_{\mathbf{X}\, p \in \varphi} \left(x_{\mathbf{X}\, p} \leftrightarrow \chi'(p)\right) \wedge \bigwedge_{p\, \mathbf{U}\, q \in \varphi} \left(x_{p\, \mathbf{U}\, q} \leftrightarrow \left(\chi(q) \vee \left(\chi(p) \wedge x'_{p\, \mathbf{U}\, q}\right)\right)\right) \tag{3.2}$$

where $\chi'(p)$ refers to the assertion corresponding to the sub-formula p, but where all the variables are substituted with their primed variants. Finally, for each $p\, \mathbf{U}\, q \in \varphi$, we add the following (symbolic) weak fairness assumption:

$$\chi(q) \vee \neg x_{p\, \mathbf{U}\, q} \tag{3.3}$$

to T_φ and set the initial condition for T_φ to be `true`.

To summarize, the symbolic transition system for the tester for the LTL formula φ is denoted as $T_\varphi \triangleq \left\langle \widetilde{V}_\varphi, \widetilde{l}_{\varphi 0}, \widetilde{R}_\varphi, \widetilde{\mathcal{F}}_{\varphi s}, \widetilde{\mathcal{F}}_{\varphi w} \right\rangle$, where:

- $\widetilde{V}_\varphi = \widetilde{V} \cup X_\varphi$.
- $\widetilde{l}_{\varphi 0} = \texttt{true}$
- $\widetilde{R}_\varphi$ is constructed as shown in (3.2).
- $\widetilde{\mathcal{F}}_{\varphi s} = \emptyset$, and
- The set $\widetilde{\mathcal{F}}_{\varphi w}$ consists of the predicates shown in (3.3), one for each sub-formula $\psi \in \varphi$,

such that the sub-formula ψ is an *until* sub-formula, *i.e.*, it has the form $p\,\mathbf{U}\,q$, where the sub-formulas p and q are the ones referred to in the Formula (3.3).

3.4 The Symbolic Synthesis Algorithm

Given two symbolic transition systems $\widetilde{A}_0$ and $\widetilde{A}_1$, where $\widetilde{A}_0 \triangleq \left\langle \widetilde{V}_0, \widetilde{l}_{00}, \widetilde{R}_0, \widetilde{\mathcal{F}}_{0s}, \widetilde{\mathcal{F}}_{0w} \right\rangle$ and $\widetilde{A}_1 \triangleq \left\langle \widetilde{V}_1, \widetilde{l}_{10}, \widetilde{R}_1, \widetilde{\mathcal{F}}_{1s}, \widetilde{\mathcal{F}}_{1w} \right\rangle$, we define the synchronous product of $\widetilde{A}_0$ and $\widetilde{A}_1$ as the symbolic transition system $\widetilde{A}_0 \parallel \widetilde{A}_1 \triangleq \left\langle \widetilde{V}_0 \cup \widetilde{V}_1, \widetilde{l}_{00} \wedge \widetilde{l}_{10}, \widetilde{R}_0 \wedge \widetilde{R}_1, \widetilde{\mathcal{F}}_{0s} \cup \widetilde{\mathcal{F}}_{1s}, \widetilde{\mathcal{F}}_{0w} \cup \widetilde{\mathcal{F}}_{1w} \right\rangle$. We present the symbolic synthesis algorithm in terms of manipulations on sets of states. A *state* in the setting of a symbolic transition relation $\widetilde{A}$ over a set of variables $\widetilde{V}$ can be considered a *valuation* of the variables in $\widetilde{V}$. Let $\mathfrak{S}_{\widetilde{V}}$ be the set of *all* valuations given a set of variables $\widetilde{V}$, *i.e.*, $\mathfrak{S}_{\widetilde{V}}$ is the set of all functions σ which maps each variable $v \in \widetilde{V}$ to a value of the appropriate type. For every predicate p over the variables $\widetilde{V}$, we denote by $|p| \subseteq \mathfrak{S}_{\widetilde{V}}$ the set of valuations that satisfy p. It follows that $|\mathtt{true}| = \mathfrak{S}_{\widetilde{V}}$. A predicate like $\widetilde{R}$ over the variables $\widetilde{V} \cup \widetilde{V}'$ can be viewed as a binary relation over $\mathfrak{S}_{\widetilde{V}}$, with the primed variables representing the second components of the pairs in the relation. Given a set $S \subseteq \mathfrak{S}_{\widetilde{V}}$ and a binary relation $R \subseteq \mathfrak{S}_{\widetilde{V}} \times \mathfrak{S}_{\widetilde{V}}$, we define $R \cap S$ as $R \cap S \triangleq R \cap (S \times \mathfrak{S}_{\widetilde{V}})$

Having established the correspondence between sets and relations with predicates, we define the operator $\mathsf{img}(P, R) \triangleq \{s \mid (s_0, s) \in R \wedge s_0 \in P\}$, where $P \subseteq \mathfrak{S}_{\widetilde{V}}$ and $R \subseteq \mathfrak{S}_{\widetilde{V}} \times \mathfrak{S}_{\widetilde{V}}$. If P and R are represented symbolically as $\widetilde{P}$ and $\widetilde{R}$ which are predicates over $\widetilde{V}$ and $\widetilde{V} \cup \widetilde{V}'$ respectively, then the symbolic equivalent of the img operator is given by:

$$\mathsf{unprime}\left(\exists\, v_1, v_2, \ldots, v_n \;\; \left(p \wedge \widetilde{R}\right)\right)$$

given that $\widetilde{V}$ is the set $\{v_1, v_2, \ldots, v_n\}$, and the function unprime substitutes primed variables with their unprimed versions. In a similar fashion, we define the pre-image operator $\mathsf{pre}(P, R) \triangleq \{s \mid (s, s_0) \in R \wedge s_0 \in P\}$, where P and R are as defined earlier in the case of the img operator. If P and R are represented symbolically as the predicates $\widetilde{P}$ and $\widetilde{R}$ as mentioned in the case of the img operator, then the symbolic equivalent of the pre operator is given by:

$$\exists\, v_1', v_2', \ldots, v_n' \;\; \left(\mathsf{prime}(p) \wedge \widetilde{R}\right)$$

given that $\widetilde{V}'$ is the set $\{v_1', v_2', \ldots, v_n'\}$ and the function prime substitutes each variable with its

primed version. Thus the img and pre operators yield the set of states (or valuations) reachable from a given set of states P within one forward or backward transition through R respectively. We define the operator img^* as:

$$\mathsf{img}^*(P, R) \triangleq P \cup \mathsf{img}(P, R) \cup \mathsf{img}(\mathsf{img}(P, R), R) \cup \cdots$$

Given that $\mathfrak{S}_{\widetilde{V}}$ is finite, the sequence of unions for img^* must converge to a fix-point after a finite number of unions. Thus $\mathsf{img}^*(P, R)$ represents the set of *all* states that are reachable from a given set of states P by zero or more forward transitions through R. Similarly, we define the set of all states that can reach a given set of states P by zero or more forward transitions through R by the function pre^* as:

$$\mathsf{pre}^*(P, R) \triangleq P \cup \mathsf{pre}(P, R) \cup \mathsf{pre}(\mathsf{pre}(P, R), R) \cup \cdots$$

Again, this sequence must also converge to a fix-point within a finite number of unions. Algorithm 3.1, GetSymbolicInterps, which has been adapted from the model checking algorithm presented in the work by Kesten et. al. [KPRS06][6] then describes the synthesis algorithm in terms of these definitions. Note that although the algorithm is described in terms of manipulations on sets, all of the operations have efficient implementations in the form of BDD manipulation routines, if we use BDDs as a symbolic representation for sets. For instance, we were able to translate these operations in a straightforward manner onto the operations provided by the BDD library CUDD [SB00], in the prototype that we developed.

3.4.1 Correctness

It has been shown in earlier work [KPRS06] that if the input $\widetilde{A}$ is not parameterized as it is in our case, then $\widetilde{A} \models \varphi$, and thus $A \models \varphi$ if and only if the value of new^{13}, *i.e.*, the value of the variable new after the execution of line 13 of Algorithm 3.1, is such that $\mathsf{new}^{13} \cap |\widetilde{\theta} \wedge \chi(\neg\varphi)| = \emptyset$. Given that in our setting, the symbolic transition relation $\widetilde{A}$ is parameterized by the set of variables $\widetilde{G}$, the set $\mathsf{new}^{13} \cap |\widetilde{\theta} \wedge \chi(\neg\varphi)|$ yields exactly the set of interpretations for $\widetilde{G}$ which result in $\widetilde{A} \not\models \varphi$. Therefore, by eliminating these interpretations from the set of all interpretations $\mathfrak{S}_{\widetilde{V}}$ in line 14 of Algorithm 3.1, and by the bi-directional implication in the proof of correctness of

[6] In fact, in the presentation here, the algorithm is *exactly* the same as that presented in [KPRS06] up-to and including line 13.

Algorithm 3.1: GETSYMBOLICINTERPS: Synthesize all correct FSM-SK completions

Input : A symbolic transition system $\widetilde{A} \triangleq \langle \widetilde{V}_A, \widetilde{l}_A, \widetilde{R}_A, \widetilde{\mathcal{F}}_{As}, \widetilde{\mathcal{F}}_{Aw} \rangle$, with parameters $\widetilde{G}$
An LTL property φ over $\widetilde{V}_A$.

Output : *All* interpretations $\mathcal{I}$ for parameters in $\widetilde{G}$, such that $A \models_{\mathcal{I}} \varphi$.

Data : $T_{\neg\varphi} \triangleq \langle \widetilde{V}_{\neg\varphi}, \widetilde{l}_{\neg\varphi}, \widetilde{R}_{\neg\varphi}, \widetilde{\mathcal{F}}_{\neg\varphi s}, \widetilde{\mathcal{F}}_{\neg\varphi w} \rangle$, the tester for the LTL formula $\neg\varphi$.
$\mathcal{D} \triangleq \langle \widetilde{V}, \widetilde{\theta}, \widetilde{R}, \widetilde{F}_s, \widetilde{F}_w \rangle$, where $\mathcal{D} = A \,\|\, T_{\neg\varphi}$.
new, old : subsets of $\mathfrak{S}_{\widetilde{V}}$
ρ : subset of $\mathfrak{S}_{\widetilde{V}} \times \mathfrak{S}_{\widetilde{V}}$

```
1  construct T_¬φ and 𝒟 as defined
2  old ← ∅
3  new ← img*(|θ̃|, |R̃|)
4  ρ ← |R̃| ∩ new
5  while new ≠ old do
6      old ← new
7      while new ≠ new ∩ pre(new, ρ) do
8          new ← new ∩ pre(new, ρ)
9      foreach F_w ∈ F̃_w do
10         new ← pre*((new ∩ F_w), (ρ ∩ new))
11     foreach (p, q) ∈ F̃_s do
12         new ← (new \ |p|) ∪ pre*((new ∩ |q|), (ρ ∩ new))
13 new ← pre*(new, ρ)
14 bad ← new ∩ |θ̃ ∧ χ(¬φ)|
15 return (𝔖_Ṽ \ bad)
```

the algorithm presented in the work by Kesten et. al [KPRS06], we immediately obtain the correctness of Algorithm 3.1.[7]

3.5 Evaluating the Symbolic Algorithm

To study how the symbolic synthesis algorithm presented in Section 3.4 performs in practice, we consider the cache coherence protocol called the Valid-Invalid (VI) protocol, which has been described in Chapter 1. This is one of the simplest cache coherence protocols, and is thus conveniently representable as a finite-state protocol. Nonetheless, it is qualitatively representative of the kinds of distributed protocols that we wish to target in this dissertation. We built a prototype tool in OCaml that implemented Algorithm 3.1, using the CUDD [Som15] BDD manipulation library as a back-end.

[7] Strictly speaking, the algorithm returns a set of valuations over $\widetilde{V}$, which includes variables other than those in $\widetilde{G}$, but nonetheless, because we have $\widetilde{V} \supseteq \widetilde{G}$ by construction, every valuation for $\widetilde{V}$ is also a valuation for $\widetilde{G}$.

3.5.1 Applying the Symbolic Algorithm to Complete the VI Protocol

We constructed a finite version of the VI cache coherence protocol to evaluate the symbolic synthesis algorithm shown in Algorithm 3.1. In the sequel, we assume that the Directory FSM-SK is of the form $D \triangleq \langle L_D, l_{D0}, I_D, O_D, R_D, \mathcal{F}_{Ds}, \mathcal{F}_{Dw} \rangle$, and each Cache FSM-SK C_i is of the form $C_i \triangleq \langle L_i, l_{i0}, I_i, O_i, R_i, \mathcal{F}_{is}, \mathcal{F}_{iw} \rangle$. We considered three instances of the completion problem:

- In the first version, the set of *tentative* transitions that were added to the directory machine is restricted to transitions of the form $\langle \textcircled{A}, \text{RSP}, \mathsf{guard}, l' \rangle$, for each $l' \in L_D$, where guard represents a fresh propositional variable to be solved for. Similarly, the set of tentative transitions added to each cache machine was restricted to transitions of the form $\langle \textcircled{B}, \text{INV}, \mathsf{guard}, l' \rangle$, for each $l' \in L_i$, where guard again represents a fresh propositional variable. In essence, the only synthesis that needs to be performed in this version is to determine the final state l' that each machine needs to transition to. In this case, symbolic algorithm was able to obtain a correct solution within 30 seconds.
- In the second version, the set of tentative transitions for the directory machine is restricted to transitions of the form $\langle \textcircled{A}, m, \mathsf{guard}, l' \rangle$, for every $m \in I_D \cup O_D$, and for every $l' \in L_D$. Similarly, the set of tentative transitions for the cache machine is restricted to transitions of the form $\langle \textcircled{B}, m, \mathsf{guard}, l' \rangle$, for every $m \in I_i \cup O_i$ and for every $l' \in L_i$. This is tantamount to determining what *message* to send or receive when at the locations labeled $\textcircled{A}$ and $\textcircled{B}$, as well as the next state to transition to after sending or receiving the message. In this case, the symbolic algorithm was able to converge on a correct solution in about ten minutes.
- In the third and final version, the set of tentative transitions for the directory machine includes *all* transitions of the form $\langle l, m, \mathsf{guard}, l' \rangle$, for every $m \in I_D \cup O_D$, and for every $l, l' \in L_D$. Similarly, the set of tentative transitions for the cache machine is restricted to transitions of the form $\langle l, m, \mathsf{guard}, l' \rangle$, for every $m \in I_i \cup O_i$ and for every $l, l' \in L_i$. These tentative transitions are added only if they do not result in non-determinism in the FSM-SKs. In this version, the completion algorithm does not have any knowledge about the starting locations of the missing transitions are, what messages to send or receive in the starting locations, as well as what the final locations of the transitions are. For this version of the problem, the symbolic algorithm was unable to obtain a correct solution even after six hours of computation time.

To summarize, the three versions of the protocol completion problem for the VI coherence protocol differ in the amount of programmer intuition conveyed to the algorithm. The observation here is that the symbolic algorithm performs better when the search space of solutions is restricted by leveraging the intuitions that a programmer has. We now discuss the reasons for why the algorithm does not scale in the hardest of cases, as well as elaborate on some of the insights obtained from this experiment.

BDDs and the Scalability of the Symbolic Algorithm

In our experiments with Algorithm 3.1, we observed that the BDDs often consisted of tens to hundreds of millions of nodes. The calls to BDD manipulation routines sometimes required tens of minutes of computation. We experimented with enabling the dynamic reordering of BDDs in CUDD. This helped keep the size of the BDDs manageable and enabled quick completion of the BDD manipulation routines. However, the cost of this was that every time the dynamic reordering was triggered, it often took tens of minutes to complete the reordering, based on the internal heuristics implemented in the CUDD library. To summarize, enabling dynamic reordering did not have a positive impact overall execution time of the algorithm. Given that the BDDs were being used to represent constraints over a set containing about 600 variables in the hardest versions of the VI completion problem, we could not exhaustively evaluate all possible static variable orderings. We did however, experiment with a few static variable orderings, that we believed were reasonable, but were unable to improve the execution times of the algorithm.

Impact of Symbolically Retaining all Solutions

Figure 3.1 depicts the reachable state space of the protocol in terms of the interpretation that is chosen (in this case, parameter valuations), at a conceptual level. We have empirically observed that checking if a *correct* version of the VI protocol satisfies all the LTL specifications can be performed rather efficiently,[8] and requires only a few seconds of computation time, even with a static BDD variable ordering. From this observation, we infer that the region G is amenable to being represented compactly using BDDs. However, Algorithm 3.1, first computes the region U, and then computes G, by removing all states (and the parameter valuations which led to their conditional reachability) that can reach an erroneous state in one or more steps. We

[8]This can be accomplished by simply executing Algorithm 3.1 until (and including) line 13, and checking that $new \cap |\widetilde{\theta} \wedge \chi(\neg\varphi)|$ is empty.

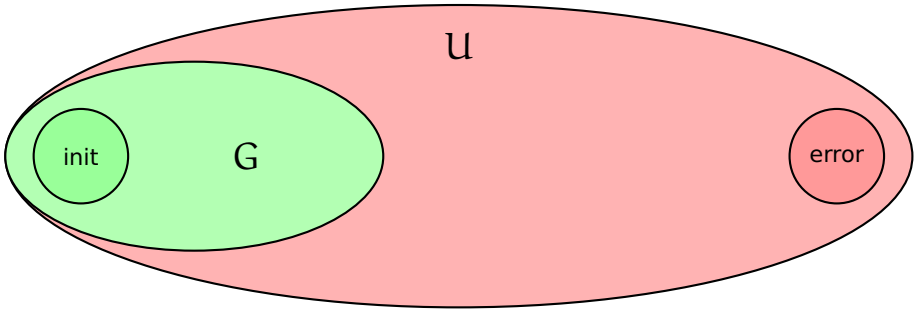


Figure 3.1: Depiction of the state space of the protocol in terms of all possible completions. The region marked U, which includes all other regions is the state space of the protocol which is reachable if the set of parameter valuations is left unconstrained, *i.e.*, this is the region that is the union of the reachable state space for every possible completion. The region marked G, which includes the region marked init, consists of the set of states of the protocol that are reachable if a *good* completion is chosen. The set U \ G denotes the set of states that are reachable if a *bad* completion is chosen. These are states that can reach an error state in zero or more steps, under a given *bad* completion.

have empirically observed that the BDDs representing G are compact. We thus conclude that representing the large parts of the set U that are conditionally reachable, together with the parameter valuations which ensure their reachability is difficult using BDDs.

To try and reduce the size of the BDDs representing these intermediate results, our implementation differs slightly from Algorithm 3.1, in the following ways:

- We separate LTL specifications that are *safety* specifications, *i.e.*, of the form **G** p, from true liveness specifications, which could involve eventualities.
- We aggressively eliminate interpretations that are proven unsafe, as early as possible, during the execution of the algorithm. Specifically, the computation in line 3 in Algorithm 3.1 is interleaved with steps to eliminate incorrect interpretations. This is done by eliminating parameter valuations that cause the currently computed under-approximation of the set of reachable states to have a non-empty intersection with the set of states where the invariant is violated.

Unfortunately, this optimization did not have much effect on the execution time of the algorithm, owing to two reasons:

1. We only eliminate a parameter valuation when it has been *proven* to reach an unsafe state. As can be seen from Figure 3.1, there is a large set of states marked U \ G, which will inevitably lead to an unsafe state, but might need several steps to do so. This causes our algorithm to retain large parts of the set U \ G as a function of the parameter valuations in a

symbolic form. And we have already discussed that this space is not compactly representable using a static BDD ordering. The problems with enabling dynamic reordering have also been discussed earlier.

2. The aggressive pruning only prunes parameter valuations which violate some *safety* specification. A large part of the specifications for the VI protocol are liveness specifications. We have empirically observed that even after pruning unsafe parameter valuations, the BDDs that evolve during the execution of the loop on line 5 of Algorithm 3.1 are often huge.

Based on these observations, we concluded that this symbolic approach, while very elegant, was unlikely to perform well in practice on larger, more complex protocols. We conclude the discussion on this symbolic synthesis strategy by summarizing some key insights which influenced the direction of the research described in the rest of this dissertation.

3.5.2 Insights from Experimenting with the Symbolic Algorithm

- Starting with the set of all possible solutions and paring it down to the set of correct solutions is difficult, especially if the state space of the protocol is maintained symbolically as a function of the current over-approximation of the set of correct solutions. More effective algorithms are possible if we require the algorithms to find *one* correct solution, rather than *all* of them, as we show in subsequent chapters of this dissertation.
- Symmetry in the state space cannot easily be exploited to reduce the size of BDDs. Although there has been work along this direction [CJEF96, EW03, EW05, WBE08], most of these techniques are geared towards checking CTL properties, and not LTL properties with fine-grained fairness assumptions. The problem is that symbolically representing the *orbit* relation between states which are equivalent modulo the symmetry assumptions requires an exponentially sized BDD, which negates any savings obtained by eliminating symmetric states.
- Explicit state model checking techniques seem more promising than symbolic techniques for synthesis. Counterexample Guided Inductive Synthesis [SLRBE05, STB$^+$06, SAT$^+$07, Sol09] can more readily be applied when using explicit state model checking techniques, as we show in the rest of this dissertation. Further, symmetry in the state space of the protocol can also be more effectively exploited, leading to exponential space savings [ID96, Dil96, ES97, SGE00].
- If a CEGIS technique is used, then a purely depth-first or breadth-first approach during the verification (or model-checking) phase is sub-optimal. We have observed this empirically in

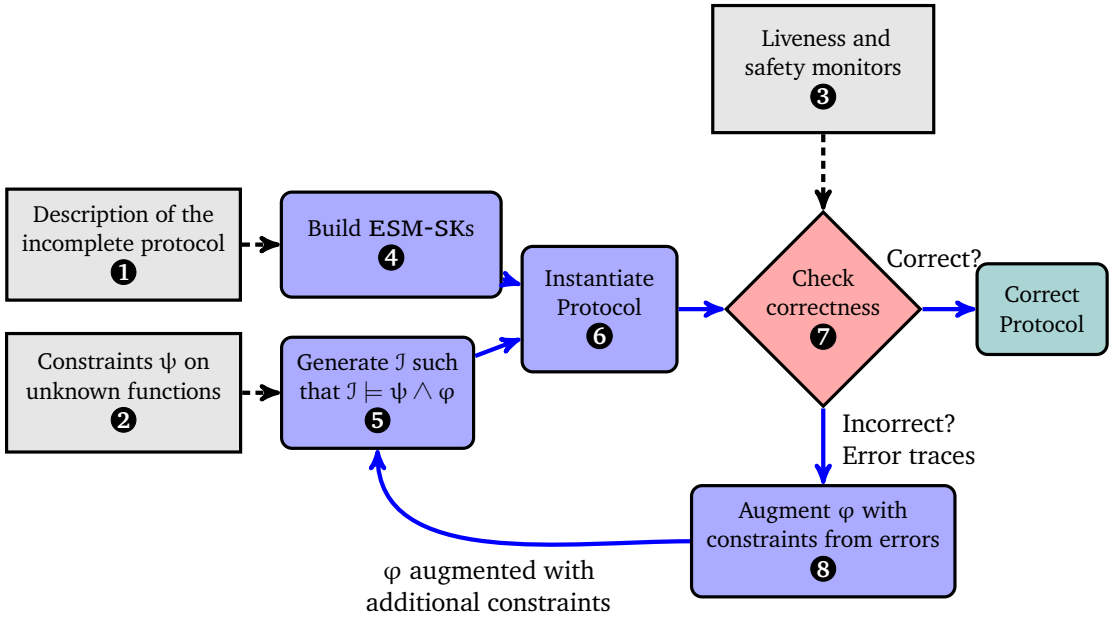


Figure 3.2: Algorithmic scheme of all the solution strategies we discuss. The gray rectangles represent inputs, the blue rounded rectangles represent computation, and the red rhombuses represent decisions. Solid blue arrows represent control and data flow, while the dashed black arrows represent data flow.

the case of the symbolic algorithm, which uses a symbolic, breadth-first search strategy. In the later chapters of this dissertation, we explore heuristics for explicit state model checking algorithms, which lead to quicker convergence of the CEGIS loop to find an interpretation that satisfies the requirements described in Section 2.3.

We conclude this chapter with a brief discussion of the solution strategies described in the rest of this dissertation.

3.6 Road-map for the Rest of the Dissertation

In our attempt to find a complete solution to the problem defined in Section 2.3, we solved several simplified versions of the problem, each progressively less simplified. Each of these techniques will be described in subsequent chapters, culminating in a complete solution for the problem defined in Section 2.3 in Chapter 8. Each of these approaches tries to solve a particular aspect of the problem and is interesting in its own right, in addition to bringing us a step closer to a complete solution. Figure 3.2 describes the general scheme of the algorithm used in our solution strategies. Every one of the solution strategies which we shall present

in the rest of this manuscript may be viewed as an instantiation of the algorithmic scheme shown in Figure 3.2. The block labeled ❶ represents the description of the incomplete protocol provided by the user. This can be an ESM sketch itself or in some other form, for example, *flows* or *scenarios* [TT08], or message sequence charts [ITU96] or live sequence charts [DH01], from which the ESM sketch is built by the block labeled ❹. The user can also specify a set of constraints ψ representing domain knowledge, as shown in the block labeled ❷, which is taken into consideration while generating a suitable interpretation in the block labeled ❺. The set of constraints φ is initially empty. Once a suitable interpretation $\mathcal{I}$ has been generated, the protocol is instantiated with this interpretation by the block labeled ❻, and checked for correctness against the user specified safety and liveness monitors (❸) by the block labeled ❼. This check is performed using a model-checker. If the protocol is found to satisfy all the safety and liveness requirements, then the algorithm terminates with a success. On the other hand, if the model-checker discovers errors, these errors are used to generate additional constraints (❽) which are then added to φ. These additional constraints rule out *at least* the current interpretation $\mathcal{I}$ from being generated again. A new interpretation $\mathcal{I}'$ is now generated taking into account the newly added constraints, and this process is repeated until a correct protocol is found. The solution strategies which we describe in this manuscript differ primarily in how the incomplete protocol is described (❶), how errors are analyzed to obtain new constraints to augment φ with (❽), and in how new interpretations are generated (❺).

Our first attempt, which resulted in a system called TRANSIT [URD+13], applied the following restrictions on the problem statement: (1) Only safety monitors were used, (2) We required the protocol designer to specify the behavior of functions in $\mathfrak{U}$ using any combination of input-output examples, and symbolic constraints, and finally, (3) we required the interpretations synthesized to only be consistent with the constraints specified by the protocol designer, and not necessarily result in a correct protocol. Upon encountering an erroneous trace, the programmer could specify the correct behaviors of the relevant functions in $\mathfrak{U}$ using purely concrete input-output examples corresponding only to the erroneous trace in question. Essentially, the approach required the programmer to be in the loop with the synthesizer, providing additional information, whenever the synthesis step resulted in an incorrect protocol. The approach is an instantiation of the algorithm described in Figure 3.2, where the block labeled ❶ is in itself an ESM-SK, the block labeled ❷ represents input-output examples or symbolic constraints, each of which refers to only *one* function $f_u \in \mathfrak{U}$. The block labeled ❸ is restricted to only

contain safety monitors, and finally, the task performed by the block labeled ❽ is performed by the user or programmer by adding the relevant input-output examples to φ. This work is described in Chapter 4.

While we were building the tool TRANSIT, we realized that the synthesis problem of synthesizing a function interpretation given symbolic constraints and input-output examples could be generalized in a manner that encompasses most other custom-built synthesizers for various domains. This observation resulted in the formalization of the SyGuS problem, and a competition of the same name. We built a general-purpose solver for the SyGuS problem, based on the ideas used in the solver built for TRANSIT, which won the SyGuS competition in 2014 [AFSSL14]. Chapter 5 provides a detailed description of the SyGuS problem, and Chapter 6 describes an enumerative solution strategy, and also builds more sophisticated and scalable algorithms based on the simple enumerative solver.

Our second solution strategy allowed full use of safety and liveness monitors, but we restricted the ESMs and the ESM-SKs to not have any state variables, as described earlier in this Chapter. In essence this meant that all state had to be encoded through the locations of the state machines, which were essentially *finite* state machines or *finite* state machine sketches — FSMs and FSM-SKs, as described earlier. Further, we also did not allow messages to have payloads. However, we relaxed the requirement that an FSM-SK be provided, and instead allowed the user to specify the known behavior of the protocol using *scenarios* or *flows* [TT08], which were then compiled into FSM-SKs. The problem then is essentially to find a set of transitions to add to the given finite-state sketches, such that the composition satisfies the provided monitors. We were able to build a system which worked completely automatically, and the system produced correct protocols, starting from a set of scenarios which described the protocol behavior in the common cases. Relating this to Figure 3.2, ❶ is now a set of scenarios or flows, and block ❽ is an algorithm that analyzes counterexample traces and adds appropriate constraints on the functions in $\mathfrak{U}$. Also, $\mathfrak{U}$ contains functions, each of whose domain the set of locations L_i of the appropriate FSM-SK in the composition A, and whose range is simply a Boolean which indicates whether the particular transition is allowed or not. We describe this approach in detail in Chapter 7.

Our third strategy treats symmetry — as defined in Section 2.3 — as a first class citizen and also allows ESMs and ESM-SKs to have typed state variables. We expect the programmer to provide the description of the protocol directly as ESM-SKs, *i.e.*, ❶ is a set of ESM-SKs. We

also support liveness requirements in the form of Büchi monitors, which are required to be satisfied under fine-grained fairness assumptions set forth by the programmer. This approach is fully automatic, and is thus a complete solution to the problem defined in Section 2.3. We discuss this approach in detail in Chapter 8. Chapter 9 discusses closely related work, both in the area of distributed protocol synthesis as well as general program synthesis. Finally, we reflect on the research problems addressed in this dissertation, the limitations of the solution strategies, avenues for further research and conclude with Chapter 10.

4

TRANSIT: Specifying Protocols with Concolic Snippets

This chapter describes a tool called TRANSIT, which we have built and evaluated as a solution to a restricted version of the problem defined in Section 2.3, and uses *concolic snippets* to describe the behavior of ESM-SKs. This chapter is based on the work originally published in [URD+13].

4.1 Overview of TRANSIT

TRANSIT allows a programmer to specify the known behavior of the protocol, using *concolic snippets*. These are sample transition fragments which describe the guards and updates of a *single* transition of an ESM-SK using constraints on the valuations of the variables of an ESM. The constraints can be (1) concrete — in which case they describe the behavior of the guard and updates on exactly one valuation of the ESM variables, (2) symbolic — in which case they describe the behavior of the transition as the post-condition which must hold whenever the specified precondition holds, or (3) they can be any combination of the concrete and symbolic constraints.

The motivation for the use of concolic snippets is that during the initial design and development phase, the programmer can use symbolic values to describe the part of the behavior of the protocol that well understood. Once an initial — and possibly incomplete — version of the protocol has been specified using symbolic snippets, it is then checked for correctness. The programmer can then codify the fixes to any counterexamples obtained during this check using concrete input-output examples that correspond to a *local* fix, which eliminates at least

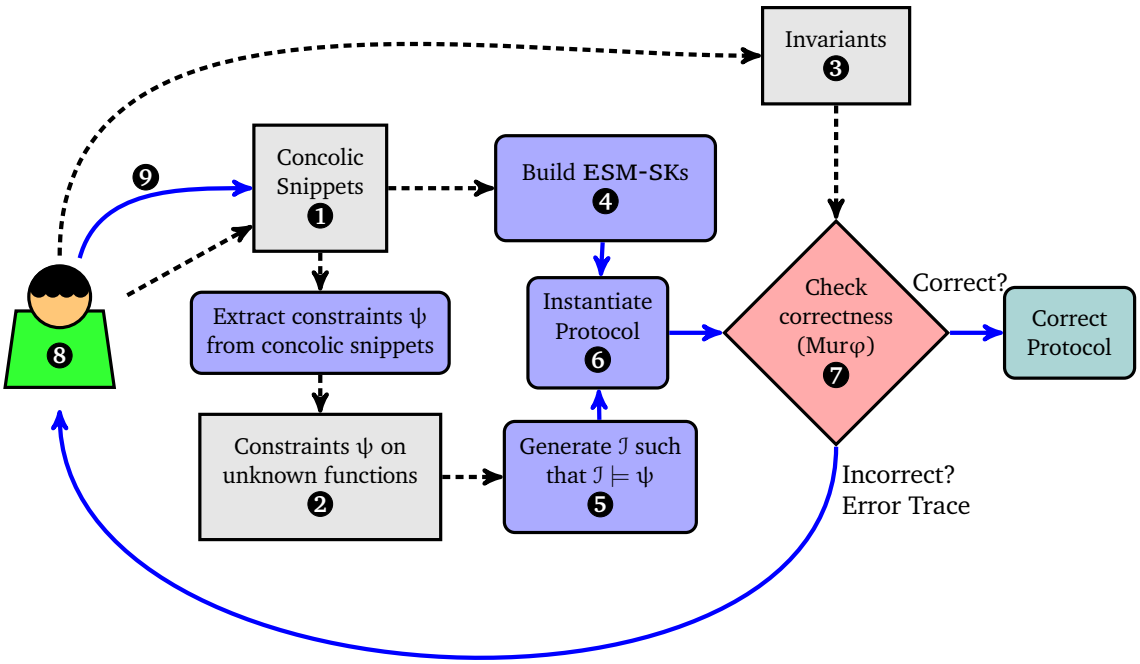


Figure 4.1: Overview of developing a protocol with TRANSIT. The algorithm is an instantiation of the algorithm shown in Figure 3.2, with the programmer analyzing error traces. The arrow labeled ❾ denotes the concolic snippets that the programmer provides to eliminate at least the current error trace in question. These concolic snippets are then used to augment the constraints ψ on the unknown functions. This process is repeated until a correct protocol is obtained.

the one counterexample. TRANSIT integrates the new (concrete) constraints with the rest of the constraints to provide a new interpretation $\mathcal{I}$ which satisfies *all* the constraints. The protocol is instantiated with the new interpretation, and the process is repeated until a correct protocol is found. The programmer is thus freed from reasoning about the global properties of the protocol when handling corner-case behavior, which were not handled in the initial version of the protocol.

Figure 4.1 provides a high-level view of the working of the TRANSIT system. It is an instantiation of the algorithmic scheme shown in Figure 3.2: the inputs are in the form of *concolic snippets*, which we will define shortly. The relevant constraints ψ on the unknown functions — functions $f_u \in \mathfrak{U}$ — are extracted from these snippets by TRANSIT; the box labeled ❷ is thus not directly provided by the programmer. The task of analyzing counterexamples and

inferring additional constraints on the unknown functions — represented by the block labeled ❽ in Figure 3.2 — is not automatic, but it instead performed by the programmer in TRANSIT. Also, TRANSIT only supports safety properties: this is not a methodological limitation, but rather due to the limitations of the model checker that we use, Murφ, which does not support the checking of liveness properties.

The synthesis algorithm presented in this chapter, assumes a specific form for the constraints ψ. We require that ψ be a conjunction of constraints $\psi \triangleq c_1 \wedge c_2 \wedge \cdots \wedge c_n$, where each conjunct c_i can be an arbitrary Boolean valued expression, but has the following properties:

- The expression, c_i refers to *exactly one* unknown function $f_u \in \mathfrak{U}$.
- The expression c_i is assumed to refer only to the set of variables $V \cup \{o\}$, where V is the set of *all* state variables of the ESM-SK which uses f_u in its description.
- All applications of f_u in c_i are of the form $f_u(V)$, *i.e.*, every occurrence of f_u in c_i has f_u applied to the *same* set of arguments, and these arguments comprise *all* the state variables of the ESM-SK which uses f_u in its description, in the same order.

These restrictions essentially ensure that the constraint ψ is *separable*, a notion which will be defined in Chapter 6. For the purpose of exposition in this chapter, we note that these restrictions essentially have the following consequences:

- For each $f_u \in \mathfrak{U}$, we can find the subset of conjuncts ψ_{f_u} in ψ that refer to f_u, and these conjuncts now form the all the constraints that an interpretation for f_u needs to satisfy. We can therefore synthesize interpretations for each $f_u \in \mathfrak{U}$ independently.
- Within each ψ_{f_u} corresponding to the constraints on $f_u \in \mathfrak{U}$, we know that all occurrences of f_u have the form $f_u(V)$, where V is the set of all state variables of the ESM-SK that refers to f_u. We can thus replace each of these occurrences with a single distinguished variable o, which has the same type as the range of f_u.

The syntax that we use for concolic snippets allow TRANSIT to easily translate the concolic snippets into a constraint ψ that has the required form.

The rest of this chapter is organized as follows: Section 4.2 explains what a concolic snippet snippet is and describes, by means of examples, how they are used in programming with TRANSIT. Section 4.3 describes an algorithm for synthesizing symbolic expressions such that they are consistent with the concolic snippets provided by the programmer. Finally, Section 4.4 presents and discusses the results of experimentally evaluating of TRANSIT to specify a few cache coherence protocols as case studies, our experience with TRANSIT, as well

```
Transition(CurrentState, InputEvent)
  [optional guard] => (NextState, OutMsg)
      Pre_1 ==>
          Post_11;
          Post_12;
          ...
      Pre_2 ==>
          ...
          ⋮
```

Figure 4.2: A concolic snippet. *CurrentState* and *NextState* are the start and end control states. The snippet specifies zero or more outbound messages. It also specifies a *guard-action* block for each *guard* containing a set of conditional updates. The expression Pre_i specifies the condition (on process variables and the fields of the received message) under which the Boolean constraints Post_{ij} hold. Each Post_{ij} constrains the updated value of exactly one process variable or output message field in terms of the old values of the process variables and the fields of the received message.

as the limitations and shortcomings of this methodology.

4.2 Concolic Snippets and Programming with TRANSIT

Figure 4.2 shows the ingredients of a concolic snippet expressed in the TRANSIT language. We compare the elements in Figure 4.2 with the notation for a transition $\mathtt{t} \triangleq \langle \mathtt{l}, \mathtt{m}, \mathsf{guard}, \mathsf{updates}, \mathtt{l}' \rangle$ which we set up in Section 2.2. For clarity, a "`Transition`" — note the `mono-spaced` font — refers to the identically named construct in the TRANSIT language, whereas a "transition" — note the serif-ed font — refers to the notion of a transition formalized in Section 2.2. *CurrentState* represents the initial location for the transition, $\mathtt{l}$ in our notation. A `Transition` in TRANSIT groups together *all* transitions which begin from a given initial location *CurrentLocation* $\in \mathtt{L}$, *i.e.*, a `Transition` in the TRANSIT language represents *all* transitions of the form $\mathtt{t} \triangleq \langle \mathit{CurrentState}, \mathtt{m}, \mathsf{guard}, \mathsf{updates}, \mathtt{l}' \rangle$, which begin at *CurrentState*. Within each `Transition`, we have multiple *guard blocks*, one for each transition which begins at location *CurrentState*. The *InputEvent* describes the input message $\mathtt{m}$ (or ϵ) which triggers the transition. Within each guard block, the guard is optional. If left unspecified, it will be synthesized by TRANSIT. Each guard block specifies a *NextState*, which is $\mathtt{l}'$ in our notation, as well as an optional output message *OutMsg*. TRANSIT allows the programmer to *fuse* a transition involving the receipt of a message with a transition involving transmitting an output message, *i.e.*, an input or internal transition, followed immediately by an output transition.

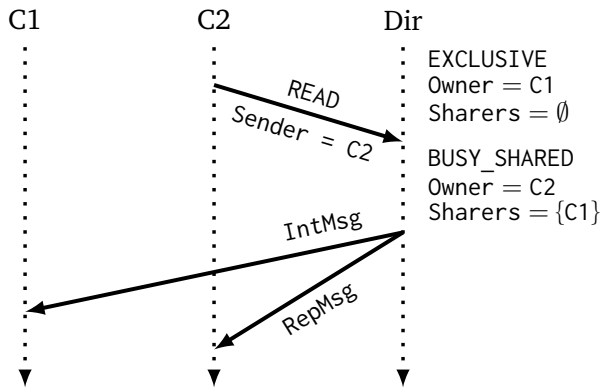


Figure 4.3: An error trace generated by the model checker in response to an incorrect completion for the SGI-Origin protocol synthesized by TRANSIT. Here, "C1" and "C2" represent two cache ESMs and "Dir" represents the directory ESM. Time progresses downward along the dotted arrows (lanes) and the state of each ESM after each transition is annotated along the lanes. Message exchanges and their contents are described using annotated arrows across lanes.

Within each guard block, TRANSIT allows multiple *Pre-Post* blocks. The semantics of each such block of the form $Pre_i \Rightarrow Post_{i1}; Post_{i2}; \ldots Post_{im};$, where Pre_i is a Boolean valued expression on the state variables and incoming message fields, $V \cup \{m_p\}$, and $Post_{ij}$ constrains the value of exactly *one* state variable or a field of the output message m_p, are that if Pre_i holds at the beginning of the transition, then $Post_{ij}, j \in [1, m]$ must hold after the transition has executed.

4.2.1 Using Snippets in TRANSIT

To illustrate the use of snippets in TRANSIT, we use an anecdote from our case study of implementing the SGI-Origin cache coherence protocol [LL97] from published informal textual rules. A directory-based cache coherence protocol, such as the SGI-Origin protocol, ensures that the copies of data maintained in the private caches of a multi-processor system are kept consistent. The protocol has a distinguished "Directory" process, which maintains the global view of the processors in the system which currently have a copy of the data in the cache. The "Cache" processes coordinate with the Directory process via exchange of messages whenever they need to perform read or write operations on a block of data. The cache and the directory processes are typically modeled as ESMs, and we refer to them as processes or ESMs interchangeably. In the SGI-Origin protocol, the directory ESM has the variable `Sharers`, whose type is a finite set of cache process (or ESM) identifiers, and a variable `Owner`, whose type is a cache process identifier. The `Sharers` variable maintains the set of caches which

have a read-only copy of the data, whereas the value of the `Owner` variable, if defined, is the identity of the sole cache ESM in the system which has a read-write copy of the data. The safety property that every cache coherence protocol needs to satisfy is the *coherence property*, which states that the value *read* by an cache ESM is the same as the value written by the *most recent* write operation by any cache ESM in the system.

One of the textual rules from the paper [LL97], describing the behavior of the directory ESM in the SGI-Origin protocol, on receiving a read request from a cache ESM reads:

> *If directory state is Exclusive with another owner, transitions to Busy-shared with requester as owner and send out an intervention shared request to the previous owner and a speculative reply to the requester. Go to 5b.*

Note that this description does not specify how the `Sharers` variable needs to be updated. The programmer indicates that the new value of the `Sharers` variable needs to contain *at least* the sender of the message received, in addition to the old contents of the `Sharers` variable. This is codified in TRANSIT using the following *concolic snippet*:

```
Transition(EXCLUSIVE, ReqNet Msg) {
  [] => (BUSY_SHARED, RepNet RepMsg, IntNet IntMsg) {
      (Msg.Type = READ & Msg.Sender != Owner) ==> {
          SubsetOf(SetUnion(Sharers, {Msg.Sender}), Sharers');
          ...
}}}
```

Note that this snippet is *active* only when the current location of the directory ESM is `EXCLUSIVE` and a request message is received. It also specifies that the location to transition to is `BUSY_SHARED`, and that the a reply message as well as an intervention message is to be transmitted, as required by the textual rule. The `ReqNet`, `RepNet` and `IntNet` declarations indicate the specific channels these messages are sent out over. This is a technicality required by the TRANSIT language is not particularly relevant to the ideas we describe, so we ignore it in the rest of the manuscript. To be consistent with this snippet, TRANSIT needs to generate code for the update of the state variable `Sharers` such that the new value of the variable (denoted by the primed variable `Sharers'`) is the super-set of union of the old value of the variable and the sender of the message. Based on the snippet provided by the user, suppose that TRANSIT generated the following code for the update of the `Sharers` variable for the transition in question:

$$\texttt{Sharers} := \texttt{Sharers} \cup \{\texttt{Msg.Sender}\}$$

An attempt to verify the protocol instantiated with this update, results in a violation of the coherence invariant. A visual representation of a simplified version of the error trace is shown in Figure 4.3. Observe that the transition shown on the directory ESM in Figure 4.3 is a concrete instance of the concolic snippet that we have described, with the cache ESM C2 being Msg.Sender and the Owner variable of the directory ESM Dir initially set to C1. Upon inspecting the error trace, the programmer recognized that in this particular case, the new value of the Sharers variable needed to include the previous value of the Owner variable as well. The programmer codifies this using the following *concrete snippet*:

```
Transition(EXCLUSIVE, ReqNet Msg) {
  [] => (BUSY_SHARED, RepNet RepMsg, IntNet IntMsg) {
      (Msg.Type = READ & Msg.Sender = C2 & Owner = C1) ==> {
          Sharers' = {C1, C2};
          ...
}}}
```

Observe that this snippet is only applicable in the specific case when the directory receives a READ request from cache C2, and the owner is cache C1. The programmer has not applied any global reasoning to come up with this snippet. With this additional snippet, TRANSIT generated a new implementation with the correct update for the Sharers variable as:

$$\texttt{Sharers} := \texttt{Sharers} \cup \{\texttt{Msg.Sender}, \texttt{Owner}\}$$

To sum up, TRANSIT allows a snippet to be (1) completely symbolic, in which case the constraints on each lvalue are simple equalities, and TRANSIT does not attempt to synthesize code for such snippets, instead treating them as the implementation itself, or (2) concolic, in which case, each of the constraints $\texttt{Post}_{ij}$ has no restriction on its form, but only constrains *one* lvalue, or (3) concrete, in which case, it still constrains *one* lvalue, but concrete values are used in both the pre- and the post-conditions in the constraints.

4.3 Expression Inference

To construct a protocol from the concolic snippets provided by the programmer, TRANSIT needs to synthesize expressions which are *consistent* with each of the snippets provided by the programmer. Let us, for the moment, assume that constraints implied by the programmer in the concolic snippets can be translated precisely into a constraint ψ, of the form described in

Section 4.1. We will return to the question of how this translation is accomplished towards the end of this section. For the purpose of the synthesis algorithm, we treat the constraints obtained from both concrete and concolic snippets in the same way, *i.e.*, as symbolic constraints over the expressions to be synthesized. For each unknown function $f_u \in \mathfrak{U}$, we can separate the constraints on f_u into a set of conjuncts ψ_{f_u}, where each conjunct refers to the set of variables V, of the ESM-SK that refers to f_u, and a *distinguished* output variable $o \notin V$. The variable o, which corresponds to the lvalue being updated. Consider the set of constraints ψ_{f_u} for one $f_u \in \mathfrak{U}$. Let us call the conjunction of the constraints in ψ_{f_u} as $\mathcal{C}$. The expression inference problem thus corresponds to the following computational problem: Given a quantifier free formula $\mathcal{C}$ over a set of typed (ESM) variables $V \cup \{o\}$, find a symbolic expression e, which refers only to variables in V, such that $\mathcal{C}[o \mapsto e]$ is valid, *i.e.*, $\neg\, \mathcal{C}[o \mapsto e]$ is not satisfiable. Here the notation $\mathcal{C}[o \mapsto e]$ denotes that every application of the function o in $\mathcal{C}$ is syntactically replaced by the expression e. Note that we can synthesize expressions for each unknown guard or update function independently because each post-condition in a snippet is required to constrain the value of *exactly one* lvalue, which corresponds to the distinguished output variable o, described earlier.

We assume a fixed vocabulary of function symbols $\mathfrak{F}_v \subseteq \mathfrak{F}$ (with fixed, known interpretations) using which the expression e is to be constructed, *i.e.*, e is a well-typed composition of function symbols in $\mathfrak{F}_v$, applied to the variables in V. The instantiation of the set of types $\mathcal{T}$ in the context of TRANSIT is the finite set of types which includes the types of all the variables in the system. This includes (1) The type Int representing integers, (2) The type Bool, which represents the Boolean type, (3) The type PID, which represents the set *process identifier*, one for each state machine in a protocol, and (4) The type Set, values of which represent *sets* of process identifiers, *i.e.*, sets of values of type PID. The type PID is implemented as a bit-vector. Table 4.1 shows the signatures and semantics of the function symbols used in the instantiation of $\mathfrak{F}_v$ in the implementation of TRANSIT. Thus the search space for an expression e is simply the set of all well-typed function compositions using functions symbols in $\mathfrak{F}_v$ applied to variables in V. Our algorithm for inferring expressions enumerates expressions from this space, in increasing order of the syntactic size of the expressions.

Consider a valuation σ of the set of variables V. Recall that a candidate expression e to be substituted for the output variable o, is built from function symbols whose interpretations are fixed and known and from variables in V. So, given a valuation σ, we can evaluate the

Function	Description
add (Int, Int) $\rightarrow$ Int	Integer Addition
sub (Int, Int) $\rightarrow$ Int	Integer Subtraction
inc (Int) $\rightarrow$ Int	Add one to an Integer
dec (Int) $\rightarrow$ Int	Subtract one from an Integer
setadd (Set, PID) $\rightarrow$ Set	Add an entry into a Set
setsize (Set) $\rightarrow$ Int	Cardinality of a Set
setunion (Set, Set) $\rightarrow$ Set	Set Union
setinter (Set, Set) $\rightarrow$ Set	Set Intersection
setminus (Set, Set) $\rightarrow$ Set	Set Difference
setof (PID) $\rightarrow$ Set	Create a singleton Set
or (Bool, Bool) $\rightarrow$ Bool	Boolean Disjunction
and (Bool, Bool) $\rightarrow$ Bool	Boolean Conjunction
not (Bool) $\rightarrow$ Bool	Boolean Negation
setcontains (Set, PID) $\rightarrow$ Bool	Membership test on a Set
iszero (Int) $\rightarrow$ Bool	Test if an integer is Zero
$\forall$t $\in \mathcal{T}$ equals (t, t) $\rightarrow$ Bool	Equality Test
ge (Int, Int) $\rightarrow$ Bool	Greater than or equal to
gt (Int, Int) $\rightarrow$ Bool	Greater than
$\forall$t $\in \mathcal{T}$, ite (Bool, t, t) $\rightarrow$ t	Conditional Expression
numcaches () $\rightarrow$ Int	# of Caches (constant)

Table 4.1: Expression Vocabulary used in Coherence Protocols

expression e over σ. Given a set of variables V, we denote by $\mathcal{S}_V$ the set of *all* valuations of V, as in Chapter 2. We denote the value of an expression e evaluated with the variable valuation σ as $e|_\sigma$. Given an ordered list of valuations $P \triangleq \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$, we define the *signature* of an expression e with respect to P as $\mathsf{signature}(e, P) \triangleq \langle e|_{\sigma_1}, e|_{\sigma_2}, \ldots, e|_{\sigma_n} \rangle$. Now, if two expressions e and e' have the same signature on a list of valuations P, then (1) either they have the same signature on *all* possible valuations $\sigma \in \mathcal{S}_V$, in which case, e and e' are equivalent, or, (2) there must be some valuation $\sigma \in \mathcal{S}_V$ which serves to distinguish e and e'. We use this observation to prune the search space of expressions.

Algorithms 4.1 and 4.2 describe the enumerative algorithm to infer an expression e consistent with a Boolean valued constraint $\mathcal{C}$. Algorithm 4.1, SynthForPoints, synthesizes an expression e such that $\mathcal{C}[o \mapsto e]$ satisfies the constraints *at least* for a given set of valuations P. It accomplishes this by a dynamic programming strategy. It begins by enumerating expressions of size zero — variables $v \in V$ and functions of arity zero, *i.e.*, constants in our setting. For each expression e that the algorithm considers, it computes $\mathsf{signature}(e)$ and determines if an

Algorithm 4.1: SYNTHFORPOINTS: Synthesize an expression consistent with a set of inputs

Input : An ordered list of valuations P.
An expression vocabulary $\mathfrak{F}_v$ over a set of types $\mathcal{T}$.
A set of typed variables $V \cup \{o\}$.
A constraint $\mathcal{C}$ over $V \cup o$.

Output : An expression e such that for every valuation $\sigma \in P$, $\mathcal{C}[o \mapsto e]\big|_\sigma = \texttt{true}$.

Data : $\mathsf{exps}_{t,j}, t \in \mathcal{T}, j \in \mathbb{N}^+$, which are sets of expressions of type t and size j, initially empty.
A set sigs which contains the signatures of expressions over P, initially empty.

```
1  baseexps ← {v ∈ V} ∪ {c ∈ F_v : arity(c) = 0}
2  foreach e ∈ baseexps do
3      s ← signature(e, P)
4      if s ∈ sigs then
5          continue
6      if ∀σ ∈ P. (C[o ↦ e]|_σ) then
7          return e
8      t ← typeof(e)
9      sigs ← sigs ∪ {s}
10     exps_{t,1} ← exps_{t,1} ∪ {e}
11 i ← 2
12 while true do
13     foreach f ∈ F_v do
14         m ← arity(f)
15         ⟨t_1, t_2, ..., t_m⟩ ← dom(f)
16         foreach m-partition ⟨r_1, r_2, ..., r_m⟩ of i − 1 do
17             foreach (e_1, e_2, ..., e_m) ∈ Π_{j=1}^{m} exps_{t_j,j} do
18                 e ← f(e_1, e_2, ..., e_m)
19                 s ← signature(e, P)
20                 if s ∈ sigs then
21                     continue
22                 if ∀σ ∈ P. (C[o ↦ e]|_σ) then
23                     return e
24                 t ← range(f)
25                 sigs ← sigs ∪ {s}
26                 exps_{t,i} ← exps_{t,i} ∪ {e}
```

expression e' with the same signature has already been considered (lines 3, 4, 19, 20). If so, then it discards e from further consideration. Otherwise, it checks to see if $\mathcal{C}[o \mapsto e]$ evaluates to `true` at all valuations $\sigma \in P$, in which case, the algorithm returns the expression e. If neither of these two cases hold, then the algorithm caches the expression in the appropriate set for subsequent use as a sub-expression in larger expressions (line 10, 26). Note that the notation $\mathsf{typeof}(e)$ is used to denote the type of the expression e, $\mathsf{dom}(f)$ is used to denote the ordered

Algorithm 4.2: SynthForAll: Synthesize an expression that is consistent for *all* inputs

Input : An expression vocabulary $\mathfrak{F}_v$ over a set of types $\mathcal{T}$.
A set of typed variables $V \cup \{o\}$.
A constraint $\mathcal{C}$ over $V \cup \{o\}$.
Output : An expression e such that for every valuation $\sigma \in \mathcal{S}_V$, $\mathcal{C}[o \mapsto e]\big|_\sigma = \texttt{true}$.
Data : An ordered list P of valuations $\sigma \in \mathcal{S}_V$, initially empty.

```
1 while true do
2     e ← SynthForPoints(F_v, V, C, P)
3     if ¬ C[o ↦ e] is unsatisfiable then
4         return e;
5     else
6         σ ← valuation such that ¬ C[o ↦ e]|_σ is true
7         append σ to P
```

list of types which are the domain of the function f and $\mathsf{range}(f)$ is used to denote the range of the function f in Algorithm 4.1.

Algorithm 4.2, SynthForAll, synthesizes an expression e such that $\mathcal{C}[o \mapsto e]$ is valid. It accomplishes this by repeatedly invoking Algorithm 4.1, SynthForPoints, with a monotonically increasing set of points in the list P. The check in line 3 of the algorithm is performed using an SMT solver. We use the SMT solver Z3 [dMB08] in our implementation. Each time that the SMT solver returns a witness for the invalidity of $\mathcal{C}[o \mapsto e]$, we use that witness to augment P, and re-invokes SynthForPoints with the augmented P.

Attempting to first find an expression that is correct for a set of valuations P which were witnesses to failed verification attempts in the past, enables the pruning by means of signatures in Algorithm 4.1. The two techniques together yields two advantages over a naïve enumeration of expressions: (1) The number of expressions enumerated is much smaller. Note that when an expression is discarded, it is also never considered to build larger expressions from as well, and thus results in a decrease in the number of expressions enumerated at the next level. We have empirically observed that this ripple effect can significantly reduce the number of expressions enumerated, and thus allow our techniques to synthesize larger expressions than possible if these optimizations were not applied. (2) The number of expensive calls to an SMT solver are reduced, when compared to a naïve algorithm which invokes the SMT solver on every expression that it considers.

Example 1. *To illustrate the working of Algorithm 4.2, consider the problem of finding an expression for the output variable* o*, which needs to be updated with the value* $\mathsf{max}(a, b)$*, where*

Expression returned by SynthForPoints	Counterexample which violates $\mathcal{C}$	Valuation σ added to P
—	—	$\langle a : 0, b : -1 \rangle$
a	$\langle a : 0, b : 1, o : 0 \rangle$	$\langle a : 0, b : 1 \rangle$
$\mathsf{ite}(\mathsf{iszero}(\mathsf{dec}(b)), b, a)$	$\langle a : 0, b : 2, o : 0 \rangle$	$\langle a : 0, b : 2 \rangle$
$\mathsf{ite}(\mathsf{gt}(b, a), b, a)$	—	—

Table 4.2: Illustration of the working of the expression inference algorithm

$a, b \in V$*, and with the expression vocabulary in Table 4.1. We can specify this with the following constraint* $\mathcal{C}$ *over the variables* a*,* b *and* o*:*

$$(o \geqslant a) \wedge (o \geqslant b) \wedge ((o = a) \vee (o = b))$$

Table 4.2 shows the expressions that were returned by the calls that SynthForAll *made to* SynthForPoints*, as well as the valuation returned by the SMT solver as a result of attempting to verify that this expression is correct, and the valuation* σ *that was a added to the set* P *maintained by Algorithm 4.2. The first row of the table seeds the set of valuations* P*, by making an query to the SMT solver. The subsequent rows indicate the expression that was attempted to be verified, and the valuation at which the expression is incorrect. We observe that the expression corresponding to* $\mathsf{max}(a, b)$ *was discovered after making only four calls to the SMT solver, although Algorithm 4.1,* SynthForPoints *enumerated approximately five hundred expressions in this process.*

4.3.1 Correctness of SynthForPoints

We now provide a proof that the optimizations in Algorithm 4.1 are sound, *i.e.*, they do not result in expressions being spuriously discarded.

Theorem 1. *Given a set of valuations* P *and a constraint* $\mathcal{C}$*, the algorithm* SynthForPoints *always terminates with a smallest expression* e *which is a well-typed composition of functions in* $\mathcal{F}$*, and satisfies* $\mathcal{C}[o \mapsto e]$ *for every valuation* $\sigma \in$ P*, if such an expression* e *exists. If no such* e *exists, then* SynthForPoints *may run forever.*

Proof. To prove this claim, we need to establish: (1) SynthForPoints always returns an expression that is a well-typed composition of function symbols in $\mathcal{F}$, (2) the expression returned by SynthForPoints satisfies $\mathcal{C}[f_u \mapsto e]$ for every valuation $\sigma \in$ P, and (3) that

SynthForPoints always returns an expression e that satisfies the first two criteria, whenever such an expression exists.

It is easy to see that algorithm SynthForPoints only ever enumerates expressions that are well-typed compositions of function symbols in $\mathfrak{U}$, so the proof for (1) is immediate. By construction, the algorithm always returns an expression e such that $\mathcal{C}[f_u \mapsto e] = \texttt{true}$ for every valuation $\sigma \in P$, so the proof of (2) is also immediate.

To prove that SynthForPoints always returns an expression e if one exists, we leverage the correctness of a naïve version of Algorithm 4.1. The naïve version performs no pruning based on signatures, *i.e.*, lines 4, 5, 9, 20, 21 and 25 are deleted from Algorithm 4.1. The rest of the algorithm remains the same. This naïve algorithm enumerates *all* expressions, and thus it definitely satisfies the theorem. Let $\prec$ be the total order in which the naïve algorithm enumerates expressions. We have as a consequence that if $e_1 \prec e_2$ for some expressions e_1 and e_2, then the size of e_1 is less than or equal to the size of e_2. Further, it is also clear that Algorithm 4.1 enumerates expressions in the *same* order, but it might skip some expressions. Let e_s be the first expression in this sequence such that $\mathcal{C}[o \mapsto e_s]$ is true at all points $\sigma \in P$, *i.e.*, for every other e in the sequence such that $\mathcal{C}[o \mapsto e]$ is true at all points $\sigma \in P$, we have that $e_s \prec e$. We now need to prove that Algorithm 4.1 never discards e_s.

We proceed by induction on the shape of e_s. If e_s is a variable or a constant, *i.e.*, e_s is an expression of size one, then the proof is immediate: the algorithm enumerates all of these, and if some expression e', such that $e' \prec e_s$ has the same signature as e_s over P, then e' is a solution as well, which contradicts the assumption that e_s is the *first* solution.

Suppose e_s consists of a function symbol at the top level, and Algorithm 4.1 does not return e_s. There are two possibilities why this might have happened:

- Algorithm 4.1 actually enumerated e_s, but it was found to have the same signature as some other expression e which was enumerated earlier. In this case, $\mathcal{C}[o \mapsto e]$ is true at all points $\sigma \in P$ as well, contradicting the assumption that e_s is the *first* solution.
- Algorithm 4.1 never enumerated e_s. This can happen if some sub-expression e_{sub} of e_s had the same signature as e'_{sub} on P, and $e'_{sub} \prec e_{sub}$. In this case, the expression $e_s[e_{sub} \mapsto e'_{sub}]$, which is e_s with its sub-expression e_{sub} replaced by e'_{sub} would have been enumerated before e_s. Because e_{sub} and e'_{sub} have the same signatures, so do e_s and $e_s[e_{sub} \mapsto e'_{sub}]$, and thus $\mathcal{C}[o \mapsto e_s[e_{sub} \mapsto e'_{sub}]]$ is true at all points $\sigma \in P$ as well, contradicting the assumption that e_s is the *first* solution.

On the other hand if no such e exists, then Algorithm 4.1, being enumerative, can never prove that no such e exists, if the space of expressions is infinite. It may thus run forever on infinite expression spaces, if no solution exists in the space. □

The rest of this section describes how the concolic snippets specified by the programmer are translated into constraints in the form described towards the beginning of this section. As mentioned earlier, we synthesize for each guard and update independently. So we only describe how the constraints $\mathcal{C}$ — which are of the form required by Algorithm 4.2 — for each lvalue to be updated and for each guard to be synthesized are generated. The same process is repeated for every update and for every guard in the protocol.

4.3.2 Constraints for Update Expressions

TRANSIT assumes a parallel assignment model. This in addition to the requirement that each post-condition in each concolic snippet constrain *exactly* one lvalue makes it straight-forward to extract constraints for update expressions independently: For each lvalue, we group together the pre- and post-conditions from a single guard block within a `Transition`. All of these pre- and post-conditions must constrain the updated value of the same lvalue. We replace this lvalue in the post-conditions by a fresh variable o of the appropriate type. Thus, for each pre- and post-condition pair, we simply make an implication of the form $\mathsf{Pre} \Rightarrow \mathsf{Post}$ and let $\mathcal{C}$ be the conjunction of these implications.

4.3.3 Constraints for Guard Expressions

A guard can be viewed as a Boolean-valued expression. The key difference between computing guards and computing update expressions is that for a given control state and input event, guards cannot be computed independently of each other. To ensure that the behavior of the ESM-SK implementations generated by TRANSIT are deterministic, the computed guards for each control location and input event pair are required to be pairwise mutually exclusive. To compute guards on transitions from a given control location, TRANSIT groups the concolic snippets with the same starting state, input event and next state into one guard-action as shown in Figure 4.2. Therefore, given a starting state and input event, each possible next state has a corresponding guard-action associated with it.

Given a set of guard-actions $\mathrm{B}_1, \ldots, \mathrm{B}_n$, the j^{th} guard-action block is a set of concolic snippets with preconditions Pre_{j1}, ..., Pre_{jk_j}. The algorithm for computing guards sequentially

computes the guards for each of the blocks, starting with B_1. Thus, before synthesizing the j^{th} guard, it has the expressions already synthesized for the guards $g_1, \ldots, g_{j-1}$ corresponding to the guard-action blocks $B_1, \ldots, B_{j-1}$ available to it. To compute a guard g_j for the guard-action block B_j, we observe that for the completion to be deterministic, g_j must evaluate to false whenever the guard g_i evaluates to true, for any $i < j$. This property is expressed with the following constraint:

$$\mathcal{C}_1 \equiv \bigwedge_{i<j} (g_i \Rightarrow \neg g_j)$$

Next, g_j must evaluate to true whenever any of the preconditions $\mathsf{Pre}_{jl}, l \in [1, k_j]$ evaluate to true. This is necessary to ensure that the guard is not too *narrow*. This property can be expressed with the following constraint:

$$\mathcal{C}_2 \equiv \left(\bigvee_{l=1}^{k_j} \mathsf{Pre}_{jl} \right) \Rightarrow g_j$$

Also, corresponding to each block B_i for which a guard has not yet been synthesized (*i.e.*, $i > j$), g_j must evaluate to false whenever any of preconditions in B_i evaluate to true. This is necessary to ensure that the guard is not too overly *broad*. This property is expressed with the following constraint:

$$\mathcal{C}_3 \equiv \bigwedge_{i>j} \left(\left(\bigvee_{l=1}^{k_i} \mathsf{Pre}_{il} \right) \Rightarrow \neg g_j \right)$$

Finally the constraint $\mathcal{C}$ required for inferring g_j is simply the conjunction of the above three constraints, *i.e.*, $\mathcal{C} \triangleq \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_3$.

4.3.4 Evaluation of the Expression Inference Algorithms

To evaluate the performance of the expression inference algorithm, we focus on the *size* of the expressions which the algorithm is able to compute successfully as a key metric. To benchmark the impact of pruning based on signatures in the algorithm SynthForPoints, a large number of random expressions of varying sizes were generated. For each expression, a set of ten concrete valuations for the input variables was generated. For each randomly generated expression, the Algorithm SynthForPoints was used to compute an expression that is consistent with the corresponding set of valuations for the input variables. Figure 4.4 shows that the "Pruned" variant — which prunes the search space using signatures, as described earlier — often explores

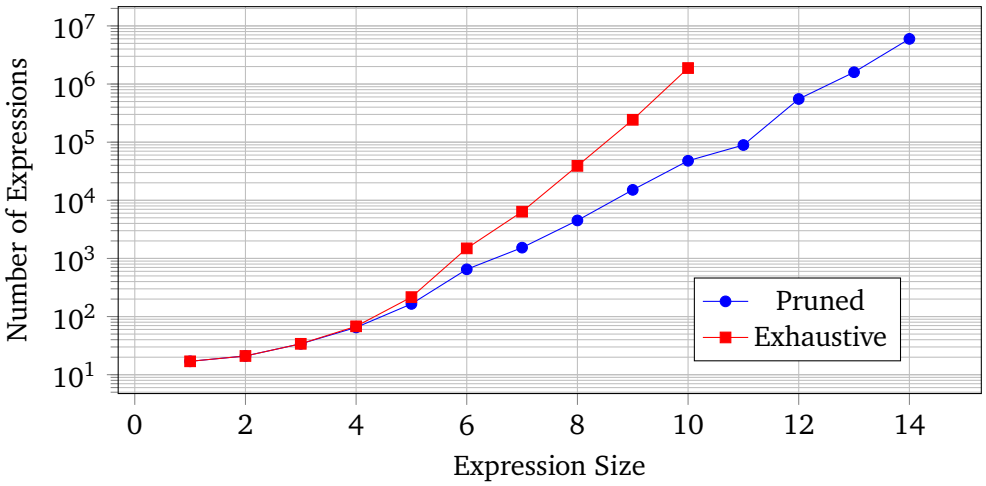


Figure 4.4: Average number of expressions explored for various expression sizes by the Pruned and Exhaustive variants of Algorithm SynthForPoints. We omit data for the Exhaustive variant for sizes greater than 10 where it exceeds the memory limit of 3.5 GB.

two to three orders of magnitude fewer expressions than the "Exhaustive" variant — which does not perform any pruning — for expression sizes larger than ten (note the logarithmic scale on the Y-axis in Figure 4.4).

To evaluate the overall expression inference algorithm, *i.e.*, the performance of SynthForAll in conjunction with SynthForPoints on symbolic constraints, we used the benchmarks shown in Table 4.3. The algorithm computes expressions of up to size 15 within a reasonable amount of time as shown in Table 4.3. The algorithm exceeds our 30 minute time-out on only one benchmark, whose solution has an expression size greater than 20. The right-most column in Table 4.3 shows that the algorithm reaches the desired solution within a few iterations of the CEGIS outer loop.

4.4 Experimental Evaluation of TRANSIT

We first validated the feasibility of using our approach by transcribing two simple, fully specified protocols from the GEMS simulation toolkit [MSB+05] into concolic TRANSIT snippets. These protocols are the Valid-Invalid (VI) protocol and a blocking version of the MSI protocol, which allows for limited concurrency. With four cache processes and one directory, the entire synthesis process took less than a second for each protocol. The key results are summarized in Table 4.4. We then evaluated the approach on three larger case studies: A non-blocking version of the MSI protocol, the MESI protocol and the industrial SGI-Origin protocol.

#	Description	Expected Expression	Expr. Size	Constraint $\mathcal{C}$	Time (s)	# Iters
1	Max. of a, b	$\mathsf{ite}(\mathsf{gt}(a, b), a, b)$	6	**(a)** $((a > b) \Rightarrow (o = a)) \wedge ((b > a) \Rightarrow (o = b))$	< 1	1
				(b) $o \geqslant a \wedge o \geqslant b \wedge (o = a \vee o = b)$	< 1	2
2	Max. of a, b, c	Similar to 1	15	Similar to 1**(a)**	536	7
				Similar to 1**(b)**	762	16
3	Sym. Diff. of s_1, s_2	$\mathsf{setunion}(\mathsf{setminus}(s_1, s_2), \mathsf{setminus}(s_2, s_1))$	7	$o \subseteq (s_1 \cup s_2) \wedge o \cap (s_1 \cap s_2) = \{\} \wedge o \cup (s_1 \cup s_2) = s_1 \cup s_2$	< 1	2
4	Sym. Diff. of 3 sets	Similar to 3	11	Similar to 3	< 1	6
5	Sym. Diff. of 4 sets	Similar to 3	15	Similar to 3	132	14
6	Conditional Update	$\mathsf{ite}(\mathsf{equals}(e, c_1), a, b)$	6	$((e = c_1) \Rightarrow (o = a)) \wedge ((e \neq c_1) \Rightarrow (o = b))$	< 1	4
7	Largest of 2 sets	$\mathsf{ite}(\mathsf{gt}(\mathsf{setsize}(s_1), \mathsf{setsize}(s_2)), s_1, s_2)$	8	**(a)** $(\lvert s_1\rvert > \lvert s_2\rvert \Rightarrow o = s_1) \wedge (\lvert s_2\rvert > \lvert s_2\rvert \Rightarrow o = s_2)$	< 1	1
				(b) $\lvert o\rvert \geqslant \lvert s_1\rvert \wedge \lvert o\rvert \geqslant \lvert s_2\rvert \wedge (o = s_1 \vee o = s_2)$	< 1	2
8	Largest of 3 sets	Similar to 7	> 20	Similar to 7**(b)**	TO	–

Table 4.3: Benchmarks and evaluation of the expression inference algorithms

Protocol	# Snippets	**Synthesis**						**State-Space**
		Updates			Guards			
		Num. synth.	Exps tried	Time (secs)	Num. synth.	Exps tried	Time (secs)	
VI	19	49	449	< 1	17	525	< 1	140K
MSI	77	157	3330	< 1	45	3710	< 1	854K

Table 4.4: Performance of snippet-based design. The column labeled "Num. synth" represents the number of guard and update expressions that needed to be synthesized. The "Exps tried" column shows the number of expressions enumerated by the expression inference algorithm, and the column labeled "State-Space" shows the number of states in the final protocol.

4.4.1 Case Study A: Non-blocking MSI

We specified the non-blocking MSI protocol described in the synthesis lectures [SHW11] using concolic snippets in TRANSIT. Note that the MSI protocol referred to in Table 4.4 is a *blocking* version of the MSI protocol. The non-blocking version of the MSI protocol considered in this case study allows a greater number of concurrent requests to be in flight, requiring the

programmer to consider a larger number of corner cases due to the increased concurrency resulting from the larger number of in-flight requests.

The scenarios described in the text resulted in a sparse initial set of snippets, as most of the tricky corner cases were either indirectly specified in the textual description or were left unspecified. Hence, the programmer added 67 more snippets over 13 debugging iterations before converging to a correct protocol. In each such iteration, the programmer either added symbolic snippets, when the behavior of the protocol in some corner case was completely unspecified, or concrete snippets, when a specification existed but was incomplete. Table 4.5 summarizes the effort and complexity in this experiment.

4.4.2 Case Study B: From MSI to MESI

The goal of our second case study was to augment the blocking MSI protocol with an "Exclusive" or the E state to arrive at the MESI protocol. The E state is an optimization that grants read-write permissions to the first reader of an unshared address (*i.e.*, not present in any cache) — as opposed to just read permission in MSI — thereby eliminating coherence traffic on a subsequent write to the same address. The synthesis lectures [SHW11] describe this protocol in terms of new scenarios and modifications to scenarios in the MSI protocol. Our approach was to add the corresponding snippets to the existing set of snippets used to specify the blocking version of the MSI protocol. Because the examples describe a MESI protocol with a non-blocking directory, we modified our baseline MSI protocol correspondingly.

The extended protocol contained five new states (four for the cache, one for the directory), and seven new message types. In the first iteration, we added 19 snippets to specify transitions involving the E state and the non-blocking behavior of the directory. These snippets described the behavior of the protocol in under-specified corner cases and scenarios involving transient states and were added in response to the errors reported by the model checker. The programmer was able to obtain a fully verified protocol by adding twelve additional snippets over eight iterations. Additional metrics gathered during this case study are presented in Table 4.5.

4.4.3 Case Study C: The SGI-Origin Protocol

For our final case study, we chose the coherence protocol used in the SGI-Origin 2000 servers [LL97], which is highly cited in the cache coherence literature. The Origin protocol is a directory-based, MESI protocol, and it supports multiple concurrent requests to the

	Case Study A	Case Study B
Snippets in the first/last version	19/86	96/108
Writing first set of snippets	2 hrs	6 hrs
Total manual effort	6 hrs	13 hrs
Number of iterations	13	8
Number of traces inspected	5	6
Number of updates/guards inferred	175/80	260/74
States in verified protocol	1.48M	1.5M

Table 4.5: Effectiveness Metrics for Snippet-based Protocol Design

same address. Processes communicate through messages that may be arbitrarily re-ordered in the network. The consequent race conditions made it an interesting candidate for this case study.

Laudon and Lenoski [LL97] describe the common case protocol behavior using request flows. In this experiment, ignoring the "poisoned" directory state (used for page-migration), we transcribed each of the read, read exclusive, upgrade, and write-back flows using symbolic snippets in TRANSIT. Except for the obvious cases, which corresponded to the well-understood parts of the protocol, we left most of the guards empty and specified all conditional attributes on message fields and process variables with pre-conditions.

The protocol skeleton comprised of the cache process and directory processes, four request types, twelve response types, the request and response networks, and an intervention network used to buffer intervention requests. We initially specified 56 transitions in the cache machine and 18 transitions in the directory machine. We also specified the guards in instances where the incoming message type was found to be inconsequential; doing so prevented the tool from exploring artificially large expressions involving the disjunction of these enumerated types. The resulting protocol resulted in an error discovered by the model checker due to the cache process receiving an unexpected message. We fixed this case by adding a concrete snippet describing the desired behavior of the cache. Once again we left the guards unspecified, but the pre-conditions and update constraints were predicated by identical values for the input message fields and internal process variables, as seen in the violating trace.

Continuing in a similar manner, we added concrete snippets to fix error traces as we encountered them. In some cases, the tool identified inconsistencies between the added trace and a pre-existing constraint. We found it straightforward to reconcile these differences before

converging to a protocol that model checked. The final synthesis step took a little over 30 minutes, exploring over four million states during model checking. The generated TRANSIT specification had a total of 50 `Transitions`.

4.4.4 Discussion and Limitations

We found the primary convenience of using TRANSIT to be the manner in which the initial specification phase and the iterative debugging phases could be expressed differently. Although it was natural to transcribe the bulk of the protocol symbolically from the algorithmic description of flows, this description was invariably incomplete. Several corner cases, for which the behavior was not explicitly specified were discovered during model checking. Most of these errors occurred due to unintended interactions between flows. The unexpected message condition cited above resulted from a cache process that was participating in a read-write-back race scenario. TRANSIT generalized the concrete fixes provided by the programmer in a manner that was guaranteed not to contradict the constituent flows. Fixing this bug symbolically would have required reasoning about the impact on both these flows. Similarly, another coherence violation was the result of the sharer set in the directory being updated incorrectly when a previous owner was downgraded. Again, the fix involved adding a snippet that concretely specified the next contents of the sharer set with the pre-condition specifying only the erroneous case.

One limitation of the TRANSIT approach is that the "shape" of the protocol is assumed to be provided and complete. For instance, if a particular transition is not provided by the programmer, then it results in a deadlock or liveness violation during the model checking run. To fix the problem, the programmer needs to add *additional* `Transition` blocks or additional guard-action blocks within a `Transition` block. This is a case where a purely concrete fix is not sufficient, because the programmer has to specify the missing behavior, perhaps using concolic snippets. If this part of the behavior is missing from the textual description, then the programmer might indeed have to resort to reasoning about the entire protocol to deduce the correct behavior. In summary, the TRANSIT approach does not infer missing transitions. Further, as we have already mentioned, liveness requirements are not supported by TRANSIT, as a consequence of the choice of model checking framework (Murφ) that TRANSIT is integrated with. We seek to address both of these limitations in the work described in subsequent sections.

5

SyGuS

After building TRANSIT, we realized that the synthesis problem which we solved with a domain specific synthesizer in TRANSIT shared a lot of similarities with other synthesis problems addressed in literature [JGST10, Gul11, GJTV11, BCG+13, SSA13]. Although the synthesis problems solved by various approaches are similar in spirit, the disparate input formats accepted by each tool, and the disparate assumptions made by each tool about the search space of programs made it impossible to compare the relative merits of various synthesis techniques. A similar problem experienced by the constraint solving community led to the creation of the SMT-LIB standards [BST10a, BST10b]. The motivation behind the SMT-LIB standards was to specify a common input language (the SMT-LIB language) and a set of background *theories*, so that SMT solvers which accepted the SMT-LIB language could be compared in a uniform manner, with the same inputs. Inspired by the SMT-LIB approach, we formulated an input format to specify synthesis problems. This effort resulted in the creation of the Syntax-guided Synthesis (SyGuS) language [RU14], and a competition [AFSSL14] along the lines of the annual Satisfiability Modulo Theories Competition (SMT-COMP) [Org05].

To encourage adoption, we attempted to keep the SyGuS language as close to the SMT-LIB language as possible. The SyGuS language extends the SMT-LIB language with constructs for specifying synthesis problems, and inherits the set of background theories from SMT-LIB. We briefly describe the components of a SyGuS problem here, by way of examples, and refer the reader to the language reference [RU14] for details on the specific syntax of SyGuS.

At a high level, the functional synthesis problem consists of finding a function f such that some logical formula φ, which captures the correctness of f is valid. In syntax-guided synthesis, the synthesis problem is constrained in three ways: (1) the logical symbols and their

interpretations are restricted to a *background theory*, (2) the *specification* φ is limited to a first order formula in the background theory with all its variables universally quantified, and (3) the universe of possible functions f is restricted to syntactic expressions described by a *grammar*. We now elaborate on each of these points, and conclude this chapter by comparing and contrasting SyGuS with other meta-synthesis frameworks proposed in literature.

5.1 Correctness Specification

For the function f to be synthesized, we are given the type (or sort, if one wishes to use SMT parlance) of f and a formula of the form $\exists f \, \forall \mathbf{x} \, \varphi[f, \mathbf{x}]$ as its correctness specification. The formula $\varphi[f, \mathbf{x}]$ is a quantifier-free Boolean combination of predicates from the background theory, symbols from the background theory, and the function symbol f, all used in a type-consistent manner.

Example 2. *Assuming the background theory is LIA (Linear Integer Arithmetic), consider the specification for a function f of type $int \times int \mapsto int$:*

$$\psi_1 \equiv \exists f \, \forall x, y \; \varphi_1[f, x, y], \text{ where,}$$
$$\varphi_1[f, x, y] \equiv f(x, y) = f(y, x) \wedge f(x, y) \geqslant x.$$

Note that all the variables in the formula ψ_1 are bound *to the universal quantifier, and all the unknown functions (in this case, just f) are existentially quantified. A given function f' satisfies the above specification if the quantified formula $\forall x, y \; \varphi_1[f \mapsto f']$ holds, or equivalently, if the formula $\varphi_1[f \mapsto f']$ is valid. The notation $\varphi[f \mapsto f']$ indicates that all occurrences of f in $\varphi[f, \mathbf{x}]$ are replaced by f'.*

5.2 Set of Candidate Expressions

To make the synthesis problem tractable, as well as to allow users to encode any domain-specific knowledge about the search space of programs, the "syntax-guided" version allows the user to impose structural (syntactic) constraints on the set of possible functions f. The structural constraints are imposed by restricting f to the set L of functions defined by a given context-free grammar G_L. Each expression in L has the same type as that of the function f, and uses the symbols in the background theory T, composed according to the rules of the grammar G_L, and the variables corresponding to the formal parameters of f.

Example 3. *Suppose the background theory is LIA, and the type of the function* f *is* $int \times int \mapsto int$. *We can restrict the set of expressions* $f(x, y)$ *to be linear expressions of the inputs by restricting the body of the function to expressions in the set* L_1 *described by the grammar below:*

$$linterm \;:=\; x \mid y \mid intconst \mid linterm + linterm$$

Alternatively, we can restrict $f(x, y)$ *to conditional expressions with no addition by restricting the body terms from the set* L_2 *described by:*

$$\begin{aligned} term \;&:=\; x \mid y \mid intconst \mid ITE(cond, term, term) \\ cond \;&:=\; term \leqslant term \mid cond \wedge cond \mid \neg cond \mid (cond) \end{aligned}$$

Grammars can be conveniently used to express a wide range of constraints, and in particular, to bound the depth and/or the size of the desired expression.

5.3 The Problem Definition

Informally, given the correctness specification ψ of the form $\psi \triangleq \exists f \forall \mathbf{x}\, \varphi[f, \mathbf{x}]$, with $\varphi[f, \mathbf{x}]$ as its quantifier free part, and the set L of candidates, we want to find an expression $e \in L$ such that if we use e as an implementation of the function f, the formula φ is valid. Let us denote the result of replacing each occurrence of the function symbol f in φ with the expression e by $\varphi[f \mapsto e]$. Note that we need to take care of binding of input values during such a substitution: if f has two arguments and expressions in L refer to the formal parameter names of f as x and y, then every occurrence of the form $f(e_1, e_2)$ in the formula φ must be replaced with the corresponding expression $e[x \mapsto e_1, y \mapsto e_2]$ obtained by replacing x and y in e by the expressions e_1 and e_2, respectively. We can now define the SyGuS problem:

> Given (1) a background theory T, (2) a typed function symbol f, (3) a quantifier-free formula $\varphi[f, \mathbf{x}]$ over the vocabulary of T along with f, and the set of variables $\mathbf{x}$, and (4) a set L of expressions over the vocabulary of T, the formal parameters of f, and of the same type as f, find an expression $e \in L$ such that the formula $\varphi[f \mapsto e]$ is valid modulo T.

Example 4. *For the specification* φ_1 *presented earlier, if the set of allowed implementations is* L_1 *as shown before, there is no solution to the synthesis problem. On the other hand, if the set*

of allowed implementations is L_2*, a possible solution is the conditional if-then-else expression* $ITE(x \geqslant y, x, y)$.

In some special cases, it is possible to reduce the decision problem for syntax guided synthesis to the problem of deciding formulas in the background theory using additional quantification. For example, every expression in the set L_1 is equivalent to $ax + by + c$, for integer constants a, b, c. If φ is the correctness specification, then deciding whether there exists an implementation for f in the set L_1 corresponds to checking whether the formula $\exists\, a, b, c\ \forall \mathbf{x}\ \varphi[f \mapsto ax + by + c]$ is true, where $\mathbf{x}$ is the set of all free variables in φ. This reduction was possible for L_1 because the set of all expressions in L_1 can be represented by a single parameterized expression in the original theory. However, the grammar may permit expressions of arbitrary depth which may not be representable in this way, as in the case of L_2.

5.4 Comparison with other Meta-synthesis Frameworks

Broadly speaking, SyGuS can be thought of as a *meta-synthesis* framework: it essentially allows a concise description of *any* synthesis problem whose solution space can be described using a context-free grammar and function symbols in some combination of background SMT theories, and whose properties can be described using a universally quantified formula. We now compare and contrast SyGuS with some other frameworks that have been proposed in recent literature, which have similar goals.

5.4.1 SKETCH and Rosette

SKETCH [SLRBE05, STB+06, SAT+07, SLJB08, Sol09] and Rosette [TB13, TB14] are both meta-synthesis frameworks that were designed to be *embedded* within a language: the SKETCH language is C-like, whereas Rosette is embedded within the functional language, Racket. As with SyGuS, the space of programs is described using a context-free grammar in Rosette, and using *generators* — which use a combination of regular and context-free constructs to describe the search space — in SKETCH.

Unlike SyGuS, these frameworks allow the specification for the program to be synthesized to be written as a program. SKETCH uses a subset of the C programming language to describe the behavior of the program to be synthesized. This C program could possibly be sub-optimal or unoptimized, with the sketch for the program describing the *shape* of an optimized version. Rosette, on the other hand, specifies the properties of the program to be synthesized using a

	SyGuS	SKETCH	Rosette	FlashMeta
Specification language	SMTLIB-like	C-like	Racket	Inductive spec.
Program space	CFG	Generators	CFG	CFG
Full formal specifications	Yes	Yes	Yes	No
Inductive specifications	Yes	Yes	Yes	Yes
Solvers extensively use ranking?	No	No	No	Yes
Language and Platform agnostic?	Yes	No	No	Relatively
Intended audience	Synthesis Researchers	Programmers	Programmers, Students	Domain experts
Existence of multiple solvers	Yes	No	No	No

Table 5.1: Comparison of various meta-synthesis frameworks

combination of assertions, pre-conditions and post-conditions on the program. Needless to say, these specification techniques can be much more expressive than the first order specifications that SyGuS allows. As a consequence, these techniques sometimes require inputs from the programmer — in the form of *pragmas* in the case of SKETCH — or restrict the language to a *safe*, and decidable subset — as is the case with Rosette.

The differences between SKETCH and Rosette on the one hand and SyGuS on the other stem from the design choices made with the intended audience in mind. SKETCH and Rosette are both intended to enable programmers synthesize usable code, whereas SyGuS intends to cleanly abstract the core synthesis problem in a language and platform agnostic manner, to encourage adoption and spur research in program synthesis techniques. Indeed, the relatively low entry barrier has led to a multitude of solvers competing in the 2015 SyGuS competition.

Lastly, we note that regardless of the exact logic used to specify properties of the program to be synthesized, SyGuS, SKETCH and Rosette all support full and formal specifications, *i.e.*, it is possible for specifications to unambiguously and formally describe the behavior of the program to be synthesized for *any* input.

5.4.2 FlashMeta

FlashMeta [PG15] is another meta-synthesis framework which is geared towards synthesis from *inductive* specifications [PG15]. An inductive specification is a quantifier free first-order predicate, where each atom constrains the behavior of the desired program on a *specific* concrete input. Various other techniques for program synthesis using inductive specifications [Gul11, SG12, LG14, BGHZ15, KG15] can be expressed using the FlashMeta framework [PG15].

Like SyGuS, FlashMeta uses a context-free grammar to describe the space of candidate programs. However, unlike SyGuS, FlashMeta does not assume the existence of background SMTLIB theories, and thus does not restrict the space of programs to consist only of function symbols from some background theory. FlashMeta allows *any* function that can be expressed as a pure C# function to be used in the context-free grammar that describes the search space for candidate programs. For programs that operate on infinite domains, such as the domain of strings and integers, inductive specifications can be viewed as an under-approximation of a complete specification. It is possible that two behaviorally different programs both satisfy a given inductive specification. FlashMeta uses domain specific *ranking* schemes to determine which program is most likely to be the program desired by the user from among a set of programs which all satisfy the inductive specification [PG15, SG15]. Ranking is especially important when inductive specifications are used, as there always exists a trivial solution which is a large case split over all the concrete inputs referred to in the inductive specification. Such a solution is undesirable, because it does not generalize well to unseen inputs.

A novel feature of FlashMeta, that is not present in any of the other meta-synthesis frameworks discussed in this dissertation, is the use of *witness functions* [PG15]. A witness function is specified by a programmer, who, in this case is assumed to be an expert, with a deep knowledge of the kinds of programs that are likely to be useful for an end user. Consider an inductive specification φ, for a function f which is to be synthesized. Further, suppose that the synthesizer is exploring the possibility that the top-level operator for f is F. The *shape* of the program is thus hypothesized to be $F(a_1, a_2, \ldots a_n)$, where the arguments a_i now need to be synthesized. A witness function $\omega_j(\varphi)$ *deduces* a specification φ_j on the j^{th} argument to F. This essentially allows FlashMeta to *decompose* the synthesis problems into multiple sub-goals, which in turn leads to scalable synthesis algorithms.

We conclude the comparison with other meta-synthesis frameworks by noting that Table 5.1 compares and contrasts the various meta-synthesis frameworks along different dimensions and summarizes the comparison that we have just presented.

6

Enumerative Strategies for SyGuS Solvers

This chapter describes how enumerative strategies can be used to solve instances of the SyGuS problem. The first strategy we describe is a straightforward extension of the algorithm used to infer expressions in TRANSIT, presented in Section 4.3. We then discuss recent advances made in the area of SyGuS solvers, and present an algorithm for a class of SyGuS instances variously termed *single invocation* [RDK+15], *separable* [ACR15], or *single-point definable* [MNS16] in recent literature. The algorithm is enumerative in spirit, but uses a divide-and-conquer approach by synthesizing multiple expressions, each of which is correct for a subset of inputs, and then attempts to *unify* [ACR15] these expressions using conditionals.

6.1 ESOLVER: An Enumerative SyGuS Solver

Having defined the SyGuS problem, as well as the language to describe instances of the SyGuS problem, we built a solver for such instances based on enumerating candidate expressions, which we dub ESOLVER. The core algorithms used in ESOLVER are similar to the algorithms for inferring expressions in TRANSIT, described in Algorithms 4.1 and 4.2. We use the notion of a signature to prune the space of expressions to be searched. The key differences from the algorithms presented in Algorithms 4.1 and 4.2 are that:

- ESOLVER does not assume that all well-typed expressions are a part of the candidate space, and instead enumerates expressions using the grammar provided as part of the problem instance.
- The notion of a signature, which we use to prune the search space, now needs to take into account the *non-terminal* in the grammar from which an expression was derived, to avoid spurious pruning.

- ESOLVER handles several extensions to the SyGuS solver — such as the let construct in constraints and grammars [RU14], which we have not described here.

We do not present the details about the implementation of ESOLVER, as it is a rather straightforward extension of the algorithms presented in Section 4.3. ESOLVER won the 2014 SyGuS competition with four other solvers participating. The implementation of ESOLVER— along with two other implementations, one based on symbolic search [GJTV11, JGST10] and the other based on a stochastic search [SSA13] — has been made available as a baseline for other participants to compare against, and possibly build upon, and is continually maintained [JRU13].

The 2015 SyGuS competition had several new solvers competing, the most notable of general-purpose solver being the CVC4 solver [RDK+15]. The CVC4 solver was the overall winner of the 2015 SyGuS competition, with ESOLVER coming in second place overall. However, despite CVC4 being the overall winner, there were a set of benchmarks which could not be solved by the CVC4 solver, but which ESOLVER could solve, as well as the other way around. In addition, a solver based on a unification approach was also proposed by Radhakrishna et. al. [ACR15], which did not participate in the 2015 SyGuS competition, but has an impressive performance nonetheless. The next section provides a brief overview of these new algorithms to solve the SyGuS problem, and discusses the capabilities and limitations of ESOLVER (and enumerative strategies in general) with respect to the newer algorithms.

6.2 Capabilities and Limitations of ESOLVER

These advances in SyGuS solvers led us to look more closely at the capabilities and limitations of enumerative solution strategies. We observed that the newer solvers performed extremely well with a class of specifications that have been termed variously as *single-invocation* specifications [RDK+15], or *separable* specifications [ACR15]. We note that the specifications in a large fraction of the SyGuS benchmark suite fall into this class. We also observed that both the newer solvers made extensive use of the specification itself in the actual synthesis algorithms; whereas ESOLVER makes very minimal use of the specifications in driving the search.

6.2.1 Separable Specifications

We treat the notion of separability as a semantic notion in this dissertation. We shall only consider SyGuS specifications which refer to *only one* unknown function to be synthesized in

the rest of this chapter. The definitions can be extended to specifications which involve multiple functions, but will not be very useful in the context of this dissertation. Also, we shall assume that the background theory T, over which the SyGuS problem is defined, is decidable.

Intuitively, a specification, which describes the constraints on an unknown function f, is *separable*, if and only if it admits a solution where, for any concrete input $\mathbf{c}_1$, the value of $f(\mathbf{c}_1)$ is *independent* of the value of $f(\mathbf{c}_2)$, where $\mathbf{c}_2 \neq \mathbf{c}_1$ is any other concrete input. This definition corresponds very closely with the definition of a *single-point definable specification*, presented in a concurrent work [MNS16].

There has been a lot of interest recently in separable specifications because the synthesis problem for such specifications can be reduced to determining the truth of a first-order sentence. This problem is decidable, provided that the background theory T is decidable.[9] We will explain this reduction in greater detail later in this chapter. Apart from this advantage, separable specifications, by definition, allow for synthesis strategies that produce solution fragments (or sub-expressions) which are correct on some subset of inputs. These sub-expressions may then be combined using an if-then-else operator, or other techniques. We explore one such algorithm in this chapter.

Although we have informally defined the semantic notion of separability, checking if a SyGuS specification is separable using this semantic notion is challenging, and is an open problem. Most recent approaches [ACR15, RDK$^+$15, MNS16] instead check if a specification satisfies some syntactic restrictions which are sufficient to prove separability [ACR15, RDK$^+$15], or check that the specification satisfies a stronger property, such as *single-point refutability* [MNS16], which is easier to check for. In this dissertation, we adopt a syntactic check for separability, which is performed after some amount of rewriting of the original specification. We now provide a few examples of SyGuS specifications which are separable and otherwise, to give the reader an intuitive feel for the notion of separability.

Example 5. *Consider the following specification, which describes a binary function* f *which computes the maximum of its arguments:*

$$\psi_{\text{sep1}} \equiv \exists f \, \forall x, y \; (f(x,y) \geqslant x \wedge f(x,y) \geqslant y \; \wedge (f(x,y) = x \vee f(x,y) = y)) \tag{6.1}$$

ψ_{sep1} *is separable, because all applications of* f *have the same arguments, and therefore never*

[9] Ignoring any syntactic restrictions on the solution.

correlates the values that f *can evaluate to, for different inputs. Further, there exists a solution* $f(x, y) \equiv \max(x, y)$*, whose output, for any given input, never depends on its output for some other input.*

This example seems to indicate that a purely syntactic definition suffices: A specification is separable if and only if all occurrences of f, which is the function to be synthesized for, in the specification involve applications of f to the same set of arguments. However, the next two examples show that this is not the case.

Example 6. *The following specifications are separable even though* f *is applied to different arguments in each specification:*

$$\psi_{\text{sep2}} \equiv \exists f\ (f(1) = 1 \wedge f(2) = 2)$$
$$\psi_{\text{sep3}} \equiv \exists f\ \forall x, y\ ((x = 1 \Rightarrow f(x) = 1) \wedge (y = 2 \Rightarrow f(y) = 2))$$
$$\psi_{\text{sep4}} \equiv \exists f\ \forall x, y\ (x = y \Rightarrow f(x) = f(y))$$

The specifications ψ_{sep2} *and* ψ_{sep3} *are separable, because each clause in each of these specifications constrains the value of* f *at exactly one point. Any solution* h*, such that* $h(1) = 1$ *and* $h(2) = 2$ *is a valid solution. The specification* ψ_{sep3} *is semantically equivalent to* ψ_{sep2}*. The specification* ψ_{sep4} *is in fact a tautology — recall that* f *is a* function*, and cannot evaluate to different results when applied to the same arguments — and therefore separable. Any function can be used as a solution for* ψ_{sep4}*.*

Thus, if all function applications are over the same arguments, then the specification is definitely separable, but this is not a necessary condition.

Example 7. *The following specifications, which state that* f *is a monotonic function, are not separable, because they correlate the value of* f *applied to different arguments:*

$$\psi_{\text{nonsep1}} \equiv \exists f\ \forall x, y\ (x \leqslant y \Rightarrow f(x) \leqslant f(y))$$
$$\psi_{\text{nonsep2}} \equiv \exists f\ \forall x\ (f(x) \leqslant f(x + 1))$$

To be a solution to ψ_{nonsep1} *and* ψ_{nonsep2}*, a function* h *needs to be such that* $h(x) \leqslant h(y)$ *for all* $x \leqslant y$*. Clearly, the output of any candidate solution* h *on a concrete input* $\mathbf{c}_1$ *cannot be chosen independently of all other concrete inputs* $\mathbf{c}$*, if monotonicity is to be maintained.*

The following example demonstrates the subtleties of the definition of separability, and also that a purely syntactic definition of separability is likely to be insufficient.

Example 8. *The following specification for the constant function* f*, which takes an integer as input and returns an integer is separable.*

$$\psi_{\text{sep5}} \equiv \exists\, f\, \forall x\, (f(0) = 0 \wedge f(x+1) = f(x))$$

Although the specification ψ_{sep5} *correlates the output of* f *applied to distinct arguments, it is equivalent to the specification* $\exists\, f\, \forall x\, f(x) = 0$*, which is obviously separable.*

As Example 8 demonstrates, the semantic notion of separability, which could involve arbitrary equivalences between formulas, is difficult to check for. Consequently, we define the notion of *plain separability*, which is a syntactic notion that is easier to check for.

Plainly Separable Specifications

Consider a SyGuS specification ψ, over some background theory T. The specification ψ can refer to functions defined in the theory T, the unknown function f, of arity n, as well as to variables in the set $\mathbf{x} = \{x_1, x_2, \ldots, x_m\}$. Further, ψ has the form $\psi \triangleq \exists\, f\, \forall\, x_1, x_2, \ldots x_m\ \varphi[f, \mathbf{x}]$. Where $\varphi[f, \mathbf{x}]$ is a quantifier-free formula over symbols in the background theory T, the unknown function f as well as the variables in $\mathbf{x}$.

We denote by φ_{cnf}, a formula which is *equivalent* to φ, and in conjunctive normal form (CNF). A formula is said to be in CNF if it has the form $c_1 \wedge c_2 \wedge \cdots \wedge c_k$, where each c_i, for $i \in [1, k]$ — called a *clause* — has the form $a_{i1} \vee a_{i2} \vee \cdots \vee a_{im_i}$, where each $a_{ij}, i \in [1, k], j \in [1, m_i]$ is an atom, and does not involve conjunctions or disjunctions, but could possibly appear negated. Thus, all negations are restricted to be applied only to atoms. Note that we require φ_{cnf} to be *equivalent* to φ and not just equi-satisfiable with respect to φ. For simplicity of presentation, we assume that the straightforward, exponential transformation to CNF is used to derive φ_{cnf} from φ. This is not a problem in practice, because φ is typically not large. If desired, techniques like Tseitin's transform [Tse83] can also be used, provided appropriate care is exercised while checking validity: checking that the negation of a equi-satisfiable formula produced by Tseitin's transform, which contains auxiliary variables introduced by the transform is unsatisfiable, may no longer imply that the original formula is logically valid. Having set up the necessary definitions and the form of the specification ψ, we can now define plain separability.

Definition 1. *The* SyGuS *specification* ψ*, of the form described above, with* φ *as its quantifier free part, is called* plainly separable *if and only if for each clause* c *in* φ_{cnf}*, we have that* c *is either a tautology, or* every *occurrence of* f *in* c *has* f *applied to the* same *arguments.*

The notion of a *single-point refutable* specification, which has been proposed in concurrent work [MNS16] is a more sophisticated definition for the concept of plain separability. But it requires that the domains and ranges of all functions, including the ones defined by the background theory T be extended by a distinguished undefined value. In principle, there exist specifications that are not plainly separable by our definition, but are still single-point refutable. Such specifications can indeed be solved for by the algorithm which we shall propose later in this chapter, but would be rejected based on our definition of plain separability. Fortunately however, all of the benchmarks in the classes that we have targeted in the SyGuS benchmark suite have plainly separable specifications.

Both the unification based solver, and the CVC4 SyGuS solver exploit the (plain) separability of specifications, when applicable, to apply an algorithm that leverages such specifications. As mentioned earlier, a large fraction of the SyGuS benchmark suite consists of separable specifications, so a better algorithm for such specifications has immediate practical value. We shall focus only on separable specifications in the rest of this chapter.

6.2.2 Black Box and White Box Algorithms

All the three baseline SyGuS solvers can be broadly classified as being *black box* algorithms, and can all be viewed as instantiations of the counterexample guided inductive synthesis (CEGIS) paradigm [SLRBE05]. These solvers use the specification φ only to verify that a proposed solution is correct, and possibly to obtain concrete values of the universally quantified variables on which the proposed solution fails. These concrete values could then be possibly used by the black box solvers to rule out the current solution from future solution proposals. The specification is not directly used to guide the search in any way. The CVC4 and unification based algorithms, on the other hand, can be considered *white box* algorithms. These algorithms make extensive use of the specification to derive a solution, and perform very minimal, if any, enumeration; instead preferring to use theory-specific synthesis algorithms. We briefly describe both of these strategies, and compare and contrast their strengths and limitations with respect to enumerative approaches. To describe the two algorithms, we consider a plainly separable SyGuS specification ψ, over some background theory T, of the form $\psi \triangleq \exists\, \mathsf{f}\, \forall\, \mathbf{x}\ \varphi[\mathsf{f}, \mathbf{x}]$, which

refers to the single unknown function f, symbols of T, and the set of universally quantified variables $\mathbf{x}$. As usual, $\varphi[f, \mathbf{x}]$ is a quantifier-free formula over symbols of T, f and variables in $\mathbf{x}$.

The CVC4 SyGuS Solver

The description of the CVC4 SyGuS solver presented here is a highly condensed version of the presentation from the original paper describing the algorithm [RDK+15]. Let us denote by $\mathbf{x} \triangleq \{x_1, x_2, \ldots, x_n\}$, the set of quantified variables in the separable SyGuS specification ψ. The type or sort of each variable x_i is denoted by d_i. Given that ψ is separable, we can replace every occurrence of an application of f in the quantifier-free part, φ, of ψ with a single fresh variable o, whose type (or sort) is the same as the range of f to obtain the following logically equivalent formula:

$$\forall \mathbf{x} \exists o \ \varphi[o, \mathbf{x}]$$

Instead of attempting to solve for this formula directly, the CVC4 SyGuS solver attempts to establish the falsehood of the *negation* of this formula, which is:

$$\exists \mathbf{x} \forall o \ \neg\varphi[o, \mathbf{x}] \tag{6.2}$$

To prove that this formula is false, consider the following game played in rounds between Eloise and Abelard. At the beginning of round i Eloise proposes a region $R_i \subseteq d_1 \times d_2 \times \cdots \times d_n$, and Abelard proposes a *term* $t_i[\mathbf{x}]$, such that $\varphi[t_i[\mathbf{x}], \mathbf{x}]$ is true in some region S_i, such that $S_i \cap R_i \neq \emptyset$. Further, we have that $R_0 \equiv d_1 \times d_2 \times \cdots \times d_n$, and that $R_{i+1} \subseteq (R_i \setminus S_i)$ for all i. Abelard wins if in some round j, Eloise is forced to propose $R_j \equiv \emptyset$. Eloise wins if in some round j, Abelard is unable to come up with a term t_j. It is easy to see that (6.2) is false if and only if Abelard wins, and is satisfiable if and only if Eloise wins.

The game just described is the essence of the quantifier instantiation procedure performed within SMT solvers to prove the falsehood of formulas such as those shown in (6.2). The CVC4 SyGuS solver, takes advantage of being closely integrated with the CVC4 SMT solver, and having access to its internals. A proof of falsehood of (6.2) can then easily be used to construct an expression which serves as the solution for the unknown function f. Continuing with the game analogy, such a proof would consist of the terms t_i and the regions R_i, proposed by Abelard and Eloise respectively, for each round. Suppose that the regions R_i are represented symbolically as predicates, then it is trivial to construct an if-then-else ladder with the predicates corresponding

the regions as the conditions controlling which branch is chosen, and the appropriate terms t_i as the branches.

As an example of how this game may be played out on the specification shown in (6.1), which is for a binary function that computes the maximum of its arguments, we first write the formula whose falsehood is to be established from (6.1):

$$\exists x, y \; \forall o \; (o < x \vee o < y \vee (o \neq x \wedge o \neq y)) \tag{6.3}$$

0. In round 0, $R_0 \equiv \texttt{true}$, and $t_0 \equiv x$, which makes the formula (6.3) true in the region $x \geqslant y$, which is a subset of R_0
1. In round 1, $R_1 \equiv x < y$, and $t_1 \equiv y$, which makes the formula (6.3) true in the entire region R_1.
2. In round 2, Eloise is forced to set $R_2 \equiv \texttt{false}$, thus proving the falsehood of (6.3).

The Unification based SyGuS Solver

As was the case with the description of the CVC4 solver, this description of the unification based solver is also a highly condensed and simplified version of the presentation in the original paper describing this algorithm [ACR15]. The algorithm is conceptually similar to the algorithm used in the CVC4 SyGuS solver. However, the unification based algorithm uses an SMT solver as a black box, and does not depend on having access to the internals of an SMT solver. Given a separable SyGuS specification φ, whose form is as described earlier, the unification based solver maintains a region R, for which a correct solution has not yet been discovered. It then selects a term $t[\mathbf{x}]$ and plugs the term $t[\mathbf{x}]$ *back into* φ to determine a region R' where the term t causes φ to be true. The algorithm then recurses on the region $R \setminus R'$.

The working of the algorithm is perhaps best illustrated with an example. Consider the specification shown in (6.1) again. At the beginning of the algorithm, $R \equiv \texttt{true}$.

1. Suppose the algorithm picks the term x. Plugging this back into (6.1) for the term $f(x, y)$, we obtain the region $x \geqslant y$. The algorithm updates R to $x < y$.
2. The algorithm then picks the term y. Plugging this term back into (6.1), we obtain the region $y \geqslant x$. The algorithm updates R to be the empty region and terminates by unifying the terms x and y using an if-then-else ladder predicated with the regions in which substituting the respective term for $f(x, y)$ in ψ_{sep1} causes the formula to become true.

The key difference between the unification based algorithm and the CVC4 algorithm is in how terms are picked. The CVC4 algorithm piggy-backs on the sophisticated quantifier instantiation mechanisms within the SMT solver. The unification based algorithm on the other hand implements domain-specific solution techniques to derive terms that are likely to result in a solution with a smaller number of conditionals. The paper by Radhakrishna et. al. [ACR15] describes two algorithms to solve for terms: one for the domain of linear integer arithmetic, and another for the domain of fixed size bit vectors.

6.2.3 A Comparison of White Box and Black Box Algorithms

The white box algorithms described in this section have some advantages over black box algorithms. In turn, the black box algorithms have their own advantages over the white box algorithms. We specifically refer to the black box algorithm implemented in ESOLVER for the purposes of this comparison, although a lot of the points are applicable to the stochastic solver [SSA13] and the symbolic solver [JGST10, GJTV11] as well.

Strengths of White Box Algorithms

- **Enhanced Scalability:** The size of the expression to be synthesized does not have a large impact on the execution time of both the white box algorithms described in this section. Indeed, both of the algorithms can easily synthesize expressions with tens or hundreds of if-then-else branches. On the other hand, enumerative algorithms struggle to synthesize large expressions. This is primarily because the number of expressions in the search space typically grows exponentially with the size of allowed expressions. Because the enumerative approach enumerates *all* expressions of a given size before trying a larger size, it severely limits the scalability of a purely enumerative algorithm. The scalability of ESOLVER, as it is implemented, is also restricted by the fact that it caches every expression that it enumerates, leading to a large memory footprint.
- **Ability to use domain-specific techniques:** The white box algorithms leverage the specification itself in synthesizing a function that satisfies the specification. As a result, they can leverage domain-specific insights and algorithms to efficiently solve the sub-problems they construct. As already mentioned, the unification based solver implements a per-domain algorithm to choose terms. Similarly, the CVC4 SyGuS solver, which is deeply embedded within the CVC4 SMT solver, has a large portfolio of quantifier instantiation and domain

specific solution techniques — implemented as part of the CVC4 SMT solver — at its disposal, and can select the most efficient strategy on a domain-by-domain basis as required.

Strengths of the Enumerative Black Box Algorithm

- **Genericity:** ESOLVER uses the exact same algorithm regardless of what the domain being solved for is. Any improvements in the algorithm result in improvements across the board, for all domains. On the other hand, the white box algorithms' use of domain-specific solvers requires re-implementing any new algorithmic advance in each of the domain-specific algorithms.
- **Ability to Generalize:** Recall that the SyGuS language fully supports inductive specifications, or specifications where the behavior of the desired function is expressed as a finite set of concrete input-output examples. Such specifications can be useful when a formal specification is difficult to write. The ICFP benchmarks which were derived from a programming contest held in conjunction with ICFP 2013 [AII$^+$13]. The specifications for these benchmarks are in the form of a set of input-output examples which describe the output of the unknown function on various inputs. Assuming that the enumerative algorithm scales, it would produce the most concise expression in the search space that behaves correctly on all the input-output examples. Further, the output of the expression would be well-defined on unseen inputs. On the other hand, the white box solvers cannot do much better than generate a case-split on the concrete inputs, rendering the output of the expression being undefined or arbitrary on unseen inputs. It is thus not surprising that both of these solvers perform poorly on the ICFP benchmarks [ACR15, RDK$^+$15]. To be fair, ESOLVER does not perform well on these benchmarks either, but due to scalability constraints rather than algorithmic ones. In fact, none of the solvers that competed in the 2015 SyGuS competition were effective at solving these benchmarks.
- **Ease of Searching through a Syntactically Restricted Space:** Observe that the description of the white box algorithms does not mention the "Syntax-Guided" nature of SyGuS at all. The CVC4 algorithm either encodes the syntactic restriction using the theory of algebraic data types built into CVC4, or applies a enumerative post-processing step to find a term which is equivalent to the solution synthesized without syntactic restrictions. The former results in large slowdowns [RDK$^+$15], and the latter can sometimes result in failure. The unification based solver does not concern itself with syntactic restrictions at all; however,

the enumerative post-processing step used in the CVC4 algorithm can be used in this setting as well. Another possibility is to use a syntax aware unification operator, however this has not been explored. Syntactic restrictions are often useful when synthesizing programs for a low-power instruction set architecture, with a restricted set of operations, and are thus not an artificial constraint.

The comparison presented above naturally makes one desire an algorithm which can be generic, has the ability to generalize as well as the ability to enforce syntactic restrictions, alongside the scalability to be able to synthesize functions which require large expression sizes to describe. We describe an algorithm that fulfills this desire, at least to some extent, in the next section.

6.3 Combining Enumeration with Unification

To develop a more efficient algorithm to solve instances of the SyGuS problem, we make the following assumptions throughout this section:

- The SyGuS specification ψ is separable, and has the form $\psi \triangleq \exists f\ \forall \mathbf{x}\ \varphi[f, \mathbf{x}]$. Here f is the only function to be synthesized, and $\mathbf{x}$ is a set of universally quantified variables of appropriate types or sorts, while $\varphi[f, \mathbf{x}]$ is a quantifier-free formula that only refers to symbols from the background theory T, the unknown function f and the variables in the set $\mathbf{x}$.
- Given that ψ is separable, we can assume that all occurrences of f in *all* clauses of φ_{cnf} have f applied to the same arguments. If this is not the case, then we can transform $\varphi_{\text{cnf}}[f, \mathbf{x}]$ to a logically equivalent formula $\varphi_{\text{can}}[f, \mathbf{x}, \mathbf{a}]$, which is also in CNF by introducing a set of additional placeholder variables $\mathbf{a} \triangleq \{a_1, a_2, \ldots, a_p\}$, where $p = \mathsf{arity}(f)$ and $\mathbf{a} \cap \mathbf{x} \equiv \emptyset$ and constraining them appropriately. For example, consider the following specification for the binary function f, whose output is required to be greater than equal to each of its arguments, whose quantifier-free part is already in CNF:

$$\exists f\ \forall x, y\ f(x, y) \geqslant x \wedge f(y, x) \geqslant x$$

This formula can be transformed into the following semantically equivalent formula, by introducing additional variables a_0 and a_1 which represent the arguments to f which are used in all terms referring to f. Note that quantifier-free portion of the transformed formula

is also is CNF, once the implications have been converted into disjunctions using standard equivalences:

$$\exists f \, \forall x, y, a_0, a_1 \, (((a_0 = x \wedge a_1 = y) \Rightarrow f(a_0, a_1) \geqslant x) \wedge ((a_0 = y \wedge a_1 = x) \Rightarrow f(a_0, a_1) \geqslant y))$$

We will refer to the version of the specification ψ, canonicalized in this manner as ψ_{can}, and assume that it has the form $\psi_{can} \triangleq \exists f \, \forall \mathbf{x}, \mathbf{a} \; \varphi_{can}[f, \mathbf{x}, \mathbf{a}]$.

- Lastly, we assume that the program space is described by *two* context-free grammars, rather than just one unified grammar. The first grammar, which is a grammar for *terms*, denoted G_T comprises of the set of all terms, and does not include any conditional expressions. All terms generated by G_T have the same type as the range (or return type) of the unknown function f. The second grammar called G_P consists of the set of all Boolean valued *atomic predicates* that can be combined using Boolean connectives — disjunctions, conjunctions and negations — for use as the conditions in conditional expressions. Note that G_P is assumed not to contain disjunctions, conjunctions or negations of atomic predicates.[10] Further, we allow G_P and G_T to be mutually recursive, in that G_P can refer to non-terminals in G_T and vice-versa. We note that most of the grammars in the SyGuS benchmark suite can be transformed into this form relatively easily, using a conservative and lightweight analysis on the context-free grammar describing the syntactic restrictions on expressions. In terms of the assumptions on the *original* SyGuS grammar, we require that the original grammar must allow a solution to the SyGuS problem, such that the solution is either a *single* term drawn from G_T, or a conditional expression of the form:

```
if (cond_0) then term_0
else if (cond_1) then term_1
⋮
else if (cond_{n-1}) then term_{n-1}
else term_n
```

Where each $cond_i$ is a Boolean combinations of atoms, with each atom drawn from G_P, and each $term_i$ is a term drawn from G_T.

[10] The algorithm described here would still be correct if G_P contains Boolean combinations of atoms as well, but it would not be as efficient.

Algorithm 6.1: LEARN-DT: An algorithm to learn a decision tree

Input : A set of samples S.
An attribute function $\mathsf{attrib} : S \to \mathbb{B}^m$.
A labeling function $\mathsf{label} : S \to L$.

Output : A decision tree that uses the attributes of samples to predict its label.

1 **if** *all samples in* S *have the same label* l **then**
2 return a tree which predicts l
3 $a_{best} \leftarrow$ attribute $a_i, i \in [1, m]$ which *best* classifies S.
4 **if** a_{best} *is undefined* **then**
5 return $\perp$
6 $S^+ \leftarrow$ subset of S where each sample has $a_{best} = \texttt{true}$
7 $S^+ \leftarrow$ subset of S where each sample has $a_{best} = \texttt{false}$
8 positive $\leftarrow$ LEARN-DT(S^+, attrib, label)
9 negative $\leftarrow$ LEARN-DT(S^-, attrib, label)
10 **return** a decision tree labeled with attribute a_{best} with positive and negative as its positive and negative sub-trees

6.3.1 Decision Trees

Consider a set of samples $S \triangleq \{s_1, s_2, \ldots, s_n\}$ — each sample is some *object*, whose nature is not relevant. Each sample s_i is associated with a vector of m Boolean valued attributes. Let $\mathsf{attrib} : S \to \mathbb{B}^m$ be a function that maps each sample $s \in S$ to its attribute vector $\mathsf{attrib}(s)$. Further, we define L as a set of *labels*, with a labeling function $\mathsf{label} : S \to L$ which maps each sample $s \in S$ to its label $\mathsf{label}(s)$. Now, consider the problem of predicting $\mathsf{label}(s)$ for each $s \in S$, given information only about $\mathsf{attrib}(s)$ for each $s \in S$. This is a well studied problem in machine learning and is typically solved by using an algorithm, such as the ID3 algorithm or the C4.5 algorithm, to learn a reasonably compact decision tree which makes decisions based solely on the attributes [Qui86, Qui87, Qui96]. Algorithm 6.1 shows an algorithm to learn such a decision tree, which is now considered folk knowledge.

An interesting aspect of Algorithm 6.1 is how the *best* attribute is chosen in line 3. It has been shown that constructing the optimal (in terms of the size of the tree) decision tree is NP-complete [HR76, Mur98]. Because typical sample sets as well as the length of attribute vectors can be large, most algorithms to learn decision trees use a heuristic to greedily pick an attribute in line 3 of Algorithm 6.1. Greedy heuristics which maximize *information gain* at each level of the learned decision tree have been shown to be particularly effective in machine learning literature [Qui86, Qui87, Qui96].

Entropy and Information Gain

The entropy of a sample set S, denoted $H(S)$ is a measure of uncertainty in the set S. The mathematical definition of entropy, adapted to our setting is as follows:

$$H(S) = -\sum_{l \in L} Pr(l) \log_2(Pr(l)) \tag{6.4}$$

where $Pr(l)$ denotes the fraction of samples in S which are labeled l. Note that we refer to the Shannon entropy, whenever we use the term "entropy" in an unqualified manner throughout this dissertation. The concept of information gain is defined in terms of entropy. The information gain obtained by splitting a sample set S on an attribute a is the measure of the difference in entropy of S and the entropy of the resulting sets S^+ and S^-, which are formed by splitting on the attribute a. Mathematically, the information gain $G(S, a)$ obtained by splitting a sample set S, based on an attribute a, into two partitions S^+ and S^- can be computed by using the following equation:

$$G(S, a) = H(S) - \left(\frac{|S^+|}{|S|} H\left(S^+\right) + \frac{|S^-|}{|S|} H\left(S^+\right) \right) \tag{6.5}$$

Having provided the reader with an overview of decision trees and algorithms to learn such decision trees, we now present how they can be used, in conjunction with enumerative strategies, to solve instances of the SyGuS problem.

6.3.2 Program Synthesis using Decision Trees

Recall the Algorithm 4.1 SynthForPoints, respectively, shown in Chapter 4. Algorithm 4.1 essentially synthesizes *one* expression such that the expression satisfies the given specification for *all* the concrete inputs in a given set P. In essence, it enumerates *all* conditional expressions *implicitly* as a part of its search.

The basic idea behind the algorithm which we now present is that we do not need to synthesize an expression which satisfies the specification for *all* concrete inputs. We can learn a *set* E of expressions, such that each expression satisfies the specifications for some subset P' of the concrete inputs P, such that for *every* concrete input in $p \in P$, there exists *some* expression in $e \in E$ such that e satisfies the specification at p. Once we have gathered such a set E, we can then enumerate a sufficient set of atomic predicates from G_P. These atomic predicates can

then be combined using Boolean connectives to form the conditions in a conditional expression that combines the terms in E to produce an expression which is correct over *all* the concrete inputs. The computational problem of generating this conditional expression, which is correct over *all* the concrete inputs, can be reduced to one of learning an appropriate decision tree, as we describe in this section.

Formally, we are given a canonicalized, separable SyGuS specification for *one* function f of the form $\psi_{\text{can}} \triangleq \exists\, \mathsf{f}\ \forall \mathbf{x}, \mathbf{a}\ \varphi_{\text{can}}[\mathsf{f}, \mathbf{x}, \mathbf{a}]$ defined earlier in this section. We are also given two grammars G_T and G_P which are as described earlier. We abuse notation slightly, and also use G_T and G_P to refer to the *sets* of terms and predicates generated by the grammars G_T and G_P respectively, whenever the context creates no opportunity for ambiguity. Further, we have a set of *valuations* P of the variables in $\mathbf{x} \cup \mathbf{a}$, where each $\sigma \in P$ maps a variable $v \in \mathbf{x} \cup \mathbf{a}$ to a value $\sigma(v)$ of the appropriate type. We define a function $\mathcal{L} : P \to 2^{G_T}$, such that a term $t \in \mathcal{L}(p)$, for any point $p \in P$ if and only if $\varphi_{\text{can}}[t[p], \mathbf{x} \cup \mathbf{a} \mapsto p]$ evaluates to `true`. Note that the notation $\varphi_{\text{can}}[t[p], \mathbf{x} \cup \mathbf{a} \mapsto p]$ denotes that first *every* occurrence of all variables from $\mathbf{a}$ in t has been replaced by its valuation according to p, which is denoted as $t[p]$. Following this, every occurrence of $\mathsf{f}(.)$ in φ_{can} is replaced by $t[p]$, and lastly, all other occurrences of variables from $\mathbf{x} \cup \mathbf{a}$ in φ_{can} are also replaced by their valuations according to p, which is denoted by $\mathbf{x} \cup \mathbf{a} \mapsto p$.

Now, we can view the set of valuations P as a sample set. The labeling function is now essentially a *multi-labeling* function $\mathcal{L}$, which maps each point $p \in P$ to a *set* of labels drawn from the set G_T. Further, for each point $p \in P$, the results of evaluating each predicate $g \in G_P$ at p forms a vector of Boolean attributes for p, which may be of infinite length. Given these parallels, it is now clear how we can treat this as a decision tree learning problem, except for one wrinkle: that each sample may be multiply labeled. The possibility that a point may be labeled with multiple terms causes problems in the computation of entropy according to Equation (6.4), which requires the fraction of samples labeled with a particular label. Applying this equation naïvely will result in $\sum_{l \in L} \Pr(l) \neq 1$ and thus the function Pr will no longer be a probability mass function.

To deal with this wrinkle, given a sample set P, we define a conditional distribution on the probabilities of labels, *i.e.*, the probability of a point p being assigned a label $l \in \mathcal{L}(p)$, *conditioned* on the fact that a particular point $p \in P$ has been chosen. In the original single label formulation of the problem, this probability is either zero or one — once we pick a point

$p \in P$, we know that it can be assigned only *one* label: $\mathsf{label}(p)$. In the multi-label case, our formulation takes the view that once a point $p \in P$ has been picked, it can be assigned *any* label $l \in \mathcal{L}(p)$ according to a probability distribution. This conditional probability distribution is defined as follows:

$$\Pr(\mathsf{label}(p) = t \mid p) = \begin{cases} 0 & \text{if } t \notin \mathcal{L}(p) \\ \dfrac{\mathsf{cover}(t)}{\sum\limits_{t' \in \mathcal{L}(p)} \mathsf{cover}(t')} & \text{if } t \in \mathcal{L}(p) \end{cases} \tag{6.6}$$

where, given a sample set P, the function $\mathsf{cover} : G_T \to \mathbb{N}$ denotes how many samples in P can possibly be labeled with a given term $t \in G_T$, and is a rough measure of how *relevant* a particular term is. This function is defined as follows:

$$\mathsf{cover}(t) \equiv |\{p \in P : t \in \mathcal{L}(p)\}| \tag{6.7}$$

Now, given the sample set P, we can determine the unconditional label probabilities by summing the conditional probability shown in Equation 6.6 over *all* the points in P. Thus, we have, the probability of a randomly chosen point from P being labeled with $t \in G_T$ is:

$$\Pr(t) = \sum_{p \in P} \Pr(\mathsf{label}(p) = t \mid p) \times \Pr(p)$$

Now, assuming that each point $p \in P$ is equally likely to be chosen, *i.e.*, we sample from P uniformly at random, we obtain:

$$\Pr(t) = \frac{1}{|P|} \sum_{p \in P} \Pr(\mathsf{label}(p) = t \mid p) \tag{6.8}$$

We can now directly use Equation 6.8 to compute the entropy according to Equation 6.4, and thus information gain according to Equation 6.5, which can then be used to learn a decision tree based on the greedy information gain heuristic. Finally, we note that the conditional distribution that we have defined in Equation 6.6 makes intuitive sense, and works well in practice, as we will demonstrate shortly. However, we note that better choices for this probability distribution might still be possible, and this conditional distribution must therefore be viewed as *tunable heuristic* for the algorithm.

Row #	$p \in P$	$\mathcal{L}(p)$	$\mathsf{attrib}(p)$
1	$\langle x : 2, y : 1 \rangle$	$\{x\}$	$\langle x < y : \mathsf{F}, x = 0 : \mathsf{F}, y = 0 : \mathsf{F} \rangle$
2	$\langle x : 1, y : 0 \rangle$	$\{x, x + y\}$	$\langle x < y : \mathsf{F}, x = 0 : \mathsf{F}, y = 0 : \mathsf{T} \rangle$
3	$\langle x : 0, y : 1 \rangle$	$\{y, x + y\}$	$\langle x < y : \mathsf{T}, x = 0 : \mathsf{T}, y = 0 : \mathsf{F} \rangle$
4	$\langle x : 1, y : 2 \rangle$	$\{y\}$	$\langle x < y : \mathsf{T}, x = 0 : \mathsf{F}, y = 0 : \mathsf{F} \rangle$

Table 6.1: A multi-labelled sample set over which a decision tree is to be learned

An Illustrative Example

We now illustrate the techniques which we have just described, with an example. Consider the following specification which describes a binary function f, over integers, which is expected to return the maximum of its arguments:

$$\exists f \, \forall x, y \; f(x, y) \geqslant x \wedge f(x, y) \geqslant y \wedge (f(x, y) = x \vee f(x, y) = y$$

Suppose that the set of terms that we're working with is $\{x, y, x + y\}$ and the set of predicates is $\{x < y, x = 0, y = 0\}$. Further, the set P for our example contains the four valuations shown in the second column of Table 6.1, with the third column showing the set of labels (terms) that satisfy the specification at each sample (or point), and the fourth column showing the attribute vector, which consists of predicates, and their truth value for the corresponding point. For instance, the row numbered one in the table considers the valuation where x is two and y is one. We see that the term x is the only term from among the terms x, $x + y$ and y that satisfies the specification this point. Lastly, for this valuation, all the predicates that we consider, *i.e.*, the predicates $x < y$, $y = 0$ and $x = 0$, evaluate to false as shown in the last column.

To learn a decision tree over this sample set, we need to evaluate the entropies that result from splitting the set of valuations on each of the atomic predicates. We then choose the predicate, splitting on which results in the smallest entropy, and split the set of valuations according to the predicate. To illustrate, let us first consider splitting this sample set according to the predicate $x < y$. Splitting the set of valuations using this predicate yields two partitions the set of valuations P. Let us refer to these partitions P_1 and P_2, where P_1 contains the rows numbered one and two — where $x < y$ evaluates to false — and P_2 contains the rows numbered three and four — where $x < y$ evaluates to true. We need to compute the entropy for each of these partitions. The total entropy for the partitioned set of valuations is then

Partition	Points in Partition	Label Probabilities			Entropy
P_1	$\langle x : 2, y : 1\rangle$ $\langle x : 1, y : 0\rangle$	$\Pr(\mathsf{label}(p) = x)$	$=$	$\frac{5}{6}$	0.650022
		$\Pr(\mathsf{label}(p) = x + y)$	$=$	$\frac{1}{6}$	
		$\Pr(\mathsf{label}(p) = y)$	$=$	0	
P_2	$\langle x : 0, y : 1\rangle$ $\langle x : 1, y : 2\rangle$	$\Pr(\mathsf{label}(p) = x)$	$=$	0	0.650022
		$\Pr(\mathsf{label}(p) = x + y)$	$=$	$\frac{1}{6}$	
		$\Pr(\mathsf{label}(p) = y)$	$=$	$\frac{5}{6}$	

Table 6.2: Entropies that result by splitting the sample set shown in Table 6.1 using the predicate $x < y$

Partition	Points in Partition	Label Probabilities			Entropy
P_1	$\langle x : 2, y : 1\rangle$ $\langle x : 1, y : 0\rangle$ $\langle x : 1, y : 2\rangle$	$\Pr(\mathsf{label}(p) = x)$	$=$	$\frac{5}{9}$	1.351644
		$\Pr(\mathsf{label}(p) = x + y)$	$=$	$\frac{1}{9}$	
		$\Pr(\mathsf{label}(p) = y)$	$=$	$\frac{1}{3}$	
P_2	$\langle x : 0, y : 1\rangle$	$\Pr(\mathsf{label}(p) = x)$	$=$	0	0.5
		$\Pr(\mathsf{label}(p) = x + y)$	$=$	$\frac{1}{2}$	
		$\Pr(\mathsf{label}(p) = y)$	$=$	$\frac{5}{2}$	

Table 6.3: Entropies that result by splitting the sample set shown in Table 6.1 using the predicate $x = 0$

the sum of entropies of each of these partitions, weighted by the fraction of valuations in the respective partition.

Table 6.2 shows the partitions that result from splitting on the predicate $x < y$, as well as the label probabilities computed according to Equation 6.8. Finally, the entropy corresponding to each partition are computed according to Equation 6.4, using the set $\{x, y, x + y\}$ as the set of all possible labels. Note that in this table, the partition named P_1 corresponds to the rows in Table 6.1 where the predicate $x < y$ evaluates to false, and the partition P_2 corresponds to the rows where the predicate $x < y$ evaluates to true. Also, for the purposes of entropy calculations, we assume that $0 \times \log_2(0) = 0$. The overall entropy that results from the split using the predicate $x < y$ is the weighted sum $\frac{1}{2} \times 0.650022 + \frac{1}{2} \times 0.650022 = 0.650022$.

Now, repeating the same procedure to determine the entropy obtained by splitting on the predicate $x = 0$ yields the results shown in Table 6.3. The overall entropy from the split is the weighted sum $\frac{3}{4} \times 1.351644 = \frac{1}{4} \times 0.5 = 1.138733$. The results of splitting on the predicate

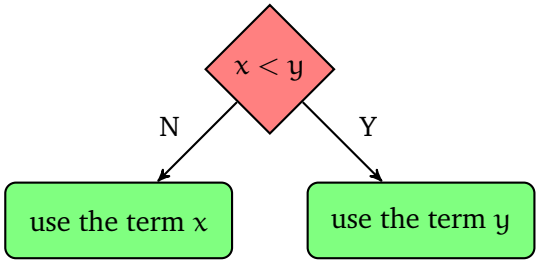


Figure 6.1: The decision tree learned for the sample set shown in Table 6.1

Algorithm 6.2: EXPANDTERMSET: Expand the labeling function $\mathcal{L}$ to include more terms

Input : A canonicalized SyGuS specification $\psi_{can} \triangleq \exists f\, \forall \mathbf{a}, \mathbf{x}\ \varphi_{can}[f, \mathbf{x}, \mathbf{a}]$.
A list of n valuations of variables in $\mathbf{x} \cup \mathbf{a}$, called P.
A stateful enumerator $\mathsf{enumerator}(G_T)$ for terms.
A map $\mathcal{L}$ from P to subsets of terms from G_T.
Output : An expanded map $\mathcal{L}'$, such that for all $p \in P, \mathcal{L}'(p) \supseteq \mathcal{L}(p)$.

```
1 new_terms ← the next K_T terms from enumerator(G_T)
2 foreach t ∈ new_terms do
3     s ← ⟨φ_can[t[p], x ∪ a ↦ p], for p in P⟩
4     if there exists a term t' ≠ t, such that for all i ∈ [1, length(P)], s[i] iff t' ∈ L(P[i]) then
5         continue
6     foreach i ∈ [1, length(p)] such that s[i] = true do
7         L[P[i]] ← L[P[i]] ∪ {t}
8 return L
```

$y = 0$ will be similar, as the cases $x = 0$ and $y = 0$ are symmetric, and will hence result in the exact same entropy and are not shown here. Thus, the entropy obtained by splitting on the predicate $x < y$ is the minimum among the choices, and will therefore yield the highest information gain. So, the decision tree learning algorithm splits according to the predicate $x < y$ at the first level. Once this has been done, notice that the sample set P_1 that results from the split, can be labeled consistently by the label x, which results in the specification being satisfied at all the valuations in the set. Similarly, the label y can be chosen for the set P_2. Thus, the decision tree learned for this example is as shown in Figure 6.1. From this tree, the expression $\mathsf{ite}(x < y, y, x)$ can easily be deduced, which is a correct solution for this example.

6.3.3 Putting it all Together

Algorithm 6.3 describes how the solver that combines enumeration and unification, which we dub EUSOLVER, computes a set of terms that when taken together could form a complete solution. The loop at line 1 of Algorithm 6.3, continues enumerating terms from the term

Algorithm 6.3: TERMSOLVE: Algorithm to find a set of expressions which together satisfy the specification for a given set of points

Input : A canonicalized SyGuS specification $\psi_{\text{can}} \triangleq \exists\, f\, \forall\, \mathbf{a}, \mathbf{x}\ \varphi_{\text{can}}[f, \mathbf{x}, \mathbf{a}]$.
A list of n valuations of variables in $\mathbf{x} \cup \mathbf{a}$, called P.
A stateful enumerator $\mathsf{enumerator}(G_T)$ for terms.
Output : A map $\mathcal{L}$ from P to non-empty sets of terms from G_T.
Data : The partially computed output, $\mathcal{L}$, which initially maps everything to $\emptyset$.

1 **while** *there exists* $p \in P$*, such that* $\mathcal{L}(p) \equiv \emptyset$ **do**
2 $\quad \mathcal{L} \leftarrow$ EXPANDTERMSET(ψ_{can}, P, $\mathsf{enumerator}(G_T)$, $\mathcal{L}$)
3 **return** $\mathcal{L}$

grammar G_T until it finds a set of terms such that for every valuation $p \in P$ there exists some term t in $\mathcal{L}$ such that the term t satisfies the specification φ_{can} when evaluated at the point p. Algorithm 6.2 *expands* the mapping $\mathcal{L}$ to include more terms. This is tantamount to expanding the set of terms that are allowed to be a part of a solution. As an optimization, if two terms t_1 and t_2 satisfy the specification on the same subset of points in P, then we only retain one of them in the map $\mathcal{L}$, in Algorithm 6.2. The number K_T referred to in Algorithm 6.2 is a tunable parameter, which was set to eight in all our experiments. Finally, Algorithm 6.3 returns the map it has built up once the stopping condition described earlier has been reached.

Given such a map $\mathcal{L}$, the algorithm UNIFYTERMS, which is shown in Algorithm 6.4 is then used to unify these terms using conditionals, where the predicates for the conditional are Boolean combinations of atoms drawn from G_P. The algorithm works by enumerating sets of the K_P atoms from G_T in each iteration. Here K_P is a parameter; in all our experiments, this was set to a value of eight. Once this set of atoms has been enumerated, the algorithm computes the points $p \in P$, where *each* atom in this set evaluates to true, and stores it in the map attrmap. An optimization similar to the one described in TERMSOLVE is applied here: if two atoms evaluate identically on all the points in P, then only one of them is retained. Once the map attrmap has been computed for the current batch of atoms, the algorithm then attempts to learn a decision tree. If this step fails, there could be two reasons for the failure:

1. The current set of atoms are sufficient to learn a correct classifier, in which case the algorithm would need to enumerate more atoms.
2. The current set of terms under consideration require that we learn a classifier to separate two points $p_1 \in P$ and $p_2 \in P$. This could happen because two distinct terms, say t_1 and t_2, satisfy the specification at p_1 and p_2, respectively, and no other terms satisfy the

Algorithm 6.4: UNIFYTERMS: Attempt to combine sub-expressions

Input : A canonicalized SyGuS specification $\psi_{can} \triangleq \exists\, f\ \forall\, \mathbf{a}, \mathbf{x}\ \varphi_{can}[f, \mathbf{x}, \mathbf{a}]$.
A list of n valuations of variables in $\mathbf{x} \cup \mathbf{a}$, called P.
A map $\mathcal{L}$ from P to non-empty sets of terms from G_T.
A stateful enumerator $\mathsf{enumerator}(G_P)$ for atoms.
A stateful enumerator $\mathsf{enumerator}(G_T)$ for terms.
Output : Either a solution e for ψ_{can}, or a valuation p of variables in $\mathbf{x} \cup \mathbf{a}$.
Data : A map attrmap, from predicates in G_P to a bit vector of length $\mathsf{length}(P)$.

```
1  do
2      aps ← the next K_P atomic predicates from enumerator(G_P)
3      foreach ap ∈ aps do
4          sig ← ⟨ap[p] for p in P⟩
5          if there exists ap′ ≠ ap, such that attrmap[ap′] ≡ sig then
6              continue
7          attrmap[ap] ← sig
8      dtree ← LEARN-DT(P, attrmap, L)
9      if dtree ≠ ⊥ then
10         e ← expression constructed from dtree
11         if VERIFY(e, ψ_can) then
12             return e
13         else
14             return a valuation σ of variables in x ∪ a which form a verification counterexample
15     else
16         L ← EXPANDTERMSET(ψ_can, P, enumerator(G_T), L)
17 while (dtree = ⊥)
```

specification at these two points. Now, it could be impossible to separate p_1 and p_2, based on the predicates defined by G_P. However, a solution might still be possible if there exists a term $t' \in G_T$, such that it satisfies the specification at both p_1 *and* p_2. In this situation, the algorithm would need to enumerate more terms.

It is not obvious how one can accurately determine which of these two reasons caused the attempt to learn a decision tree to fail. Given this difficulty, Algorithm 6.4 conservatively expands the set of terms currently in consideration (at Line 16 in Algorithm 6.4), as well as the set of atoms used to construct a decision tree (at Line 2, at the beginning of the next iteration of the loop), whenever an attempt to learn a decision tree fails. Note that attrmap retains its value across iterations of the outermost loop in Algorithm 6.4. On the other hand, if it was possible to learn a decision tree, the algorithm extracts an expression from the learned decision tree. This can be achieved in multiple ways; one possible way is to walk down every path from the root to the leaves, gathering the atoms that internal nodes are labeled with, together

Algorithm 6.5: EUSOLVE: Solve for a SyGuS specification ψ_{can}

Input : A canonicalized SyGuS specification $\psi_{\text{can}} \triangleq \exists\, f\, \forall\, \mathbf{a}, \mathbf{x}\ \varphi_{\text{can}}[f, \mathbf{x}, \mathbf{a}]$.
A grammar for terms G_T.
A grammar for atoms G_P.
Output : A solution e for the SyGuS specification ψ_{can}
Data : A list of valuations P of variables in $\mathbf{x} \cup \mathbf{a}$, initially empty.
enumerator(G_T), a systematic, stateful enumerator which enumerates terms from G_T.
enumerator(G_P), a systematic, stateful enumerator which enumerates atoms from G_P.

```
1  while true do
2      if length(P) = 0 then
3          e ← the first term from enumerator(G_T)
4          if VERIFY(e, ψ_can) then
5              return e
6          else
7              σ ← a valuation of variables in x ∪ a which forms a verification counterexample
8              append σ to P
9              continue
10     L ← TERMSOLVE(ψ_can, P, enumerator(G_T))
11     solorcex ← UNIFYTERMS(ψ_can, P, L, enumerator(G_P), enumerator(G_T))
12     if solorcex is an expression then
13         return solorcex
14     else
15         append solorcex to P
16         continue
```

with their polarity. When a leaf node is reached, the label at the leaf node provides the term, and the conjunction of the accumulated atoms forms the condition under which the term can be used. Once such an expression e has been built, the algorithm attempts to verify that e is a solution to the SyGuS specification ψ_{can}. The verification step is performed by posing an appropriate query to an SMT solver. We use the SMT solver Z3 [dMB08] in our implementation. If this verification succeeds, it returns e. Otherwise, it returns a counterexample to verification, which is a valuation of the variables in $\mathbf{x} \cup \mathbf{a}$, such that the expression e does not satisfy the specification on that valuation.

Finally, the algorithm EUSOLVE, shown as Algorithm 6.5, shows how the TERMSOLVE and UNIFYTERMS algorithms are composed to form a complete SyGuS solver. The algorithm maintains a list of valuations P, which are built up from counterexamples returned by the algorithm UNIFYTERMS. It repeatedly calls the algorithm TERMSOLVE, followed by the algorithm UNIFYTERMS, augmenting the list of valuations P in each iteration, until UNIFYTERMS returns a solution that has been verified to be a correct solution to the SyGuS specification ψ_{can}.

Correctness of the Algorithm EUSOLVE

We now argue that Algorithm 6.5 is a semi-decision procedure, *i.e.*, if there exists a solution in the form of a conditional expression in the grammars defined by G_T and G_P, the algorithm terminates with a correct solution. If the grammars G_T and G_P do not admit a solution in the form of a conditional expression, then Algorithm 6.5 can run forever. We now formalize and prove these guarantees, that are provided by Algorithm 6.5, EUSOLVE, in the following theorem.

Theorem 2. *Given a plainly separable* SyGuS *specification* ψ*, a term grammar* G_T *and a predicate grammar* G_P*, if there exists a solution of the following form:*

$$
\begin{aligned}
&\texttt{if } (c_0) \texttt{ then } t_0\\
&\texttt{else if } (c_1) \texttt{ then } t_1\\
&\vdots\\
&\texttt{else if } (c_{n-1}) \texttt{ then } t_{n-1}\\
&\texttt{else } term_n
\end{aligned}
$$

where $c_0, c_1, \ldots, c_{n-1}$ *are Boolean combinations of atomic predicates drawn from* G_P *and* $t_0, t_1, \ldots, t_n$ *are drawn from* G_P*, then Algorithm 6.5,* EUSOLVE *terminates and returns a correct solution.*

Proof. We first note that it is sufficient to consider conjunctions of *literals*, where a literal is either an atomic predicate or its negation in the conditionals $\{c_i\}$. Suppose that the grammars admit a solution of the form:

$$
\begin{aligned}
&\texttt{if } (l_0 \vee l_1) \texttt{ then } t_0\\
&\texttt{else } t_1
\end{aligned}
$$

then, by leveraging that an if-then-else construct is essentially disjunctive, it also admits the following equivalent solution:

$$
\begin{aligned}
&\texttt{if } (l_0) \texttt{ then } t_0\\
&\texttt{else if } (l_1) \texttt{ then } t_0\\
&\texttt{else } t_1
\end{aligned}
$$

So, without loss of generality, we will assume that a correct solution admitted by the grammars G_T and G_P has the following form:

$$
\begin{aligned}
&\texttt{if } (l_{0,0} \wedge l_{0,1} \wedge \cdots \wedge l_{0,k_0}) \texttt{ then } t_0 \\
&\texttt{else if } (l_{1,0} \wedge l_{1,1} \wedge \cdots \wedge l_{1,k_1}) \texttt{ then } t_1 \\
&\vdots \\
&\texttt{else if } (l_{n-1,0} \wedge l_{n-1,1} \wedge \cdots \wedge l_{n-1,k_{n-1}}) \texttt{ then } t_{n-1} \\
&\texttt{else } t_n
\end{aligned}
$$

where each $l_{i,j}$ is either an atomic predicate drawn from G_P or its negation, and each t_i is a term drawn from G_T. Let us define the set $\mathsf{Terms} \equiv \{t_0, t_1, \ldots, t_n\}$, as well as the set $\mathsf{Lits} \equiv \{l_{0,0}, \ldots, l_{0,k_0}, \ldots, l_{n-1,0}, \ldots, l_{n-1,k_{n-1}}\}$. Note that both the sets Terms and Lits are finite. We now make the following observations:

1. For any given set of terms and literals, there are only finitely many syntactically distinct conditional expressions that can be formed using the available terms and literals.
2. The set of distinct decision trees over a finite set of terms and literals, and given a finite set of samples, is also finite.
3. We can map every decision tree over a finite set of terms and literals, which classifies a finite set of samples, to a syntactically unique conditional expression. The number of terms in such a conditional expression is equal to the number of leaves in the decision tree, and the condition on each branch is the conjunction of literals along the path to the corresponding leaf (term).
4. Algorithm 6.5 makes *progress*: If the verification of a particular expression fails — either in Algorithm 6.5 or in Algorithm 6.4 — then that particular expression will never be presented to the SMT solver for verification at any subsequent point during the execution of the algorithm. To see that this is true, observe that Algorithm 6.1 *always* returns a decision tree which correctly classifies the sample set, or reports that no decision tree exists. Further Algorithm 6.1 is *sound* and *complete*, *i.e.*, it always returns a decision tree which correctly classifies the sample set if one exists. A verification attempt only occurs when a decision tree can be learned. Now, suppose that a particular verification attempt resulted in the candidate expression being proved incorrect. A valuation that demonstrates the incorrectness of the candidate must have been added to the list P maintained by Algorithm 6.5. Now, if the same decision tree was ever returned by Algorithm 6.1, then that decision tree will incorrectly classify this newly added point (valuation). This is in contradiction with the fact that Algorithm 6.1 always returns a correct classifier for a given sample set.

Based on these observations, we now only need to prove that a sufficient set of terms and atomic predicates will eventually be enumerated by the Algorithm 6.3 and Algorithm 6.4 respectively. This follows from the observations of finiteness and progress made above, and the fact that the sets defined by the grammars G_T and G_P are recursively enumerable. Thus, at some point it must be the case that:

$$\bigcup_{p \in P} \mathcal{L}(p) \supseteq \mathsf{Terms} \tag{6.9}$$

where $\mathcal{L}$ is the mapping returned by the Algorithm 6.3, TermSolve. In other words, a sufficient set of terms will eventually be enumerated by the algorithm. We can use a similar argument to prove that the algorithm also eventually enumerates a sufficient set of atomic predicates corresponding to the set of literals Lits. Formally it must be the case that at some point during the execution of Algorithm 6.4, it must be the case that:

$$\bigcup_{ap \in aps} \{ap, \neg ap\} \supseteq \mathsf{Lits} \tag{6.10}$$

where aps is the set of atomic predicates generated during the execution of Algorithm 6.4. We now argue that once the conditions described by the formulas 6.9 and 6.10 are met, the mapping $\mathcal{L}$ and the set aps in Algorithm 6.3 and Algorithm 6.4 respectively, remain unchanged in all future invocations.

To see why this is true, recall that we made the assumption that there exists a solution involving only the terms in the set Terms and the literals in the set Lits. This means that for *any* set of concrete valuations P (maintained by Algorithm 6.5), there must be some term $t \in \mathsf{Terms}$ that satisfies the specification for that valuation. So based on its termination condition, Algorithm 6.3 will never enumerate a larger set of terms. Furthermore, for any set of valuations P, there must also exist a decision tree that correctly classifies the valuations using the predicates as splitting attributes and the terms as labels. So, Algorithm 6.4 will never need to enumerate a larger set of predicates — because Algorithm 6.1 will always return some decision tree.

Based on the observations of finiteness and progress that we have made earlier, we know that there are only a finite number of expressions that can be formed using the set of terms in the (now unchanging) map $\mathcal{L}$, and the (again, now unchanging) set of atomic predicates aps.

```
(set-logic BV)

(define-fun shr1 ((x (BitVec 64))) (BitVec 64) (bvlshr x #x0000000000000001))
(define-fun shr4 ((x (BitVec 64))) (BitVec 64) (bvlshr x #x0000000000000004))
(define-fun shr16 ((x (BitVec 64))) (BitVec 64) (bvlshr x #x0000000000000010))
(define-fun shl1 ((x (BitVec 64))) (BitVec 64) (bvshl x #x0000000000000001))
(define-fun if0 ((x (BitVec 64)) (y (BitVec 64)) (z (BitVec 64))) (BitVec 64)
                  (ite (= x #x0000000000000001) y z))

(synth-fun f ((x (BitVec 64))) (BitVec 64)
             ((Start (BitVec 64)
               (#x0000000000000000 #x0000000000000001 x
                (bvnot Start) (shl1 Start) (shr1 Start)
                (shr4 Start) (shr16 Start) (bvand Start Start)
                (bvor Start Start) (bvxor Start Start)
                (bvadd Start Start) (if0 Start Start Start)))))

(constraint (= (f #x85c12c65236e72be) #x85c1ade52f6f73fe))
(constraint (= (f #xe1207ed6c7320aa4) #x70903f6b63990553))
.
.
.WWW

(check-synth)
```

Figure 6.2: Anatomy of an ICFP Benchmark

The progress property ensures that the same expression is never submitted for a verification attempt more than once. Thus, we can conclude that eventually, Algorithm 6.5 will attempt to verify the correct solution and return it. □

6.3.4 Evaluation of EUSOLVER

We built a prototype version of EUSOLVER using the Z3 SMT solver [dMB08] for verification. The prototype implemented the expression enumeration parts and the high level algorithm in Python, whereas the decision tree learning algorithms as well as some performance critical bit vector manipulation routines were implemented in C++. Our experiments were conducted on an Intel Core i7 processor running at 2GHz. All experiments were run with a time out 1800 seconds per benchmark. We evaluated EUSOLVER on the following subset of the SyGuS main track benchmarks:

- **Integer Arithmetic:** We evaluated EUSOLVER on a set of benchmarks which compute the maximum of a some number of arguments. The actual number of arguments can be

parameterized. We were able to scale reasonably well on this set of benchmarks, as the value for the parameter was increased.

- **ICFP Benchmarks:** As mentioned earlier, the specifications for these 50 benchmarks were in the form of a number of input-output examples which describe the output of the function to be synthesized for various inputs. No other solver has been able to solve more than a handful of these benchmarks, to the best of our knowledge. EUSOLVER was able to solve more than 80% of the benchmarks (42 out of 50) with a 30 minute time limit for each benchmark.

We did not evaluate EUSOLVER on the other tracks, because the solutions to these tracks did not consist of large if-then-else expressions. Also, the original ESOLVER could solve most of these benchmarks. For a more universal solver, one could imagine running a portfolio solver with the original ESOLVER algorithm running on one thread, with the EUSOLVER algorithm running on another thread. Such a solver would be able to solve a sizeable fraction of the SyGuS benchmark suite as it stands today. Table 6.4 summarizes the results of running EUSOLVER on the ICFP benchmarks. In contrast, CVC4, the winner of 2015 SyGuS contest could only solve one ICFP benchmarks when syntactic restrictions were applied, and 43, when syntactic restrictions were not applied. We note that all our solutions are within the syntax specified by the benchmarks. Lastly, we did not observe the solver memory usage using exceeding 100 MB for any benchmark. As a final comparison EUSOLVER was able to produce syntactically valid solutions for 42 out of 50 ICFP benchmarks in a total of 7630 seconds, whereas, the CVC4 solver could solve 43 out of the 50 benchmarks in 3400 seconds [RDK+15], but the solutions were not syntactically valid and used arbitrary function symbols form the SMTLIB theory of fixed-size bit-vectors.

To provide some context to the reader, Figure 6.2 shows a typical ICFP benchmark. Note that the benchmark has been reproduced almost verbatim from the actual benchmark used in the SyGuS competition. The only changes we have made are to elide a large set of input-output constraints from line 17 – 20, and some whitespace and adjustments, for better readability. Further, we emphasize that the syntactic restrictions that we have discussed earlier are an integral part of the benchmark. The first line declares that the logic of fixed-size bit-vectors is to be used. Lines 2 – 5 declare the macros named `shr1`, `shr4`, `shr16`, `shl1`, each of which takes a 64-bit bitvector as an argument and returns another 64-bit bitvector, shifted by right or left by the appropriate constant. Lastly, lines 6 and 7 declare a macro named `if0`, which is a

Benchmark	Time (s)	Exp. size	|P|
icfp_103_10	38.9	55	9
icfp_104_10	1.0	24	3
icfp_105_100	2.3	23	4
icfp_105_1000	24.5	22	4
icfp_113_1000	114.9	11	2
icfp_114_100	665	26	3
icfp_118_10	10.1	54	6
icfp_118_100	51.4	49	4
icfp_125_10	19.7	28	7
icfp_134_1000	TO	–	–
icfp_135_100	158	13	2
icfp_139_10	3.3	10	2
icfp_143_1000	TO	–	–
icfp_144_100	1525	39	11
icfp_144_1000	TO	–	–
icfp_147_1000	TO	–	–
icfp_14_1000	TO	–	–
icfp_150_10	4.7	52	7
icfp_21_1000	1069	28	5
icfp_25_1000	125	29	5
icfp_28_10	0.17	2	1
icfp_30_10	40.4	14	4
icfp_32_10	25.9	14	2
icfp_38_10	13.1	27	5
icfp_39_100	40.7	12	2

Benchmark	Time (s)	Exp. size	|P|
icfp_45_10	0.48	9	2
icfp_45_1000	32.2	9	2
icfp_51_10	4.62	11	2
icfp_54_1000	69.8	11	2
icfp_56_1000	TO	–	–
icfp_5_1000	60.5	32	4
icfp_64_10	46.1	33	4
icfp_68_1000	37.6	46	7
icfp_69_10	1.82	11	4
icfp_72_10	47.9	13	2
icfp_73_10	1.15	24	3
icfp_7_10	1.61	24	5
icfp_7_1000	66.2	30	9
icfp_81_1000	1318	37	7
icfp_82_10	17.1	32	7
icfp_82_100	31.7	30	10
icfp_87_10	13.1	31	5
icfp_93_1000	174	29	5
icfp_94_100	2.58	24	4
icfp_94_1000	30.1	24	4
icfp_95_100	829	47	35
icfp_96_10	35.1	48	8
icfp_96_1000	TO	–	–
icfp_99_100	876	25	4
icfp_9_1000	TO	–	–

Table 6.4: Experimental Results for EUSOLVER on the ICFP benchmarks. The column labeled "Time" indicates the time taken to arrive at a solution. The column labeled "Exp. size" indicates the size of the computed expression, and the column labeled |P| indicates the number of counterexamples that were considered by the algorithm before arriving at a correct solution. TO indicates a timeout.

restricted form of conditional which takes three 64-bit bitvectors as arguments and returns the second argument if the first argument is equal to the bitvector constant "1", otherwise returns the third argument. Line 8 declares a function `f` which is to be synthesized, which takes in one 64-bit bitvector as an argument, which is referred to as `x` — this is the formal parameter name — and returns a 64-bit bitvector. Lines 9 – 14 describe the grammar for the interpretation of `f`. Line 9 declares a non-terminal named `Start` which expands to a 64-bit

Benchmark	EUSOLVER Time (s)	EUSOLVER Exp. Size	EUSOLVER |P|	CVC4 Time (s)	STUN Time (s)
max2	0.05	6	2	0.01	0.094
max3	0.16	30	15	0.02	0.087
max4	0.56	94	43	0.03	0.097
max5	3.18	254	160	0.05	0.179
max6	17.3	634	544	0.1	0.167
max7	131.7	1510	2080	0.3	0.230
max8	1296	3490	7734	1.6	0.267
max9	TO	–	–	8.9	0.277
max10	TO	–	–	81.5	0.333
max11	TO	–	–	ND	0.371
max12	TO	–	–	ND	0.441
max13	TO	–	–	ND	0.554
max14	TO	–	–	ND	0.597
max15	TO	–	–	ND	0.675

Table 6.5: Experimental Results for EUSOLVER on the MAX benchmarks. The first four columns have the same meaning as in Table 6.4. The next two columns show the times taken by the CVC4 solver [RDK+15] and the STUN solver [ACR15] on the same benchmarks. TO indicates a time-out and ND indicates that the data was not available.

bitvector value. Line 10 lists three expansions: The constants "0", "1", or the formal parameter `x`. Lines 11 – 14 describe other, recursive expansions, involving standard functions like `bvnot`, `bvadd`, etc., from the SMTLIB theory of fixed-size bitvectors, as well as macros defined in lines 2 – 7. The constraints on the behavior of `f` are described from line 16 onwards. Each constraint is an input-output example, which constrains the result of `f` applied to a constant value, to another constant value.

All the 50 ICFP benchmarks are similar in structure to the one shown in Figure 6.2, *i.e.*, they all use the same set of macros and the same grammar. However, the constraints themselves differ to describe different functions `f`. These constraints are obviously underspecified; they do not completely describe the behavior of `f` on all inputs. To successfully solve such constraints, a SyGuS solver would need to perform a non-trivial amount of generalization. As we demonstrate, EUSOLVER is able to generalize well from these constraints and successfully solve a large fraction of the ICFP benchmarks within a reasonable amount of time.

Table 6.5 demonstrates the performance of EUSOLVER on the parametric MAX benchmark from the SyGuS suite. On this set of benchmarks, EUSOLVER performs better than the original

ESOLVER, which times out on all benchmarks beyond max3. However, it is not as performant as the CVC4 and the STUN solvers on these benchmarks. Our investigations reveal that a majority of the time is spent in decision tree learning on the larger MAX benchmarks. Indeed, the number of counterexamples points added shown in the column labeled |P| in Table 6.5 seems to grow very rapidly with larger instantiations of the MAX benchmarks. The reasons for why such a large number of counterexamples are considered by the algorithm are unclear and warrant a closer investigation. In contrast, the CVC4 and STUN solvers show a much smaller slowdown on larger instances of the MAX benchmark.

A Note on Expression Sizes

The expression sizes reported in Tables 6.4 and 6.5 were for the expressions obtained by the simplistic strategy to convert a decision tree into an expression, which we have described earlier in Section 6.3.3. Such a strategy returns an expression with a *flat* conditional structure, *i.e.*, with only a top level case split and no nested conditionals. In some cases, it is possible that by allowing nested conditionals and applying slightly more sophisticated simplification steps as post-processing, a smaller sized expression can be obtained. We did not explore such simplifications and post-processing steps.

To conclude this chapter, we have presented a generic enumeration based algorithm to solve separable SyGuS instances, where the grammar can be easily separated into a grammar for atomic predicates and a grammar for terms. We have demonstrated the efficacy of the new algorithm in solving a large fraction of the ICFP benchmarks in the SyGuS benchmark suite, while respecting all the syntactic restrictions. To the best of our knowledge, this is algorithm is the first to be able to successfully solve such a large fraction of the ICFP benchmarks. This chapter concludes the digression towards the SyGuS problem. We now turn our attention back to the problem of distributed protocol synthesis in the subsequent chapters of this dissertation.

7

Synthesis of Finite-state Protocols from Scenarios and Specifications

We now turn our attention back to the protocol completion problem. As mentioned in Section 4.4, the TRANSIT tool has some limitations: (1) TRANSIT cannot synthesize transitions which are missing from the input, (2) TRANSIT does not handle liveness properties, and (3) TRANSIT requires the programmer to be in the loop. We seek to at least partially address these limitations with the work described in this section. We develop a fully automatic approach to synthesize protocols which are described using scenarios over finite-state machines — *i.e.*, no state variables — given a set of safety and liveness requirements that the completed protocol is expected to satisfy. This chapter is based on the work originally published in [AMR+14].

7.1 Overview of Finite-state Protocol Synthesis

Figure 7.1 provides a high-level overview of the process of synthesizing finite-state protocols from scenarios as an instantiation of the algorithmic scheme shown in Figure 3.2. The programmer provides a set of scenarios — which are essentially execution traces of the protocol under construction, and will be described in detail in subsequent sections — and the protocol skeleton. The protocol skeleton lists the state machines and state machine sketches that comprise the protocol, their input and output alphabets, and the set of locations L for each state machine (in the case of uncontrollable environment state machines) or state machine sketch (in the case of state machines that are required to be synthesized). Note that the state machines are required to be *finite state*, *i.e.*, they do not have any state variables and messages do not have payloads, as defined in Chapter 3. As in Chapter 3, we will refer to finite state machines and

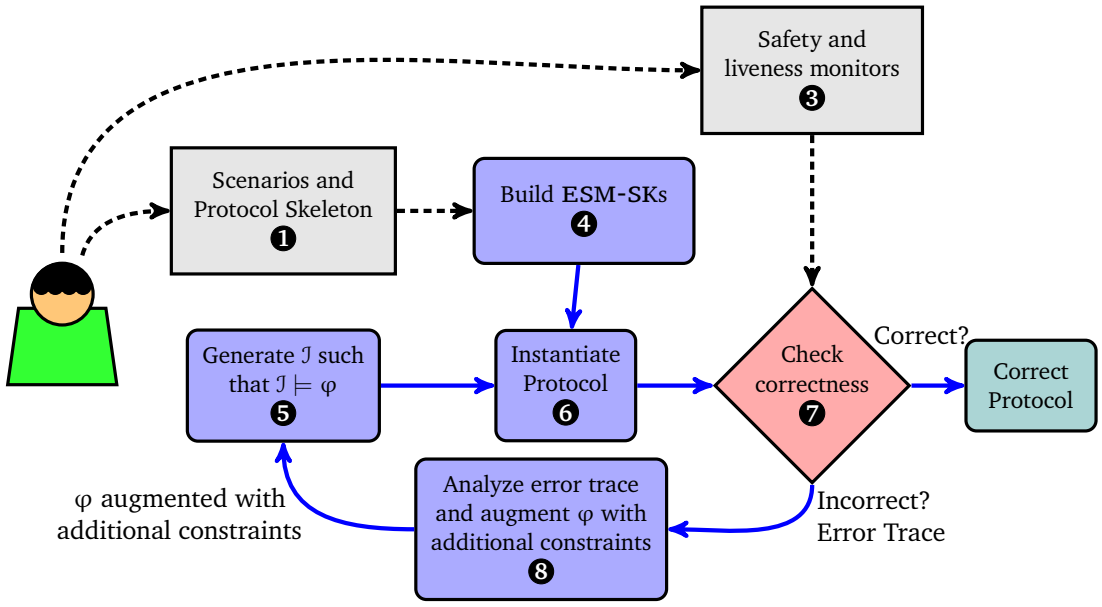
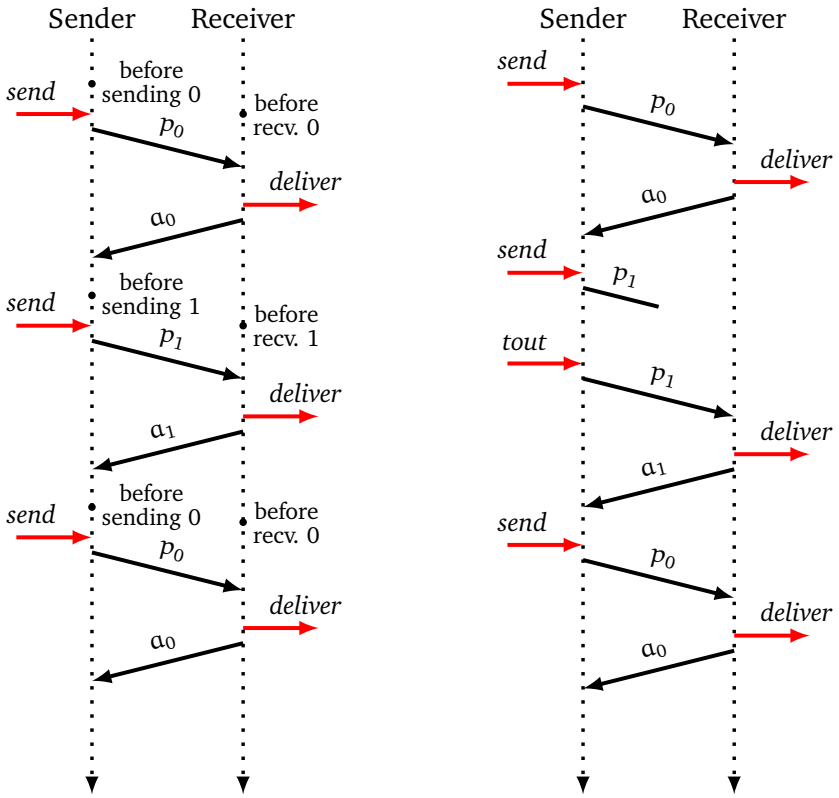


Figure 7.1: Algorithm for Synthesizing Finite-state Protocols from Scenarios

finite state machine sketches as FSMs and FSM-SKs respectively. The tool builds the FSM-SK of the full protocol using these scenarios. Given that the state machines here are finite-state and have no state variables, the interpretation $\mathcal{I}$ that is to be generated is thus a set of Boolean valued functions of the form $g : L \times \Sigma \cup \{\epsilon\} \times L \rightarrow \mathbb{B}$, which determines whether a transition to a location $l' \in L$, emitting (receiving) a message $m \in \Sigma \cup \{\epsilon\}$ is allowed when the ESM-SK is in location $l \in L$, where L is the set of locations of the ESM-SK under consideration. We use an Integer Linear Program (ILP) solver to generate this interpretation. The set of constraints φ can therefore be considered an integer linear program. The resulting protocol is then checked for correctness against the safety and liveness properties specified by the programmer. If the protocol is correct, then the algorithm terminates. Otherwise, the protocol automatically analyzes an error trace and augments the ILP φ with additional constraints which rule out at least this erroneous execution from future solutions.

We describe the two key parts of this algorithm in rest of this section. Section 7.2 introduces the notion of a *scenario* and describes how a set of scenarios is translated into a set of FSM-SKs and sets up the synthesis problem as one of completing this set of FSM-SKs. Section 7.3 describes a CEGIS based algorithm to solve the completion problem, and how error traces are analyzed to augment the ILP φ with additional constraints. Finally, Section 7.4 presents our experience on using this methodology to specify the alternating bit protocol [KR09], a cache



(a) Scenario 1 (b) Scenario 2

Figure 7.2: The first two scenarios for the Alternating-bit Protocol. The arrows colored red indicate the events involving the environment. The first scenario describes the normal operation of the protocol without any timeouts or lost messages. The second scenario describes the behavior when a packet loss occurs.

coherence protocol and a protocol for solving the distributed consensus problem using atomic registers.

7.2 Scenarios to FSM-SKs

A scenario is a sample execution trace of a protocol, which shows the exchange of messages that occur in the execution among the state machines that make up the protocol, with the passage of time. Abstractly, we can view a scenario as describing partial order on the events (sending and receiving messages) that occur across state machines in an execution. We will illustrate the use and semantics of scenarios using the example scenarios that describe the behavior of the Alternating-bit Protocol (ABP), which is a fundamental protocol in computer networking,

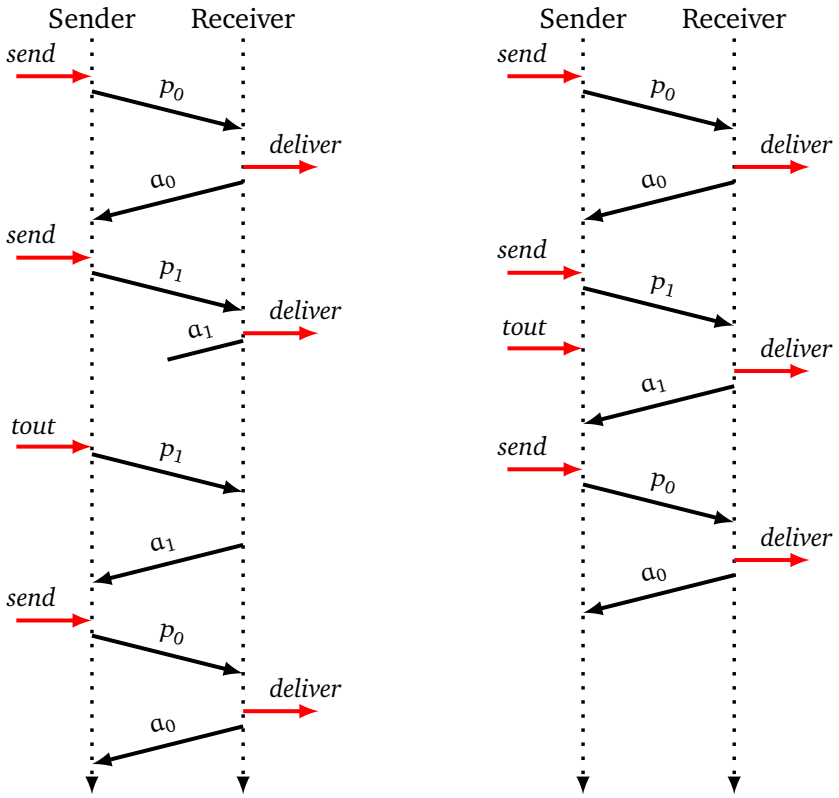


Figure 7.3: The two remaining scenarios for the Alternating-bit Protocol. Again, the arrows colored red indicate events involving the environment. The event labeled *tout* indicates a timeout event. The third scenario depicts the behavior when an acknowledgment is lost, and the final scenario describes the behavior on a premature timeout or a packet duplication

and is used for the reliable transmission of data across a channel which is *unreliable*. In this dissertation, we assume that an unreliable channel is capable of losing packets or messages, as well as duplicating them.

The Alternating-bit protocol consists of the state machines named Sender and Receiver, with a pair of ordered, duplicating and lossy channels between them: the Forward channel, relaying data packets (labeled p_0 and p_1) from the Sender to the Receiver, and the Backward channel, relaying acknowledgment packets (labeled a_0 and a_1 corresponding to the data packets p_0 and p_1) from the Receiver to the Sender. The goal of the protocol is to ensure reliable packet delivery despite the possibility that the channels may non-deterministically drop packets or duplicate them. This requirement is expressed by the liveness monitors provided by the programmer, which are not shown here.

We will use the scenarios shown in Figures 7.2 and 7.3 to describe the behavior of the ABP protocol. They come from a textbook on computer networking [KR09]. The first scenario describes the behavior of the protocol when no packets or acknowledgments are lost or duplicated. The second and the third scenarios correspond to the expected behaviors of the protocol in the event of the loss of a packet and in the event of the loss of an acknowledgment respectively. Finally, the fourth scenario describes the behavior of ABP on premature timeouts and/or packet duplication. Note that scenarios may be annotated with labels. This is shown in Figure 7.2(a), where the labels "before sending 0" and "before recv. 0" are used to indicate that the states of the Sender and the Receiver automata at two different points of the scenario are the same. These are used in the construction of automata from the scenarios as we will describe shortly. Labels can also be used to indicate that two states of an automaton are the same even across different scenarios. Furthermore, labels are essential for specifying recurring behaviors in scenarios and the structure of the incomplete state machine constructed depends on the number and positions of labels used. Also, note that these scenarios omit the environment state machines for simplicity. In particular the state machines corresponding to the channels are omitted, however, we will use a primed version of a message when referencing it on the state machine that receives it.

The idea for transforming scenarios into state machines is simple. First, for every "lane" in a given scenario, we identify the corresponding (complete or incomplete) state machine in the overall system. For example, in each scenario shown in Figures 7.2 and 7.3, the left-most lane corresponds to ABP Sender and the right-most lane to ABP Receiver.

Second, for every state machine P in the protocol whose behavior needs to be synthesized, we generate an incomplete state machine (or FSM-SK) A_P as follows. For every message *history* ρ (ρ is a finite sequence of messages received or sent by the state machine) specified in some scenario in the lane for P, we create a location s_ρ in A_P. If $\rho' = \rho \cdot x$ is an extension of history ρ by one message x, then there is a transition $s_\rho \xrightarrow{x} s_{\rho'}$ in A_P. At this point, we check that the inputs and outputs of A_P are included in the interface of P in the protocol skeleton and that A_P is deterministic.

Third, we merge states which have the same label. Merging occurs for states of a single scenario as well as across multiple ones if the same label is used in different scenarios. If consistent labels are given to the initial and final positions in all lanes of the scenarios the resulting incomplete automata could have cyclic behavior.

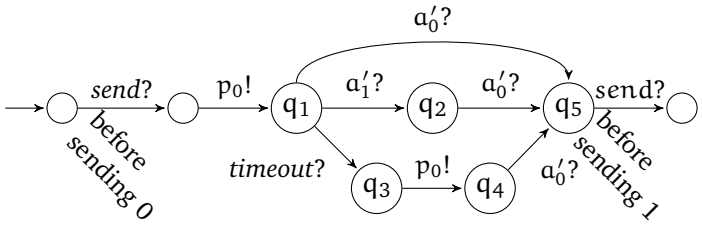


Figure 7.4: FSM-SK for the ABP Sender from all scenarios of Figures 7.2 and 7.3 and their symmetric versions after merging labeled states. (Only one half of the FSM-SK is shown, the rest is the symmetric case for packet p_1)

Finally, symmetric versions of scenarios are inferred from the given set of scenarios. For example, all the ABP scenarios express valid behaviors if p_0 and a_0 messages are consistently replaced with p_1 and a_1 messages respectively and vice-versa. Thus, the framework allows for scenarios to be characterized as symmetric.

As an example, the resulting FSM-SK for ABP Sender after applying the steps described above, given the scenarios shown in Figures 7.2 and 7.3 is shown in Figure 7.4. Note that the primed messages correspond to the unprimed messages of the same name which have been transmitted through a channel. Essentially, if a channel state machine receives a message p, then it outputs a message p' on its other end-point. This is needed because the output alphabets of the state machines in a protocol are required to be pairwise disjoint for the composition to be defined, as explained in Section 2.2. Once we have transformed the input scenarios into FSM-SKs, the problem is now one of protocol completion, as formalized in Section 2.3. The completion in this case, is to add appropriate transitions the FSM-SKs, which we now describe.

7.3 Completion of FSM-SKs

Given a set of incomplete finite-state automata or finite-state ESM-SKs, we associate a Boolean variable with every candidate transition that can be added to the individual incomplete automata. In other words, For every incomplete automaton A, and for every triple $\langle l, m, l' \rangle$, where $l, l' \in L$, the set of locations of A, $m \in I \cup O \cup \{\epsilon\}$, I and O are the input output alphabets of A, we associate a Boolean variable $t_{lml'}$, which indicates whether a transition from l to l' is permitted while receiving (transmitting) the message m. The completion task is to find a valuation for these Boolean variables, such that the resulting protocol (which is formed by composing the completed ESM-SKs with the environment automata) is (1) deterministic, (2)

deadlock-free, and (3) satisfies the safety and liveness monitors specified by the programmer. In the remainder of this section we will use the names t_i to refer both to candidate transitions that can be added to the automata as well as the Boolean variables corresponding to the transitions.

7.3.1 State Coverage

Note that the number of states in the incomplete automata are influenced by the scenarios used to construct them as well as the labels used in the scenarios. We have just set up the synthesis problem as a *completion* problem which only involves adding transitions to incomplete automata. As such, for the synthesis step to be successful, it is necessary that there exist a correct implementation of the protocol such that (1) It uses only the set of locations represented in the scenarios, and (2) Every provided scenario is an actual execution of such a correct implementation. If this is the case, then we say that the scenarios provide adequate *state coverage* for the synthesis to succeed.

7.3.2 Analysis of Counterexample Traces

To solve the finite-state protocol completion problem, we maintain a set of constraints φ on the transition variables t_i defined in Section 7.2. φ is initialized with determinism and deadlock constraints. The first enforce that the protocol automata are deterministic. For the second, we explore the reachable state space of the product of the environment and incomplete automata; for every deadlocked state, we add constraints that guarantee that at least one transition will be enabled out of that state.

The algorithm then works iteratively as follows. At the beginning of every iteration, a constraint solver — an ILP solver in our implementation — produces an assignment to the transition variables such the assignment satisfies the constraints φ. If the constraints are unsatisfiable, the algorithm concludes that no solution is possible and terminates. Otherwise, we translate the assignment to a set of transitions T, such that for every transition variable that the assignment sets to true, the corresponding transition is added to the appropriate incomplete state machine. Let the current set of transitions added, across all incomplete state machines be $T = \{t_1, \ldots, t_n\}$. We instantiate the protocol with transitions from T added to the appropriate incomplete state machines, form their product with the environment automata, and check for the absence of deadlocks, safety, and liveness violations using a model checker. The following cases are possible:

1. No violations are found. In this case, T is a correct completion, and the algorithm terminates.
2. A safety violation is found. This case means that the candidate solution T is incorrect. Moreover, any candidate T' obtained by adding extra transitions to T, i.e., $T' \supseteq T$, will also be incorrect, because adding extra local transitions can only add, but not remove, global transitions. This in turn implies that any reachable error state with T will also be a reachable error state with T', so any safety violation with T will also be a safety violation with T'. To enforce that no super-set of T is included in any future candidate set, we add the formula $\neg(t_1 \wedge t_2 \wedge \cdots \wedge t_n)$ to the constraint set.
3. A liveness violation is found. This case also means that the candidate solution T is incorrect. A liveness violation, corresponds to a fair infinite accepting run, represented by a reachable cycle, such that the run causes a liveness monitor to reach an accepting state infinitely often. Although adding more transitions cannot eliminate the cycle, it is possible that additional transitions can render a fair run unfair: if a particular output $o \in O_f$ was not enabled in the cycle, then adding local transitions can cause o to become enabled. Let $T' = \{t'_1, \ldots, t'_m\}$ be the set of transitions that, if added, would make the infinite run unfair.[11] We add as a constraint the formula $\neg(t_1 \wedge t_2 \wedge \cdots \wedge t_n) \vee (t'_1 \vee t'_2 \vee \cdots \vee t'_m)$. The constraint guarantees that in all future candidate sets, the cycle will be unreachable, broken, or not fair.
4. A deadlock state is found. In this case as well, T is incorrect, but could potentially be made correct by adding more transitions. Let $T' = \{t'_1, \ldots, t'_m\}$ be the set of candidate transitions such that, if any transition in T' is added, a transition is enabled out of the deadlock state. We add the constraint $(t_1 \wedge \cdots \wedge t_n) \rightarrow (t'_1 \vee \cdots \vee t'_m)$.

In every iteration, either a correct completion is found or the search space is pruned. We use an ILP solver to generate candidate sets from the constraints with an objective function that minimizes the size of the candidate set. In that way, in each iteration, we examine the smallest set of transitions that satisfies the constraints. This keeps the size of the product of the automata small and allows for faster checking of the properties.

We employ the following heuristic to prune the search space faster. Assume that a candidate set $T = \{t_1, \ldots, t_n\}$ is tested in an iteration of the algorithm and a safety violation is discovered. As described so far, the algorithm will remove all super-sets of T from the search space by

[11] For simplicity, we assume that process automata only communicate with environment automata. The constraint for the general case is more complicated but conceptually similar.

adding the constraint $\neg(t_1 \wedge \cdots \wedge t_n)$. However, if the safety violation is reachable by using only a subset of T, T″, then it is safe to also remove all super-sets of T″ from the search space. Ideally, one would find all minimal subsets of T that alone can lead to a violation and remove all super-sets of them. We approximate this by finding a minimal path to a safety violation using breadth-first search. If the path contains a subset of the transitions in T, we remove all super-sets of that subset from the search space.

7.3.3 Complexity of the FSM-SK Completion Problem

Theorem 3. *The* FSM-SK *completion problem as defined at the beginning of Section 7.3 is* PSPACE-complete.

Proof. It is easy to see that the problem is in NPSPACE. We can *guess* a completion; the space of all possible completions for each FSM-SK is bounded by $|L|^2 \times |\Sigma|$, where L is the set of locations and Σ is the message alphabet, and is thus polynomial in the size of the input. Once a completion has been guess, checking for determinism can also be accomplished in polynomial time. Finally, checking if the completion is *correct* is tantamount to LTL model checking, which is known to be PSPACE-complete [SC85]. From Savitch's theorem, we know that NPSPACE = PSPACE. Thus the FSM-SK completion problem is in PSPACE.

To prove hardness, we observe that in the special case where the FSM-SKs in the protocol have the following property: Adding *any* transition to *any* FSM-SK in the protocol results in the FSM-SK being non-deterministic. In this case, there is only one possible "completion": which is to not add any additional transitions. Determining whether this sole completion is correct is again tantamount to LTL model checking, which we know to be PSPACE-complete. This completes the proof that the FSM-SK completion problem is PSPACE-complete. □

7.4 Experimental Evaluation

In this section we evaluate the effectiveness of scenarios and our methodology for specifying finite-state protocols. We use three benchmarks: the ABP protocol, a cache coherence protocol, and a consensus protocol. We first check manually whether the corresponding scenarios provide sufficient state coverage to be able to synthesize a correct implementation. We then evaluate our synthesis algorithm on those benchmarks and investigate the effectiveness of scenarios in reducing the empirical complexity of the automata completion problem. Lastly, we discuss the interaction between the number of scenarios used to construct the initial incomplete

Benchmark	time (s)	# iterations	# candidate transitions
ABP1	2.8	44	84
ABP2	9.9	87	172
ABP1-4	11.5	59	240
ABPcolored1	63.8	197	260
ABPcolored2	168.9	273	652
ABPcolored1-4	409.4	293	1012
VI-no-data	28.6	208	1170
VI	183.7	215	4538
Consensus-fail	0.3	5	264
Consensus-success	13.8	162	112
Consensus-success+1	21.4	163	216
Consensus-no-test-and-set	11.2	156	88

Table 7.1: Summary of experimental results for finite-state protocol synthesis from scenarios.

automata and the number of requirements that are necessary to synthesize a correct protocol. A quantitative summary of our experiments can be found in Table 7.1. Each row corresponds to a combination of benchmark and set of input scenarios used for that benchmark, column "time" shows the total time that the synthesis algorithm took to find a correct completion, column "# iterations" shows the number of iterations of the algorithm, i.e., the number of candidate sets of transitions tested, and "# candidate transitions" is the total number of candidate transitions for all process automata. Note that this last number, n, represents individual local transitions and not number of candidate completions. The size of the space of all possible completions is the number of subsets of the set of candidate transitions, i.e., 2^n.

7.4.1 Alternating-bit Protocol

We have already described the working of this protocol using scenarios in Section 7.2. We use different sets of input scenarios to create three versions of this benchmark. ABP1 used only the first scenario shown in Figure 7.2 to construct the incomplete automata, ABP2 used only the second scenario, while ABP1-4 used all four scenarios. Although the text-book presentation uses four scenarios to describe the protocol, each of the these subsets of scenarios provided the state coverage necessary for our algorithm to synthesize a correct and complete protocol.

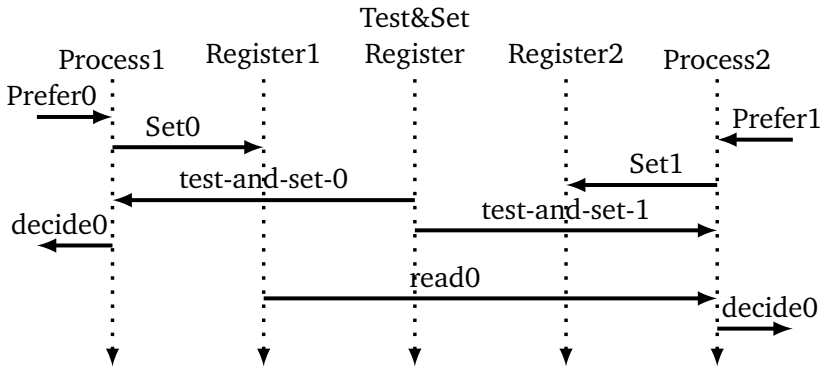


Figure 7.5: Scenario for the consensus protocol.

We also constructed a variant of the Alternating-bit protocol that also models the ability of the clients to send different *payloads* in the message packets. In the protocol described in Section 7.2, the implicit assumption was that the payload was unique and irrelevant. In the experiments ABPcolored1, ABPcolored2, and ABPcolored1-4, there are two "colors" of messages that can be sent and received. The different "colors" essentially represent the distinct values of data that the client of the sender automaton might wish to send to be delivered to the corresponding client of the receiver automaton using the Alternating-bit protocol.

7.4.2 The VI Cache Coherence Protocol

We have briefly mentioned the VI cache coherence protocol in Section 4.4. Here, we consider a finite-state version of the VI protocol, whose behavior is as described in Figure 1.3, in Chapter 1. The finite-state version of the VI protocol considered here is very similar to the version considered in Chapter 3.

We examine two variations of the VI protocol: one where there is a unique value for the data, in which case the protocol reduces to a distributed locking protocol (VI-no-data), and one where the data can take values 0 or 1, which captures the essence of the VI cache coherence protocol (VI) and ensures that the resulting protocol actually satisfies the coherence invariant as well, in addition to the liveness properties satisfied by the version of the protocol which assumes a unique data value.

7.4.3 The Consensus Protocol

In this problem we specify a protocol that describes how two processes can reach consensus on one value. Each process chooses initially a preferred value and then they coordinate using

shared memory to decide which of the two values to choose. The properties that the protocol has to satisfy are agreement (the two decisions must be the same), validity (the common decision must equal one of the preferred values), and wait-freedom (at any point, if only one process makes progress it will be able to make a decision). It has been shown that wait-freedom can be achieved only if a test-and-set register is used. The test-and-set register allows a process to write a value to it and read its previous value, with both steps occurring as an atomic operation.

Figure 7.5 shows the single scenario used for the consensus protocol. Both processes begin by non-deterministically choosing a value, messages "Prefer0" and "Prefer1", then write their choices in shared registers, "Register1" and "Register2", and then compete on setting the common test-and-set register which is initialized with 0. In this case, Process1 succeeds, the return value of the test-and-set operation is 0, and Process1 decides on its preferred value with message "decide0". On the other hand, Process2 fails, the test-and-set register returns 1, and Process2 reads the value chosen by Process1, and decides on that with messages "read0" and "decide0".

We first attempt to synthesize the protocol starting from the incomplete automata constructed from the "success path", i.e., only the lane for Process 1 in the scenario, and the "fail path", i.e., only the lane for Process 2 in the scenario. These two experiments correspond to rows "Consensus-success" and "Consensus-fail" of the Table 7.1. Finally, we implement a consensus protocol that does not use a test-and-set register, row "Consensus-no-test-and-set". Note that the protocol synthesized when a test-and-set register is not used is not wait-free.

7.4.4 Discussion

State Coverage

We observe that in all our experiments, except for "Consensus-success" and "Consensus-no-test-and-set", the states of the incomplete automata constructed by the scenarios cover all states of the protocols. In the "Consensus-success" experiment, the incomplete automaton is constructed using only the successful path of the protocol. A large part of the protocol's logic is missing from the input scenario, leaving the automaton with not enough states. The synthesis algorithm terminates and thus proves that no successful completion is possible. When we add an extra state in the incomplete automata without any edges to or from the rest of the states, the synthesis algorithm returns a completion that uses the extra state to implement the missing

behavior. Row "Consensus-success+1" corresponds to that experiment. This seems to indicate that apart from being a natural way to describe the behavior of distributed protocols, scenarios also contain enough information to mechanically fill in any missing detail.

Generalization and inference of unspecified behaviors

In all cases where the given scenarios covered all the states of the desired implementation the synthesis algorithm terminated with a correct completion. For the case of ABP with just one scenario specified, the algorithm successfully performs the generalization required to obtain a correct completion. The generalization performed is non-obvious: the correct protocol behaviors on packet loss, loss of acknowledgments and message duplication are inferred, even though the scenario does not describe what needs to happen in these situations. The incomplete automata constructed from the scenario describe only the protocol behavior over loss-less channels. The algorithms are guided solely by the liveness and safety specifications to infer the correct behavior. In contrast, when all four scenarios are used, the scenarios already contain information about the behavior of the protocol when a single packet loss or a single message duplication occurs. The algorithm thus needs to only generalize this behavior to handle an arbitrary number of losses and duplications.

The same is true about the generalizations made by the algorithm in the other benchmarks. Specifically, in the case of VI, the synthesis algorithm correctly infers that in a complete protocol write-back and invalidate messages should be treated in the same way both from the caches and from the directory. Note that this behavior cannot be inferred by looking at caches and directory independently: they both have to implement it for the result to be correct.

Interplay between scenarios and requirements

We observed that when fewer scenarios were used we needed to specify more properties — some of which were non-obvious — so that the algorithms could converge to a correct completion. For instance, when only one scenario was specified, we needed to include the liveness property that every deliver message was eventually followed by a send message. Owing to the structure of the incomplete automata, this property was not necessary to obtain a correct completion when all four scenarios were specified. Another property which was necessary to reject trivial completions when no scenarios were specified was that there has to be at least one send message in every run. Therefore, in some cases, using scenarios can compensate for the lack of detailed formal specifications.

Limitations and shortcomings

One primary limitation of the techniques described in this section is that they require the individual state machines to be finite-state, without any state variables. Although this is not a restriction from a theoretical perspective, because most interesting distributed protocols are indeed finite-state, it is often tedious to express a distributed protocol — even if it is indeed finite-state — without using any state variables. The resulting representation is often extremely low-level and unintuitive to human beings, who typically model such protocols as *extended state machines* with state variables and describe the evolution of the system by symbolic updates to these state variables, with the transitions themselves conditioned on symbolic guard expressions over these state variables. The work described in the next section attempts to remedy this shortcoming, and to solve the full protocol completion problem formulated in Section 2.3.

8

Completion of Distributed Protocols with Symmetry

In this section, we describe a fully automated solution to solve an unrestricted version of the problem formulated in Section 2.3, where the programmer provides the ESM-SKs for the protocol, and each ESM and ESM-SK may have an arbitrary number of state variables. Furthermore, no restrictions are applied on the kinds of payloads that messages can carry. This chapter is based on the work originally published in [ARS+15].

8.1 Overview of Symmetric Protocol Completion

Figure 8.1 shows the algorithm for symmetric protocol completion as an instantiation of the algorithmic scheme shown in Figure 3.2. The input here is a set of ESMs and ESM-SKs, so the block labeled ❹ — which compiles the input provided by the user into ESM-SKs— in Figure 3.2 is no longer necessary here. Based on the ESMs and ESM-SKs our algorithm generates the necessary determinism and symmetry constraints φ_0, which we require every interpretation to satisfy. The rest of the algorithm is conceptually similar to the algorithm described in Chapter 7. We model the unknown functions used in the guards and updates in the ESM-SKs as *uninterpreted* functions — rather than as Boolean variables as was the case in the algorithm described in Chapter 7 — and ask the SMT solver for an interpretation which satisfies the set of constraints maintained by the algorithm. The protocol is then instantiated with this interpretation and checked for correctness using a custom-built model-checker. Errors discovered during this check are automatically analyzed to obtain constraints on the uninterpreted functions which make at least the particular error trace in question infeasible in future iterations. This process is repeated until we obtain an interpretation that results in a correct, completed protocol.

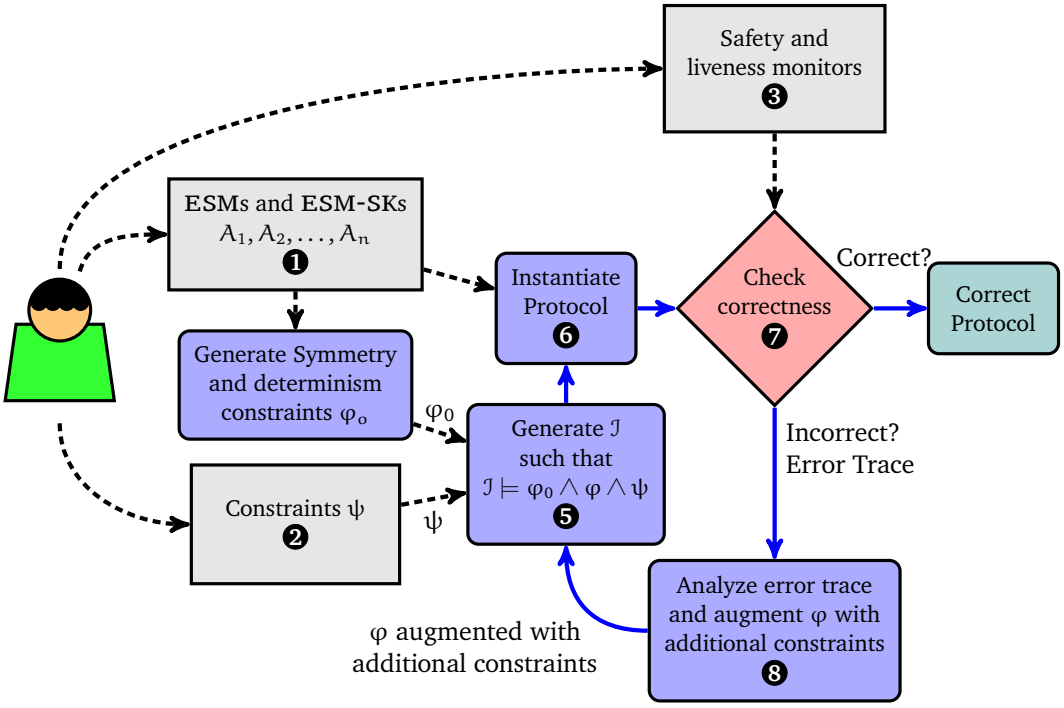


Figure 8.1: Overview of our approach for symmetric protocol completion.

The rest of this section is organized as follows: Section 8.2 describes the algorithm we have developed to solve the symmetric protocol completion problem in depth. Section 8.3 describes the model checking algorithms used check correctness of a proposed protocol, and also includes a description of a general purpose model checking framework which was built as a part of this effort. Section 8.4 describes the results of the using the techniques described in this chapter to synthesize a mutual exclusion protocol, Dijkstra's self stabilization protocol and several variants of a moderately sized cache coherence protocol.

8.2 Solving the Symmetric Protocol Completion Problem

We first describe how the initial constraints φ_0 shown in Figure 8.1 are generated. We then describe how counterexamples obtained from a suitable model checker, which supports symmetry and fine-grained fairness assumptions, are automatically analyzed to augment φ with additional constraints, which rule out at least the particular counterexample in question. We conclude this section with a short discussion on optimizations and heuristics which play a crucial role in getting the algorithm to scale to larger instances of the symmetric protocol completion problem.

8.2.1 Initial Constraints

The initial constraints φ_0 can be thought of being comprised of two disjoint sets. One set of constraints to ensure that any interpretation chosen renders the instantiation of the state machines deterministic; and another set of constraints to ensure that any interpretations that satisfies them will result in a protocol which satisfies the symmetry assumptions specified by the programmer. We will refer to these two sets of constraints as *determinism* constraints and *symmetry* constraints, respectively.

Determinism Constraints

Recall that an ESM-SK is deterministic under an interpretation $\mathcal{I}$ if and only if for every state (l, σ) if there are multiple transitions enabled at (l, σ), then they must be input transitions on distinct input channels. We constrain the interpretation $\mathcal{I}$ chosen at every step such that all ESM sketches in the protocol are deterministic under $\mathcal{I}$. Consider the ESM-SK for Peterson's algorithm shown in Figure 1.13(b). We have two transitions from the location L_3, with guards $g_{\text{crit}}(\mathtt{Pm}, \mathtt{Po}, \mathit{flag}, \mathit{turn})$ and $g_{\text{wait}}(\mathtt{Pm}, \mathtt{Po}, \mathit{flag}, \mathit{turn})$. We ensure that these expressions never evaluate to true simultaneously with the constraint $\neg\exists v_1 v_2 v_3 v_4\, (g_{\text{crit}}(v_1, v_2, v_3, v_4) \wedge g_{\text{wait}}(v_1, v_2, v_3, v_4))$. Although this is a quantified expression, which can be difficult for SMT solvers to solve, note that we only support finite types, whose domains are often quite small. So our tool unrolls the quantifiers and presents only quantifier-free formulas to the SMT solver.

Symmetry Constraints

Consider the case where the interpretation chosen for the guard g_{crit} shown in Figure 1.13(b), was such that $g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle \bot, \top \rangle, \mathtt{P0}) = \mathtt{true}$. Then, for the interpretation $\mathcal{I}$ to be symmetric with respect to the appropriate set of types for Peterson's algorithm, we require that $\mathcal{I}$ is such that $g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \bot \rangle, \mathtt{P1}) = \mathtt{true}$ as well, because the latter expression is obtained by applying the permutation $\{\mathtt{P0} \mapsto \mathtt{P1}, \mathtt{P1} \mapsto \mathtt{P0}\}$ on the former expression. Note that the elements of the *flag* array in the preceding example were flipped, because *flag* is an array indexed by the symmetric type processid. In general, given a function $f \in U_i$, we enforce the constraint $\forall \pi \in \mathsf{perm}(\mathcal{T}).\, \forall d \in \mathsf{dom}(f).\, (f(\pi(d)) \equiv \pi(f(d)))$, where $\mathcal{T}$ is the set of all types as described in Section 2.2. As with determinism constraints, these quantified constraints are unrolled before they are presented to the SMT solver.

8.2.2 Analyzing Counterexample Traces

We now describe in detail how we perform the analysis of counterexamples returned by the model checker. Our implementation first composes the ESM sketches to form a *product* ESM-SK Π. It then compiles down this product ESM-SK Π into guarded commands. These guarded commands operate over a set of variables which include the state variables of every ESM and ESM-SK in the protocol, as well as a distinguished variable that tracks the location of each ESM or ESM-SK. The guards and updates of each guarded command are as defined in Section 2.2, and the updates include the update to the distinguished location variable for each ESM and ESM-SK as well. The guards and updates of the guarded commands are also transformed by the compiler to use select, store, project and update functions[12] for reads and updates of arrays and records respectively. Furthermore, repeated assignments to the same variable in a guarded command are coalesced into a single assignment. In effect, each variable (be it of a scalar type, an array type or a record type) has *at most one* assignment to it in the list of updates associated with each guarded command. These transformations on the guarded commands make it easier to compute the weakest preconditions of predicates with respect to the guarded commands, as we shall now explain.

Let the set of guarded commands be G, given a guarded command $\mathsf{cmd} \in \mathsf{G}$, we define $\mathsf{guard}(\mathsf{cmd})$ to be the guard of cmd and $\mathsf{update}(\mathsf{cmd})$ to be the list of coalesced updates of cmd. The weakest precondition of a predicate φ with respect to an assignment statement $\mathsf{stmt} \triangleq \mathsf{l} := e$ is defined as $\mathsf{wp}(\mathsf{stmt}, \varphi) \equiv \varphi[\mathsf{l} \mapsto e]$, where $\varphi[\mathsf{l} \mapsto e]$ is the expression obtained by replacing all instances of the sub-expression l in φ with the expression e. We extend the definition of the weakest precondition of a predicate φ with respect to a sequence of statements in the natural way. The weakest precondition of a predicate φ with respect to a guarded command cmd is defined is defined as $\mathsf{wpcmd}(\mathsf{cmd}, \varphi) \equiv \mathsf{guard}(\mathsf{cmd}) \rightarrow \mathsf{wp}(\mathsf{update}(\mathsf{cmd}), \varphi)$. In the rest of this section, we use the symbol $\top$ to refer to the Boolean constant `true` and the symbol $\bot$ to refer to `false`, respectively, for brevity and readability.

Analyzing Deadlocks

In Figure 1.13(b), consider the candidate interpretation where both g_{crit}, g_{wait} are set to be universally `false`. Two deadlock states are then reachable: $S_1 = ((L_3, L_3), \{\mathit{flag} \mapsto \langle \top, \top \rangle, \mathit{turn} \mapsto$

[12] These are functions defined in the theory of arrays and records by the SMTLIB2 standard. For details, see `http://smt-lib.org/`.

$\mathtt{P1}\}$ and $S_2 = ((L_3, L_3), \{\mathit{flag} \mapsto \langle \top, \top \rangle, \mathit{turn} \mapsto \mathtt{P0}\}$. We strengthen φ by asserting that these deadlocks do not occur in future interpretations: either S_1 is unreachable, or the system can make a transition from S_1 (and similarly for S_2). In this example, the reachability of both deadlock states is not dependent on the interpretation, *i.e.*, the execution that leads to the states does not exercise any unknown function, hence, we need to make sure that the states are not deadlocks. The possible transitions out of location (L_3, L_3) are the transitions from L_3 to L_3 (waiting transition) and from L_3 to L_4 (critical transition) for each of the two processes. In each deadlock state, at least one of the four guards has to be true. So in the case of the deadlock in state S_1, we add the following disjunction to the set of constraints:

$$g_{\text{wait}}(\mathtt{P0}, \mathtt{P1}, \langle \top, \top \rangle, \mathtt{P1}) \vee g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle \top, \top \rangle, \mathtt{P1}) \vee$$
$$g_{\text{wait}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \top \rangle, \mathtt{P1}) \vee g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \top \rangle, \mathtt{P1})$$

Similarly for the case of the deadlock in state S_2, we add the following disjunction to the set of constraints:

$$g_{\text{wait}}(\mathtt{P0}, \mathtt{P1}, \langle \top, \top \rangle, \mathtt{P0}) \vee g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle \top, \top \rangle, \mathtt{P0}) \vee$$
$$g_{\text{wait}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \top \rangle, \mathtt{P0}) \vee g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \top \rangle, \mathtt{P0})$$

The two disjunctions are added to the set of constraints, since any candidate interpretation has to satisfy them in order for the resulting product to be deadlock-free.

Analyzing Safety Violations

Consider now an erroneous interpretation where the critical transition guards are true for both processes when *turn* is $\mathtt{P0}$, that is: $g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle \top, \top \rangle, \mathtt{P0})$ and $g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \top \rangle, \mathtt{P0})$ are set to true. Under this interpretation the product can reach the error location (L_4, L_4). We perform a weakest precondition analysis on the corresponding execution to obtain a necessary condition under which the safety violation is possible. In this case, the execution crosses both critical transitions and the generated constraint is $\neg g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle \top, \top \rangle, \mathtt{P0}) \vee \neg g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \langle \top, \top \rangle, \mathtt{P0})$. Note that the constraints obtained from this analysis are necessary: the protocol under any interpretation that satisfies the negation of the constraints would exhibit the same safety violation.

More formally, given an error trace that is a non-repeating execution (*i.e.*, a witness for a safety violation or a deadlock) which consists of an initial state valuation σ_0, and a sequence

of guarded commands from G, say, $\mathsf{cmd}_1, \mathsf{cmd}_2, \ldots, \mathsf{cmd}_n$. Given a predicate Γ, we define $\mathsf{pre}_0(\Gamma) \equiv \Gamma$, and recursively define $\mathsf{pre}_i(\Gamma) \equiv \mathsf{wpcmd}(\mathsf{cmd}_{n-i-1}, \mathsf{pre}_{i\text{-}1}(\Gamma))$. Then, if the trace is a witness for a safety violation, we add the constraint $C \triangleq \mathsf{pre}_n(\Gamma)[\nu \mapsto \sigma_0(\nu)]$, for every variable ν in the system, to our set of constraints φ, where Γ is the invariant which was violated. We note that after simplifications, C will be a constraint that refers only to the unknown functions $f_u \in \mathfrak{U}$; also, all arguments to an unknown function f_u will be *concrete values* in C. In fact, C will not refer to *any* variable at all, once it has been appropriately simplified. Our prototype employs extensive simplifications at each step during the computation of weakest preconditions to ensure that the formulas do not grow to be unmanageably large.

On the other hand if the trace is a witness for a deadlock, we add the constraint $C \triangleq \mathsf{pre}_n\left(\bigvee_{\mathsf{cmd}\in G} \mathsf{guard}(\mathsf{cmd})\right)[\nu \mapsto \sigma_0(\nu)]$, for every variable ν in the system, to the set of constraints φ maintained by the algorithm. This constraint ensures that if this particular execution is ever permitted under an interpretation for the unknown functions $\mathcal{U}$ chosen in the future, then some guarded command is enabled at the end of the execution, under that interpretation, therefore no longer rendering the final state of the execution a deadlock.

Analyzing Liveness Violations

An interpretation that satisfies the constraints gathered above is one that, when *turn* is `P0`, enables both waiting transitions and disables the critical ones. Intuitively, under this interpretation, the two processes will not make progress if *turn* is `P0` when they reach L_3. The executions in which the processes are at L_3 and either P_0 or P_1 continuously take the waiting transition is an accepting one. As with safety violations, we eliminate liveness violations by adding constraints generated through weakest precondition analysis of the accepting executions. In this case, this results in two constraints: $\neg g_{\text{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top, \top\rangle, \mathtt{P0})$ and $\neg g_{\text{wait}}(\mathtt{P1}, \mathtt{P0}, \langle\top, \top\rangle, \mathtt{P0})$. However, in the presence of fairness assumptions, these constraints are too strong. This is because removing an execution that causes a fair liveness violation is not the only way to resolve it: another way is to make it unfair. Given the weak fairness assumption on the transitions on the $\text{critical}_{\mathtt{Pi}}$ channels, the correct constraint generated for the liveness violation of Process P_0 is: $\neg g_{\text{wait}}(\mathtt{P0}, \mathtt{P1}, \langle\top, \top\rangle, \mathtt{P0}) \vee g_{\text{crit}}(\mathtt{P0}, \mathtt{P1}, \langle\top, \top\rangle, \mathtt{P0}) \vee g_{\text{crit}}(\mathtt{P1}, \mathtt{P0}, \text{true}, \text{true}, \mathtt{P0})$, where the last two disjuncts render the accepting execution unfair.

To describe the process of analyzing liveness counterexamples more formally, we assume that infinite accepting executions are given as a pair of a finite stem execution of size n and a finite

cycle execution of size m. First, we describe the case where no fairness assumptions exist in the system. The constraint computed from an accepting execution asserts either that the sequence of transitions should not be enabled or that the state of the system at the beginning of the cycle should be not be the same as the state at the end. If the set of variables of Π is $\{v_1, \ldots, v_N\}$ we introduce symbolic constants $v'_1, \ldots, v'_N$ and set $\Gamma \equiv v_1 \neq v'_1 \vee v_2 \neq v'_2 \vee \cdots \vee v_N \neq v'_N$. We first compute $C = \mathsf{pre}_m(\Gamma)$ on the cycle execution and then substitute $v'_1, \ldots, v'_N$ for $v_1, \ldots, v_N$ in C: $C' = C[v_1 \mapsto v'_1, \ldots, v_N \mapsto v'_N]$. We then get the final constraint by computing $\mathsf{pre}_n(C')$ on the stem execution.

We now describe the case where strong fairness assumptions are present. Let $\mathcal{F}_s$ be the set of strong fairness assumptions and G be the union of all sets $F \in \mathcal{F}_s$ such that every guarded command in F is disabled *everywhere* in the cycle. We adapt the computation of pre_i in the cycle execution as follows: $\mathsf{pre}'_i(\Gamma) \equiv \mathsf{wpcmd}(\mathsf{cmd}_{n-i-1}, \mathsf{pre}_{i\text{-}1}(\Gamma) \vee \left(\bigvee_{\mathsf{cmd} \in G} \mathsf{guard}(\mathsf{cmd})\right))$. Enabling a command cmd in G at a step in the cycle execution has the effect of making the accepting cycle unfair: since cmd is never executed in the cycle, enforcing $\mathsf{guard}(\mathsf{cmd})$ makes cmd infinite often enabled but never taken.

The case where weak fairness requirements are present is similar: we set G to be the union of all the sets $F \in \mathcal{F}_w$, such that: (1) there exists at least one state in the cycle which has the property that *every* guarded command in F is disabled at that state, and (2) no guarded command in F is ever executed anywhere in the cycle, *i.e.*, there do not exist states s_1 and s_2 in the cycle such that s_2 can be reached from s_1 by executing some command in F. The rest of the process to obtain a constraint that ensures that the cycle is unfair is the same as for strong fairness assumptions.

8.2.3 Heuristics and Optimizations

We describe a few key optimizations and heuristics that we have applied in the model checking step, as well as in the constraints that we present to the SMT solver. We have empirically observed that these techniques improve the scalability and predictability of our technique.

Not all counterexamples are created equal

The constraint we get from a single counter-example trace is weaker when it exercises a large number of unknown functions. Consider, for example, a candidate interpretation for the incomplete Peterson's algorithm which, when *turn* $=$ `P0`, sets both waiting transi-

tion guards g_{wait} to true, and both critical transition guards g_{crit} to false. We have already seen that the product is not live under this interpretation. From the infinite execution leading up-to the location (L_3, L_3), and after which P_0 loops in L_3, we obtain the constraint[13] $\neg g_{\text{wait}}(\texttt{P0}, \texttt{P1}, \langle \top, \top \rangle, \texttt{P0})$. On the other hand, if we had considered the longer self-loop at (L_3, L_3), where P_0 and P_1 alternate in making waiting transitions, we would have obtained the weaker constraint $\neg g_{\text{wait}}(\texttt{P0}, \texttt{P1}, \langle \top, \top \rangle, \texttt{P0}) \vee \neg g_{\text{wait}}(\texttt{P1}, \texttt{P0}, \langle \top, \top \rangle, \texttt{P0})$. In general, erroneous traces which exercise fewer unknown functions have the potential to prune away a larger fraction of the search space and are therefore preferable over traces exercising a larger number of unknown functions.

In each iteration, the model checker discovers several erroneous states. In the event that the candidate interpretation chosen is blatantly incorrect, it is infeasible to analyze paths to all error states. A naïve solution would be to analyze paths to the first n errors states discovered (where n is configurable). But depending on the strategy used to explore the state space, a large fraction these errors could be similar[14], and would only provide us with rather weak, or even identical sets of constraints. On the other hand, exercising as many unknown functions as possible, along different paths, has the potential to provide stronger constraints on future interpretations. In summary, we bias the model checker to *cover* as many unknown functions as possible along different paths, such that along any given path, the number of unknown functions that are exercised is kept as small as possible.

Heuristics/Prioritization to guide the SMT solver

As mentioned earlier, we use an SMT solver to obtain interpretations for unknown functions, given a set of constraints. When this set is small, as is the case at the beginning of the algorithm, there exist many satisfying interpretations. At this point the interpretation chosen by the SMT solver can either lead the rest of the search down a "good" path, or lead it down a futile path. Therefore the run time of the synthesis algorithm can depend heavily on the interpretations returned by the SMT solver, which we consider a non-deterministic black box in our approach.

To reduce the influence of non-determinism of the SMT solver on the run time of our algorithm, we bias the solver towards specific forms of interpretations by asserting additional constraints. These constraints associate a *cost* with interpretations and require an interpretation with a given bound on the cost, which is relaxed whenever the SMT solver fails to find a solution.

[13] Ignoring fairness assumptions.
[14] We observed this phenomenon in our initial experiments.

We briefly describe the most important of the heuristics/prioritization techniques: (1) We minimize the number of points in the domain of an unknown guard function at which it evaluates to true. This results in minimally permissive guards. (2) Based on the observation that most variables are unchanged in a given transition, we prioritize interpretations where update functions leave the value of the variable unchanged, as far as possible. (3) Another possibility that we have explored is to try to minimize the number of arguments on which the value of an unknown function depends.

8.3 Model Checking

To effectively and repeatedly generate constraints to drive the synthesis loop, a model checker needs to: (a) support checking liveness properties, with algorithmic support for fine grained notions of strong and weak fairness, (b) dynamically prioritize certain paths over others, as explained in Section 8.2.3, and (c) exploit symmetries inherent in the model. The fine grained notions of fairness over sets of transitions, rather than bulk process fairness are crucial. For instance, in the case of unordered channel processes, we often require that no message be delayed indefinitely, which cannot be captured by enforcing fairness at the level of the entire process. The ability to prioritize certain paths over others is also crucial so that candidate interpretations are exercised to the extent possible in one model checking run. Finally, support for symmetry-based state space reductions can greatly speed up each model checking run.

Surprisingly, we found that none of the well-supported model checkers met all of our requirements. SPIN [Hol97] only supports weak process fairness at an algorithmic level and does not employ symmetry-based reductions. Our efforts to encode the necessary fine grained strong fairness requirements as LTL formulas in SPIN resulted in the Büchi monitor construction step either blowing up or generating extremely large monitor processes. Support for symmetry-based reductions is present in Murφ [ID96, Dil96], but it lacks support for liveness checking.[15] SMC [SGE00] is a model checker with support for symmetry reduction and strong and weak process fairness. Unfortunately, it is no longer maintained, and has very rudimentary counterexample generation capabilities. Finally, NuSMV [CCG+02] does not support symmetry reductions, but supports strong and weak process level fairness. But, due to bugs, we were unable to obtain counterexamples in some cases.

[15] There exists an unmaintained version of Murφ which does support checking of some restricted forms of LTL properties, but it only supports weak fairness.

We therefore implemented a model checker based on the ideas used in Murφ [Dil96] for symmetry reduction, and an adaptation of the techniques presented in [ES97] for checking liveness properties under fine grained fairness assumptions. At a high level, the model checking algorithm consists of the following steps: (1) construct the symmetry-reduced state graph of the model. (2) If the reachable state space does not contain a state where an invariant is violated, then construct the symmetry-reduced product graph, obtained by composing the model with the Büchi monitors representing the LTL requirements. (3) find accepting strongly connected components (SCCs) in the symmetry-reduced product graph. (3) delete unfair states from each SCC; repeat steps (3) and (4) until either a fair SCC is found or no more accepting SCCs remain. We now provide a brief description of the architecture of our model checking and synthesis framework, called KINARA[16] and of how each of these steps are implemented in KINARA.

8.3.1 Architecture of KINARA

Figure 8.2 depicts the high level architecture of the KINARA framework. KINARA is implemented as a C++ library, which can support multiple front-ends for describing models and requirements, including the Murφ language. The arrows in Figure 8.2 denote inter module dependencies. For example, the arrow from the module labeled "Low-level Model Representation" to the module labeled "Expression Representation" indicates that the model representation depends on the functionality provided by the module responsible for expression representation.

At the heart of KINARA is an extensible library for representing expressions. The expression module deals with the syntax of expressions, and also provides APIs to create types. The module also supports expressions involving array indexing, field references of a record. Expressions can also be quantified over values of a type.

The low-level model representation APIs provide constructs for describing the model as a set of guarded commands. The *guard* of each guarded command is a Boolean valued expression, and the updates are a sequence of simple assignments to lvalues. Because KINARA only supports *finite* types, we assume that loops are unrolled in the low-level model description. The low-level representation does not check that symmetry breaking constructs are not used. It assumes that a higher-level front-end handles these aspects. In fact, in the low-level representation, all *objects* that are parameterized by a set of symmetric types are assumed to have already

[16] KINARA is not another recursive acronym, or KINARA is not another reachability analyzer. KINARA is open source software and is publicly available at `https://github.com/abhishekudupa/kinara`

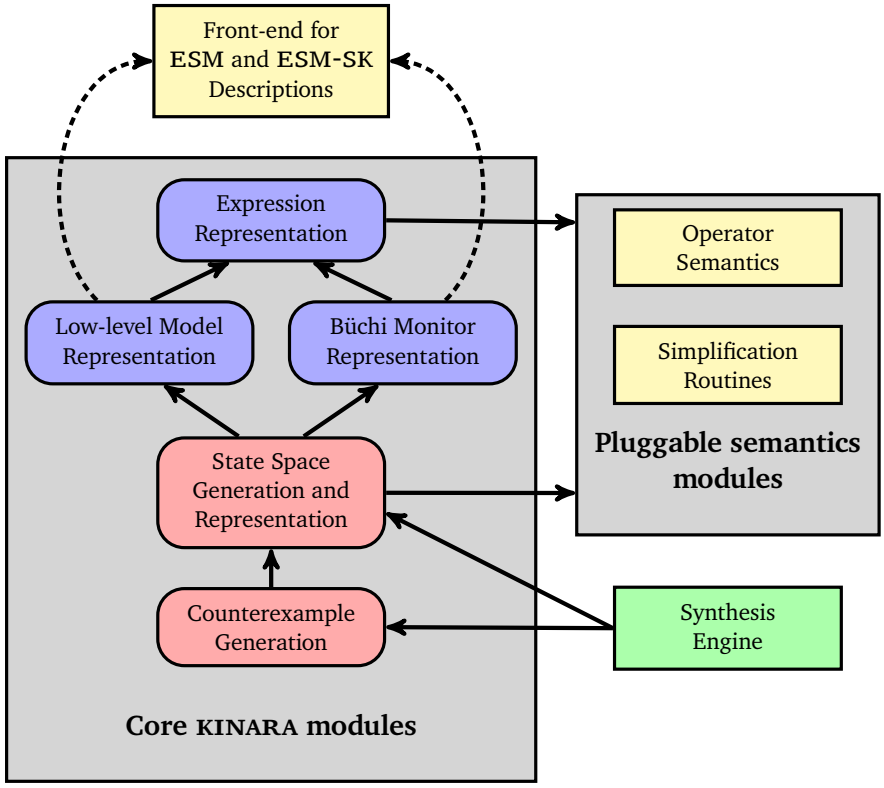


Figure 8.2: Architecture of the KINARA framework for model-checking and synthesis

been *unrolled*, with the exception of Büchi monitors. These are handled differently, as we explain in Section 8.3.3. For example, the low-level representation of the Peterson's mutual exclusion algorithm shown in Figure 1.13 will have *two* processes, one for each instantiation of the parameters. The values of the parameters `Pm` and `Po` in each of the machines will be substituted with concrete values from the set $\{\texttt{P0}, \texttt{P1}\}$ corresponding to the instantiation. The definitions of symmetry in terms of executions from Chapter 2, imply the following constraints on the low-level model representation:

- For every guarded command cmd, there must also exist the corresponding guarded command $\pi(\mathsf{cmd})$, for every $\pi \in \mathsf{perm}(\mathcal{T})$. Here $\pi(\mathsf{cmd})$ is obtained by syntactically permuting cmd by the permutation π.
- The transitions of every Büchi automaton must also be symmetric in the same manner. The notion of symmetry for Büchi automata will be described in greater detail in Section 8.3.3.
- For every fairness set $\mathsf{F} \in \mathcal{F}_w$, (respectively $\mathcal{F}_s$), of the form $\mathsf{F} \triangleq \{\mathsf{cmd}_1, \mathsf{cmd}_2, \ldots, \mathsf{cmd}_m\}$, it must be the case that for every permutation $\pi \in \mathsf{perm}(\mathcal{T})$ the fairness set $\mathsf{F}' \in \mathcal{F}_w$

(respectively $\mathsf{F}' \in \mathcal{F}_s$), where the fairness set F' is F permuted according to π and has the form $\mathsf{F}' \triangleq \{\pi(\mathsf{cmd}_1), \pi(\mathsf{cmd}_2), \ldots, \pi(\mathsf{cmd}_m)\}$.

Also, note that the low-level model representation represents the *product* of all the ESMs and ESM-SKs that form the protocol, where the product construction is as described in Chapter 2.

The framework also provides APIs to construct Büchi monitors. These are restricted to be state machines without state variables, but which can *inspect* the state of the model to determine the next state to transition to. These may be symmetric as well, but we defer a discussion on symmetric Büchi monitors to Section 8.3.3.

The module for state space generation provides mechanisms for efficiently representing the state space, where each state is represented as a sequence of bytes which encodes the valuation of variables in the state. Given a state which violates an invariant, or a fair, accepting strongly connected component, the counterexample generation module presents a counterexample as a simple path or a stem and a loop respectively. Given that the state space is represented in a *compressed* or symmetry-reduced manner, generating a usable counterexample is non-trivial, especially in the case of counterexamples which demonstrate the violation of a liveness requirement. Owing to this complexity, we have chosen to implement the counterexample generation as a separate module.

We note that the user is not restricted to specifying the model and requirements using the low-level model description APIs. Front-ends that support higher levels of abstraction can be implemented to translate models specified using these abstractions to the low-level model description. In our instantiation, we have implemented a front-end library that allows the specification of the model as communicating state machines and state machine sketches. This is indicated using dashed lines in Figure 8.2. The front-end which we have implemented ensures that symmetry breaking constructs are not used. This is done by enforcing the following rules at a syntactic level: (1) no reference to a concrete value of a symmetric type is allowed anywhere, and (2) the only operation defined on values of symmetric types is equality. In effect, this requires that the only way that the values of a symmetric type can be referred to is in the context of a *for each* construct, which is quantified over the values of the symmetric type. This ensures that there exists *one* instance of the quantified object — which could be an assignment, a transition, a message, a fairness assumption, or even an ESM or ESM-SK — for each value in the symmetric type. In addition, invariants and safety properties can only refer to values of a symmetric type using a universal or existential quantifier over the type. These syntactic

restrictions are sufficient to prevent a user from specifying invariants that are not symmetric over the symmetric model.

Recall that the KINARA expression module deals only with the syntax of expressions. The semantics of expressions are not a part of the core KINARA library. Instead, KINARA allows the semantics to be specified using pluggable C++ modules. In our instantiation, we implemented a module which described the semantics of the most commonly used operators in protocols, namely basic arithmetic operators including multiplication, modulus and division as well as conditional operators. The simplification routines are also implemented using pluggable modules to the core KINARA framework.

The synthesis engine uses the mechanisms provided by the state space representation routines to drive the model checking along paths which lead to more fruitful constraints as described in Section 8.2.3. It also leverages the counterexample generation routines to provide it with a usable counterexample whenever the model checking phase discovers a safety or liveness violation.

We now provide a brief description of the model checking algorithms, which are adaptations of the algorithms presented in earlier work [ID96, Dil96, ES97], are implemented in KINARA.

8.3.2 Construction of the Annotated Quotient Structure

Consider a model represented using the low-level representation of KINARA. The model refers to a set of variables V, each of which has a finite type drawn from the set of types $\mathcal{T}$, some of which may be symmetric types as defined in Chapter 2. The initial state of the model is described using the valuation σ_0[17] of the variables V. We denote by $\mathfrak{S}_V$, the set of *all* valuations over V. The evolution of the variables is described by a set of guarded commands $G \triangleq \{\mathsf{cmd}_1, \mathsf{cmd}_2, \ldots \mathsf{cmd}_n\}$. Given that all the variables range over finite domains, we can represent each valuation $\sigma \in \mathfrak{S}_V$ using a finite-length array of bytes $s \in \mathcal{S}$, where $\mathcal{S}$ represents the set of *all* arrays of bytes of the given finite length. In the context of the description of the model checking algorithms, we refer to both σ and s as a *state* of the model. Now, the reachable state space of the model can be described using a graph $M = (N_M, E_M)$, where $N_M \subseteq \mathcal{S}$ represents the vertices of the graph, and $E_M \subseteq N_M \times N_M \times \mathbb{N}$ represents the edges, where an edge $(s_1, s_2, i) \in E_M$ if and only if it is the case that the state executing the guarded command cmd_i in s_1, results in the state s_2. The *initial state* of M is s_0, the state corresponding to σ_0.

[17] We assume a single initial state. If multiple initial states are required, it can be emulated by having a (dummy) single initial state from which the system non-deterministically transitions to one of the actual initial states.

Thus M represents the state space of the model. We can ensure that no safety violation or deadlock is possible in the model, by merely inspecting the reachable subset of M. For liveness, we will need to construct the product of M with a Büchi monitor M_B and check for cycles in the reachable product state space.

In case of a symmetric model, storing the reachable state space of the model using the graph M is wasteful. Consider the set of system-wide permutations $\mathsf{perm}(\mathcal{T})$ over the set of types $\mathcal{T}$ that the model is defined over. Given that the model is symmetric, then for any edge $e = (s_1, s_2, i) \in E_M$, and for any $\pi \in \mathsf{perm}(\mathcal{T})$, we must also have that the edge $e' = (\pi(s_1), \pi(s_2), j) \in E_M$, where $\pi(s)$ denotes the state obtained by permuting s according to the permutation $\pi \in \mathsf{perm}(\mathcal{T})$, and j is the index of the guarded command obtained by permuting the guarded command with index i according to π. Note that the command cmd_j must belong to the set of guarded commands G, otherwise, the protocol is not symmetric. The set of system-wide permutations $\mathsf{perm}(\mathcal{T})$ thus induces an equivalence relation $\sim_{\mathcal{T}}$ over $\mathcal{S}$, where $s_1 \sim_{\mathcal{T}} s_2$ if and only if $\pi(s_1) \equiv s_2$ for some $\pi \in \mathsf{perm}(\mathcal{T})$. Because $\mathsf{perm}(\mathcal{T})$ forms a group with respect of composition of permutations, we have that $\sim_{\mathcal{T}}$ is reflexive, symmetric and transitive, and thus $\sim_{\mathcal{T}}$ is an equivalence relation. Furthermore, given that $\mathcal{S}$ is a set of finite-length byte arrays, we can define a total, lexicographic ordering $\prec$ on the elements of $\mathcal{S}$. Let us denote by $[s]$ the set of all states s' such that $s \sim_{\mathcal{T}} s'$. For every state $s \in \mathcal{S}$, we define the state $s_{can} \in [s]$, such that there does not exist a state $s' \in [s]$, such that $s' \neq s_{can}$, and $s' \prec s_{can}$. Thus s_{can} is a *representative* for the set of states $[s]$.

We can now represent M in a compressed form $\overline{M} = (\overline{N}_M, \overline{E}_M)$ as follows: The set of vertices $\overline{N}_M \triangleq \{s_{can} \mid s \in N_M\}$, *i.e.*, only the *representatives* from the equivalence classes in N_M are stored as vertices of $\overline{M}$. The set of edges $\overline{E}_M \subseteq \overline{N}_M \times \overline{N}_M \times \mathbb{N} \times \mathsf{perm}(\mathcal{T})$ is constructed such that $(s_1, s_2, i, \pi) \in \overline{E}_M$ if and only if executing the command cmd_i in state s_1, results in a state $\pi^{-1}(s_2)$, where π^{-1} denotes the inverse of the permutation π. The initial state of $\overline{M}$ is s_{0can}, *i.e.*, the representative of $s_0 \in N_M$. The structure $\overline{M}$ is called an Annotated Quotient Structure (AQS) [ID96, Dil96, ES97], and has the potential to be exponentially more compact that M, while retaining the same information.

While $\overline{M}$ has been described in terms of M, in practice, the structure M is never constructed. Instead, the AQS $\overline{M}$ is constructed on-the-fly by using a depth-first or breadth-first strategy, where each *new* state s encountered is first *canonicalized* to get the state s_{can}, and building the AQS using only these canonical representatives. The implementation in KINARA performs this

canonicalization using an exhaustive search over all the permutations in $\mathsf{perm}(\mathcal{T})$. We did not find the cost of this exhaustive canonicalization prohibitive for the protocols that we considered. However, if this proves to be prohibitively expensive, we note that the the canonicalization is implemented as a separate module in KINARA. Thus, heuristic canonicalization techniques which have been proposed earlier literature [ID96, Dil96, ES97] can be implemented in a relatively straight-forward manner, as additional canonicalization modules which can be plugged in as necessary.

Finally, we note that the construction of the AQS $\overline{M}$ is sufficient to verify safety properties. Assuming that the safety properties are symmetric as well, it has been shown in earlier work [ID96, Dil96, ES97] that $\overline{M}$ satisfies exactly the set of symmetric safety properties as M.

8.3.3 Construction of the Annotated Product Structure

The Annotated Product Structure (APS) is constructed for checking liveness properties under fairness assumptions. The core KINARA framework assumes that the representation of the Büchi automata is symmetric as well, *i.e.*, any given Büchi automaton is itself parameterized by zero or more symmetric types, depending on the properties that they check. For example, consider the Büchi automaton shown in Figure 1.13(c). This monitor is parameterized by a variable called `PID`, which can take values of type `processid`. One can imagine this parameterized Büchi automaton to be representing a *set* of symmetric automata, of size $|\mathsf{processid}|$, with one automaton for each value that the variable `PID` can take. Together, these automata check that *both* processes satisfy their respective (and symmetric) liveness requirements. Furthermore, the symmetric Büchi automata are assumed to correspond to the *negation* of the LTL property that the protocol is expected to satisfy. In other words, an execution that is *accepted* by a Büchi automaton is an execution that *violates* the corresponding liveness requirement.

The core KINARA framework also assumes that the fairness assumptions are symmetric, as described earlier in Section 8.3.1. The check that the Büchi automata as well as fairness assumptions are symmetric are handled by the front-end in our case.

Consider a Büchi automaton B over the set of locations L_B and whose transition relation is $R_B \subseteq L_B \times L_B \times \mathcal{S}$, with a set of *initial* locations $Q_0 \subseteq L_B$, and a set of *accepting* locations $Q_{acc} \subseteq L_B$. Suppose that B is parameterized by a set of symmetric types $\mathcal{T}_B \subseteq \mathcal{T}$, where $\mathcal{T}_B \equiv \{T_1, T_2, \ldots T_k\}$, then this essentially represents that there are $K_B = |T_1| \times |T_2| \times \cdots \times |T_k|$ symmetric instances of B, with one instance for each value in $I \triangleq T_1 \times T_2 \times \cdots \times T_k$.

The APS $\widehat{M}_B$ is defined as a graph with the set of vertices $\widehat{N}_{MB} \subseteq \overline{N}_M \times L_B \times I$, and the set of edges $\widehat{E}_{MB} \subseteq \widehat{N}_{MB} \times \widehat{N}_{MB} \times \mathbb{N} \times \mathsf{perm}(\mathcal{T})$. An edge $((s_1, q_1, i), (s_2, q_2, j), k, \pi) \in \widehat{E}_{MB}$ if and only if all of the following hold:

1. (s_1, s_2, k, π) is an edge in $\overline{M}$
2. $(q_1, q_2, s_1) \in R_B$
3. $\pi(i) \equiv j$, recall that i and j are tuples from $T_1 \times T_2 \times \cdots T_k$, and can thus be permuted.

The state (s_{0can}, q_0, i), for all $q_0 \in Q_0$ and for all $i \in I$ is an initial state of $\widehat{M}_B$. A vertex $(s, q, i) \in \widehat{N}_{MB}$ is called *green* if $q \in Q_{acc}$. Green vertices in $\widehat{M}_B$ will be used to characterize cycles in $\widehat{M}_B$ that lead to the Büchi monitor B visiting an accepting state infinitely often.

In the actual implementation, each product state simply retains a pointer to the corresponding AQS state. The state of the Büchi monitor is stored as a small width integer, and the values from I are again encoded as integers. This helps keep the size of the product structure manageable by not duplicating the states of the AQS $\overline{M}$. The implementation builds $\widehat{M}_B$ by considering only the reachable portion of $\overline{M}$ and executing an on-the-fly BFS construction which builds the APS while simultaneously constructing the product of $\overline{M}$ with B.

8.3.4 Checking for a Fair, Accepting Cycle

Consider the APS $\widehat{M}_B$ as described in Section 8.3.3. We denote an edge of the form $(\widehat{s}_1, \widehat{s}_2, k, \pi)$, where $\widehat{s}_1, \widehat{s}_2 \in \widehat{N}_{MB}$ as $\widehat{s}_1 \xrightarrow{k,\pi} \widehat{s}_2$. A *path* $\widehat{x}$ in $\widehat{M}_B$ is a finite sequence states such that every two adjacent states are related by an edge and is denoted as $\widehat{x} \triangleq \widehat{s}_0 \xrightarrow{k_1,\pi_1} \widehat{s}_2 \xrightarrow{k_2,\pi_2} \cdots \xrightarrow{k_n,\pi_n} \widehat{s}_n$. Given such a path $\widehat{x}$, we denote the composition of permutations along that path by $\pi_{\widehat{x}}$, *i.e.*, $\pi_{\widehat{x}} \equiv \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1$. Then, we have from earlier work [ES97] that an APS $\widehat{M}_B$ has a fair accepting cycle if and only if there exists a strongly connected sub-graph $\widehat{C}$ in $\widehat{M}_B$ which has the following properties:

1. $\widehat{C}$ contains at least one *green* state, in which case, we call $\widehat{C}$ itself as being *green*.
2. For every $F \in \mathcal{F}_w$, of the form $F \triangleq \{\mathsf{cmd}_1, \mathsf{cmd}_2, \ldots, \mathsf{cmd}_m\}$ either (1) there exists a state in $\widehat{C}$, such that every command in F is *disabled*, *i.e.*, the guard of every command in F evaluates to `false` on that state or (2) for every state $\widehat{s}$ where some $\mathsf{cmd}_i \in F$ is enabled, there exists a path $\widehat{x}$ lying entirely within $\widehat{C}$, beginning at $\widehat{s}$ and terminating at a state $\widehat{s}'$, such that a $\widehat{s}'$ is reached on $\widehat{x}$ by executing the command $\pi_{\widehat{x}}(\mathsf{cmd}_i)$. In other words, the path $\widehat{x}$ has the form $\widehat{x} \triangleq \widehat{s} \xrightarrow{k_1,\pi_1} \cdots \xrightarrow{k_j,\pi_j} \widehat{s}'$, where k_j is the index of the $\pi_{\widehat{x}}(\mathsf{cmd}_i)$. We will refer to this property as "Property(A)"

Algorithm 8.1: Algorithm to find a fair, *green* strongly connected subgraph

```
compute the strongly connected components of M̂_B
do
    deleted ← false
    foreach strongly connected component Ĉ which is green do
        if Ĉ satisfies Property(A) and Property(B) then
            return Ĉ
        if Ĉ does not satisfy Property(A), but satisfies Property(B) then
            continue
        if Ĉ does not satisfy Property(B) then
            foreach F ∈ F_s such that Property(B) is violated for F do
                delete from M̂_B all states where some command cmd ∈ F is enabled
                deleted ← true
while deleted is true
return ⊥
```

3. For every $\mathsf{F} \in \mathcal{F}_S$, of the form $\mathsf{F} \triangleq \{\mathsf{cmd}_1, \mathsf{cmd}_2, \ldots, \mathsf{cmd}_m\}$ either (1) No command in F is enabled in *any* state in $\widehat{C}$, or (2) for every state s such that some command $\mathsf{cmd}_i \in \mathsf{F}$ is enabled, there exists a path $\widehat{x}$ lying entirely within $\widehat{C}$, beginning at $\widehat{s}$ and terminating at a state $\widehat{s}'$ such that $\widehat{s}'$ is reached on $\widehat{x}$ by executing the command $\pi_{\widehat{x}}(\mathsf{cmd}_i)$. We will refer to this property as "Property(B)"

Intuitively, these two requirements track the permutations that occur along a path and ensure that the path satisfies all the fairness assumptions, despite the fact that the path involves permuations and could possibly be compactly encoding a large number of un-permuted paths.

Algorithm 8.1 shows how the existence of such a strongly connected subgraph. It begins by trying to find a maximal strongly connected subgraph which satisfied all the fairness assumptions. This can be done efficiently using Tarjan's algorithm for discovering strongly connected components in a graph [Tar72]. Suppose that a maximal strongly connected subgraph — which is the same as a strongly connected component — is not fair, due to some *strong* fairness assumption not being satisfied, the algorithm then decomposes the strongly connected subgraph by deleting some vertices, and restarts the process.

8.4 Experimental Evaluation

Having described the KINARA model checking and synthesis framework in some detail, we proceed to a description how well the framework performed on a few protocol synthesis tasks.

We combine the description of the protocol synthesis task with an explanation of how the KINARA prototype fared in the subsequent parts of this section.

8.4.1 Peterson's Mutual Exclusion Algorithm

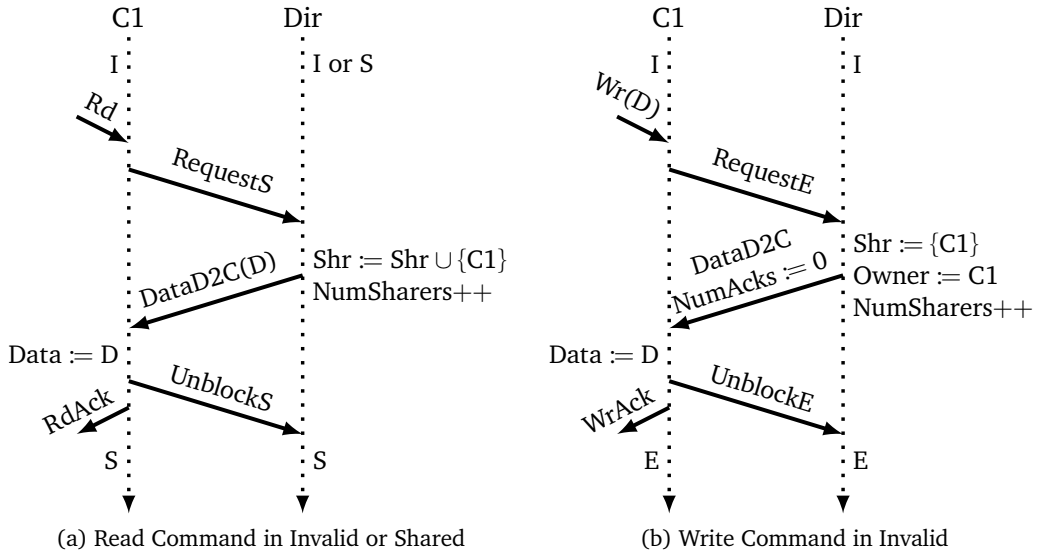
We evaluated the proposed method to synthesize Peterson's algorithm, which was described in Section 1.2.3. In addition to the missing guards g_{grit} and g_{wait}, we also replace the update expressions of flag[Pm] in the (L_1, L_2) and (L_4, L_1) transitions with unknown functions that depend on all state variables. In the initial constraints we require that $g_{\text{crit}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn}) \vee g_{\text{wait}}(\texttt{Pm}, \texttt{Po}, \mathit{flag}, \mathit{turn})$. The synthesis algorithm returns with an interpretation in less than a second. Upon submitting the interpretation to a SyGuS solver, to obtain symbolic representations of the interpretations assigned to the unknown functions, the synthesized symbolic expressions match the ones shown in Figure 1.13(b).

8.4.2 Self Stabilizing Systems

Our next case study is the synthesis of self-stabilizing systems [Dij74]. A distributed system is self-stabilizing if, starting from an arbitrary initial state, in each execution, the system eventually reaches a global *legitimate* state, and only legitimate states are ever visited after. We also require that every legitimate state be reachable from every other legitimate state. Consider N processes connected in a line. Each process maintains two Boolean state variables *x* and *up*. The processes are described using guarded commands of the form, "`if guard then update`". Whether a command is enabled is a function of the variable values *x* and *up* of the process itself, and those of its neighbors. We attempted to synthesize the guards and updates for the middle two processes of a four process system P_1, P_2, P_3, P_4. Specifically, the ESM-SK for P_2 and P_3 have two transitions, each with an unknown function as a guard and two unknown functions for updating its state variables. The guard is a function of x_{i-1}, x_i, x_{i+1}, up_{i-1}, up_i, up_{i+1}, and the updates of x_i and up_i are functions of x_i and up_i. We followed the definition in [GT14] and defined a state as being legitimate if exactly one guarded command is enabled globally. We also constrain the completions of P_2 and P_3 to be identical.

8.4.3 Cache Coherence Protocol

Recall that a cache coherence protocol ensures that the copies of shared data in the private caches of a multiprocessor system are kept up-to-date with the most recent update to that



(a) Read Command in Invalid or Shared

(b) Write Command in Invalid

Figure 8.3: Simple Cases for Read and Write Commands

shared data, by any other processor in the system. We describe the working of the German cache coherence protocol, which has often been used as a case study in model checking research [CMP04, TT08]. The protocol consists of a *Directory* process, n symmetric *Cache* processes and n symmetric *Environment* processes, one for each cache process. Each cache may be in the E,[18] S or I state, indicating read-write, read, and no permissions on the data respectively. All communication between the caches and the directory is non-blocking, and occurs over buffered, unordered communication channels.

The environment issues *read* and *write* commands to its cache. In response to a *read* command, the cache C sends a *RequestS* command to the directory. The directory sends C the most up-to-date copy of the data, after coordinating with other caches, grants read access to C, notes that C is a *sharer* of the data. In response to a *write* request from the environment, the cache C sends a *RequestE* command to the directory. The directory coordinates with every other cache C' that has read or write permissions to revoke their permissions, then grants C exclusive access to the data, and notes that C is the owner of the data.

We consider a more complex variant of the German cache coherence protocol to evaluate the techniques we have presented so far, which we refer to as German/MSI. The main differences from the base German protocol are: (1) Direct communication between caches is possible in

[18]Not to be confused with the E state in the MESI protocol described in Section 4.4.

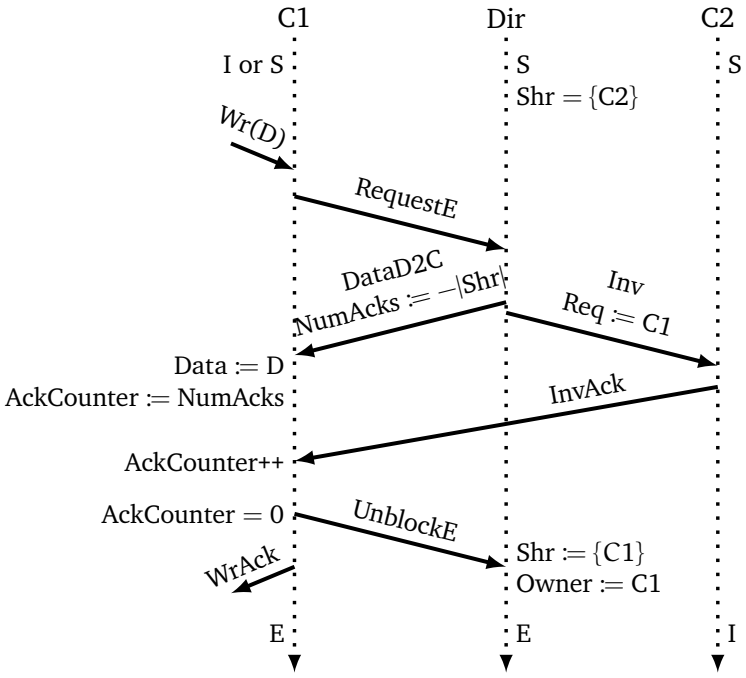


Figure 8.4: Write Command in Shared State

some cases, (2) A cache in the S state can silently relinquish its permissions, which can cause the directory to have out-of-date information about the caches which are in the S state. (3) A cache in the E state can coordinate with the directory to relinquish its read/write permissions over the block. The complete German/MSI protocol, modeled as communicating extended state machines, is fairly complex, with a symmetry-reduced state space of about 20,000 states when instantiated with two cache processes and about 450,000 states when instantiated with three cache processes.

We now describe the working of the protocol used in the experimental evaluation using scenarios which demonstrate the expected behavior of the caches and directory in response to various stimuli from their environments.

Simple Cases for Read and Write Commands

We first consider the case where a cache process receives a read or (resp. write) command from the environment, and no other cache in the system has exclusive permissions on data (resp. any permissions on the data).

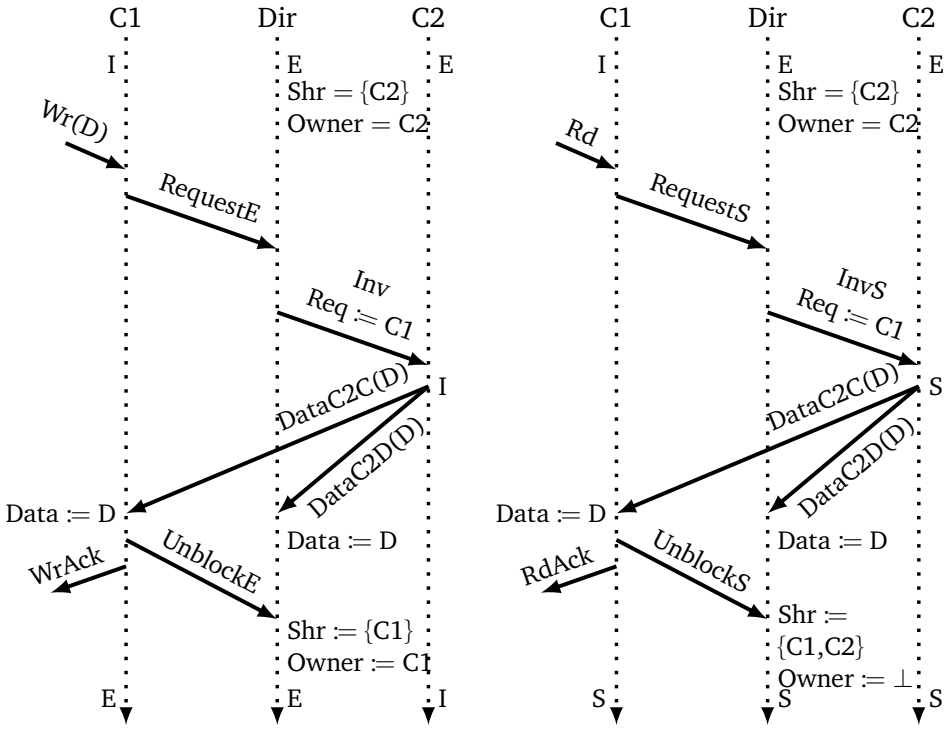
Figure 8.3a shows the actions performed by the various processes when a cache receives a *read* command from its environment. It sends a *RequestS* message to the directory. In this

particular scenario, the directory has recorded that all other caches are either in the I or S state, and proceeds to send the most up-to-date copy of the data in a *DataD2C* message. The cache then updates its local copy of the data, notifies its environment that the command has been processed and transitions to the S state. Figure 8.3b shows how a cache processes a *write* command from its environment which contains the new data value D to write. In this particular case, the directory knows that all other caches are in the I state and thus proceeds to acknowledge the *RequestE* message from the cache with a *DataD2C* message which also contains the number of acknowledgments the cache needs to wait for before gaining write permissions on the data. In this case, since all other caches are in the I state, the number of acknowledgments to wait for is zero. The cache therefore, immediately updates its local copy of the data with the new value D and notifies its environment that the command has been processed and transitions to the E state. The case where all the other caches are not in the I state will be described shortly, using another scenario.

Read and Write Commands which require Invalidations

On the other hand, Figure 8.4 depicts the scenario when a *write* command is received by a cache and some other cache is in the S state. In this case, the directory sends invalidations to all the caches in the S state, and sends a *DataD2C* message to the requesting cache with the *NumAcks* field set to the number of sharers, notifying the cache that it needs to wait for as many invalidate acknowledgments. The other caches directly communicate with the requesting cache by sending acknowledgment of the invalidation from the directory. Note that this is not part of the base German/MSI coherence protocol, where the directory collects acknowledgments instead. With the extension, the cache-to-cache communication reduces the amount of processing that needs to be done in the centralized directory, and also reduces the latency (in terms of number of message hops needed to service a request from the enviroment) for some requests.

Figure 8.5a describes the behavior of the protocol when a cache receives a *write* command and some other cache in the system is in the E state. The actions are similar to the case where some other cache is in the S state, except that the cache already in the E state directly sends its data to the requesting cache, as well as to the directory. And the requesting cache does not need to wait for any acknowledgments. Note that this is again an extension to the base German/MSI protocol, where the data is sent to only the directory, and the directory forwards



(a) Write Command in Exclusive State

(b) Read Command in Exclusive State

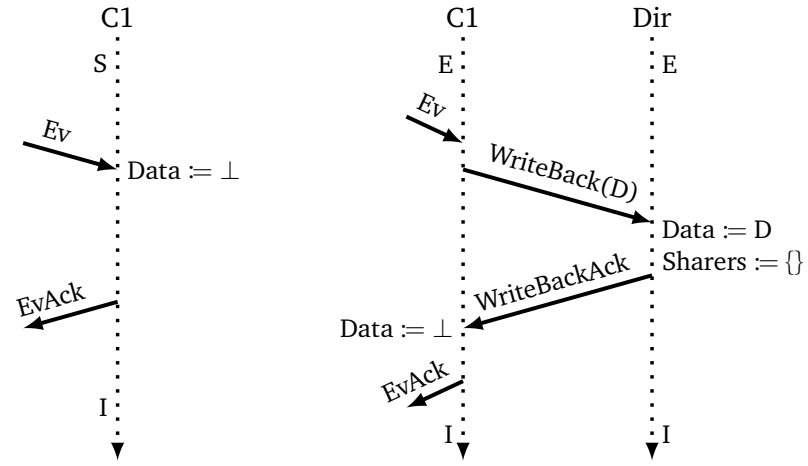
Figure 8.5: Commands in Exclusive State in the German/MSI Protocol

the data back to the requesting cache. Again, this extension reduces the amount of processing that needs to be handled at the centralized directory.

The scenario when a cache receives a *read* command from the environment when some other cache in the system is in the E state is shown in Figure 8.5b. As in the scenario shown in Figure 8.5a, the directory sends an invalidation to the cache in the E state, which in turn responds by sending the most up-to-date copy of the data to the directory as well as to the requesting cache. It then downgrades its permissions to the S state. Both the cache and the directory update their local copies of the data. The directory notes that the cache earlier in the E state is now in the S state and also adds the requesting cache to set of sharers.

Relinquishing Permissions (Evictions)

Figure 8.6a and Figure 8.6b describe the behavior of the protocol in the case where a cache wishes to relinquish its permissions. This is not a scenario that occurs in the base German/MSI protocol, but is necessary in a real-world coherence protocol, where a block of data that has been unused for some period of time may need to be evicted to make room for some other data.



(a) Evict in Shared State (b) Evict in Exclusive State

Figure 8.6: Evict Commands in the German/MSI protocol

This situation is depicted in Figure 8.6 by the receipt of an *Ev* command from the environment by the cache. In the event that the cache in the S state, it silently evicts the line, without notifying the directory. This can be done, only because the directory already has the most up-to-date copy of the data — recall that the S state only grants read permissions to the cache, hence it could not have modified the data. On the other hand if the cache is in the E state, then it needs to send the most up-to-date copy of the data to the directory. Therefore it sends a *WriteBack* message to the directory which contains the most up-to-date copy of the data. The directory then updates its local copy of the data with this copy and notes that all caches in the system are in the I state.

Corner-cases in the German/MSI Protocol

We now describe the corner-cases that could occur in the MSI/German protocol due to the asynchronous interleaving of the scenarios presented so far. Consider the case where cache C1 is in the I state. In contrast, the directory records that C1 is in state S and is a sharer, due to C1 having silently relinquished its read permissions at some point in the past, according the the scenario shown in Figure 8.6a. Now, both caches C1 and C2 receive *write* commands from their respective environments. Cache C2 sends a *RequestE* message to the directory, requesting exclusive write permissions. The directory, under the impression that C1 is in state S, sends an *Inv* message to it, informing it that C2 has requested exclusive access and C1 needs to acknowledge that it has relinquished permissions to C2. Concurrently, cache C1 sends a

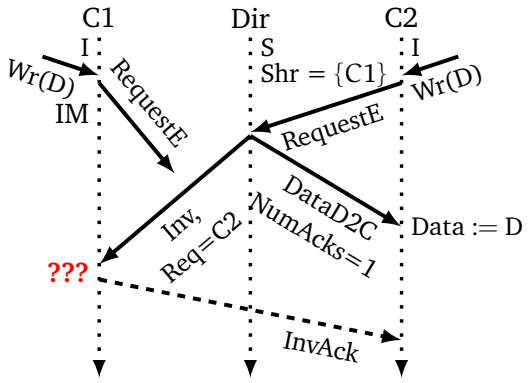


Figure 8.7: The racy scenario in the MSI/German Protocol

RequestE message to the directory requesting write permissions as well, which gets delayed. Subsequently, the cache C1 receives an invalidation when it is in the state IM, the behavior for which is not described by any of the scenarios provided by the programmer. The correct behavior for the cache in this situation (shown by dashed arrows), is to send an *InvAck* message to the cache C2. The guard, the state variable updates, as well as the location update is what we have left unspecified in the case of this particular scenario.

The MSI/German protocol as described in this section has four other corner-cases. Two of these are similar to the one shown in Figure 8.7, with the difference being that either the *RequestS* message is sent by C1 in response to a *Read* command from the environment, or that C1 begins in the S state, and sends a *RequestE* message in response to a *Write* command from the environment.

We now describe the last two corner-cases. These are depicted in Figure 8.8. The scenarios shown in Figures 8.6b and 8.5b interleave, to obtain the situation shown in Figure 8.8. The cache C1 having sent a *WriteBack* message to the directory is not expecting an *Inv* message. Similarly, the directory, having sent an *Inv* to cache C1 is not expecting a *WriteBack* message from it. The correct way for the processes to behave in this situation is show by dashed arrows in Figure 8.8. The cache behaves as if the *Inv* message was a *WritebackAck* message and notifies its environment of completion. The directory updates its local copy of the data with the one from the *WriteBack* message, and then sends this data over to the cache C2, informing it that it need not wait for any acknowledgment. After this point, both the cache and directory behaviors know how to interact with each other as shown in Figure 8.3a. For completeness, the way the scenario plays out is shown in Figure 8.8 as well, using dashed arrows.

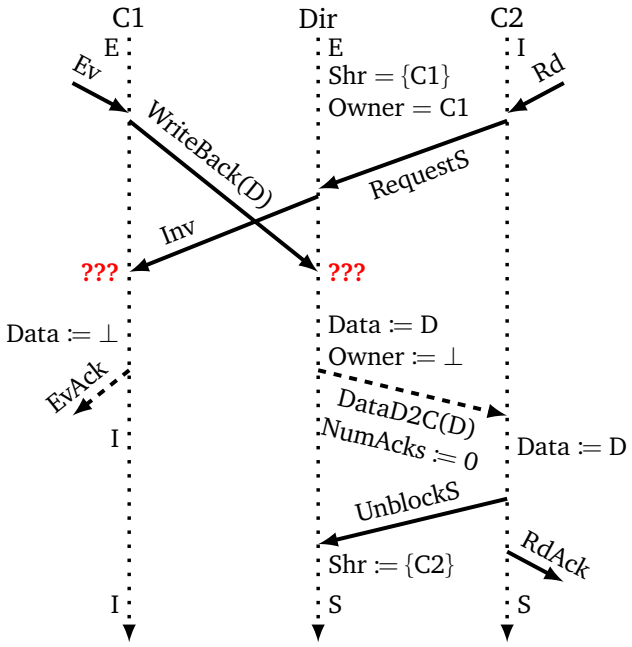


Figure 8.8: A Corner-case in the German/MSI Protocol

As part of the evaluation, we left *all* the five corner-case behaviors just described unspecified in the incomplete protocol. Our tool was able to successfully synthesize the behavior for the unspecified parts of the German/MSI protocol correspond to all of the five corner-cases described in this section, within a reasonable amount of time.

8.5 Summary of Experimental Results

Table 8.1 summarizes our experimental findings. All experiments were performed on a Linux desktop, with an Intel Core i7 CPU running at 3.4 GHz. with 8 GB of memory. The columns show the name of the benchmark, the number of unknown functions that were synthesized (# UF), the size of the search space for the unknown functions, the number of states in the complete protocol (# States), "symm. red." denotes symmetry reduced state space. The "# Iters." column shows the number of iterations required by the algorithm, where each iteration corresponds to analyzing one or more counterexample traces to generate additional constraints and querying the SMT solver for a new interpretation that also satisfies the newly added constraints. The last two columns show the total amount of time spent in SMT solving and the end-to-end synthesis time. We used the SMT solver Z3 [dMB08] in our implementation. Further, Z3 was used in its incremental mode as constraints were added across iterations.

Benchmark	# UF	Search Space	# States	# Iters.	SMT Time (s)	Total Time (s)
Peterson	3	2^{36}	60	14	0.1	0.13
Dijkstra	6	2^{192}	∼2000	30	27	64
German/MSI-2	16	$\sim 2^{4700}$	∼20000 (symm. red.)	217	31	298
German/MSI-4	28	$\sim 2^{7614}$	∼20000 (symm. red.)	419	898	1545
German/MSI-5	34	$\sim 2^{9000}$	∼20000 (symm. red.)	525	2261	3410

Table 8.1: Experimental Results for Completion of Protocols with Symmetry

The "German/MSI-n" rows correspond to the synthesizing the unknown behavior for the German/MSI protocol, with n out of the five unknown transitions left unspecified. In each case, we applied the heuristic to obtain minimally permissive guards and biased the search towards updates which leave the values of state variables unchanged as far as possible, except in the case of the Dijkstra benchmark, as mentioned earlier. Also, note that we ran each benchmark multiple times with different random seeds to the SMT solver to offset the variance due to random restarts in the SMT solver. The run times reported in Table 8.1 are the worst of the run times that we measured over these multiple runs.

8.5.1 Discussion

We now briefly discuss some qualitative aspects of our experiences with experimenting with the prototype tool, and highlight on the aspects that were crucial in making the approach work well.

Programmer Assistance

In all cases, the programmer specified the kinds of messages to handle in the states where the behavior was unknown. For example, in the case of the German/MSI protocol, the programmer indicated that in the IM state on the cache, it needs to handle an invalidation from the directory (see Figure 8.7). In general, the programmer specified *what* needs to be handled, but not the *how*. This was crucial to getting our approach to scale. Without this information, the algorithm would be required to handle *every possible* event in every possible situation. We have observed that sheer size of the constraints generated with this approach is too large for SMT solvers to process efficiently.

Synthesizing Symbolic Expressions

The interpretations returned by the SMT solver are in the form of tables, which specify the output of the unknown function on specific inputs. We mentioned that if a symbolic expression is required we can pass this output to a SyGuS solver, which will then return a symbolic expression. We were able to synthesize compact expressions in all cases using the enumerative SyGuS solver [ABJ+13].

Overhead of Decision Procedures

We observe from Table 8.1 that for the longer running benchmarks, the run time is dominated by SMT solving. In all of these cases, a very large fraction of the constraints asserted into the SMT solver are constraints to implement heuristics which are specifically aimed at guiding the SMT solver, and reducing the impact of non-deterministic choices made by the solver. Specialized decision procedures that handle these constraints at an algorithmic level [BP14] can greatly speed up the synthesis procedure. Another possibility is to not rely on an SMT solver to generate interpretations, but rather, use a SyGuS solver to generate symbolic expressions which will serve as interpretations. From our empirical observations, for small expression sizes (as is the case in most of the benchmarks we have evaluated), a SyGuS solver returns an interpretation much faster than an SMT solver. Unfortunately, the available SyGuS solvers do not handle the synthesis of multiple correlated functions well. So either (1) Existing SyGuS solvers would need to algorithmically support the synthesis of multiple correlated functions, or (2) The problem would need to be massaged into a form such that the SyGuS solver is only ever called upon to generate an interpretation for a *single* function.

9

Related Work

Synthesis of reactive systems has been studied extensively in literature. To describe, or even list all past research in this area would be a Herculean endeavor in itself, which we shall not attempt to undertake. Instead, we shall highlight a few key ideas that have shaped the landscape of this research area. We begin with describing the *classical* approaches to reactive synthesis purely from LTL, CTL, or CTL* specifications. We then briefly describe synthesis approaches based on *partial* or *incorrect* system descriptions, and proceed to a discussion of other approaches that share similarities, in spirit, with the approaches we propose in this manuscript. We conclude the discussion of related work by describing recent work in the area of straight-line and recursive program synthesis.

9.1 Classical Reactive Synthesis Techniques

Classical approaches to solve the synchronous (and non-distributed) version of reactive synthesis, *i.e.*, where the environment and the system make transitions synchronously in discrete steps, can be broadly classified into linear-time and branching-time techniques, depending on whether the specification is in LTL or a branching-time logic like CTL or CTL* respectively. In linear-time synthesis, the overall strategy [PR89, Ros92] is to (1) construct a (non-deterministic) Büchi automaton corresponding to the LTL specification φ, (2) Determinize the Büchi automaton [Saf88] into a deterministic Rabin automaton, (3) Interpret the resulting Rabin automaton as tree automaton on infinite trees over the Boolean predicates that are part of the specification and (4) Check if the resulting tree automaton is empty. This approach has a complexity of $\mathcal{O}(2^{2^n})$ (in fact, it is 2EXPTIME-complete [Ros92]) where n represents the syntactic size of the original LTL specification φ. Other approaches for LTL synthesis view the problem as a

controller-synthesis problem [RW89, TW94a, TW94b, MPS95]. This view of reactive synthesis as controller-synthesis has also been applied in the branching-time world for a subset of CTL and is shown to be NP-complete for full CTL [Ant95], when the controller is memory-less. It has also been shown that CTL control with memory is EXPTIME-complete, and CTL* control is 2EXPTIME-complete [Kup95a, Kup95b, KV96]. Distributed reactive synthesis has been shown to be undecidable [PR90, LT00, Tri04, FS05].

Classical LTL synthesis algorithms have found little practice, owing mostly to Safra's determinization procedure [Saf88] being notoriously difficult to implement [ATW06, THB95]. This has led to the development of "Safraless" approaches to synthesis [KPV06, KV05, EK14], which eschew Safra's determinization procedure and use alternative structures. Along the other dimension, it has been shown that restricting the specification language to a reasonably expressive subset of full LTL renders the reactive synthesis problem tractable, with polynomial time complexity [BJP$^+$12]. To deal with the undecidability of distributed protocol synthesis, recent advances have proposed bounded techniques for synthesis [FS05, FS13], often combined with symbolic reasoning [Ehl11, Ehl12]. Another interesting approach approximates the eventuality properties of an LTL specification, successively refining the approximation until an implementation can be synthesized [FJR11, FJR13]. Randomization, in the form of a genetic programming based algorithm has also been used to skirt around the undecidability of the problem [KP08, KP09].

One of the drawbacks of most classical synthesis techniques is that they operate over the Boolean domain and often, the final output of such a synthesis algorithm is a transition relation (or function) over propositional variables. This is typically not how a human views a reactive system, and as a result such a representation can be rather opaque to a human being. Also, while writing declarative specifications in a temporal logic does have theoretical elegance, it is non-trivial for a human being to describe a protocol using *only* such specifications.

The work described in this manuscript differs from classical synthesis techniques in that we allow a *mix* of specification languages: the high-level properties of the system are specified using declarative constructs, while the common-case behavior of the protocol is described in an operational manner and not required to be complete. Our techniques aim to fill in the tricky details using the high-level temporal logic specifications. A pleasant consequence of this is that the final artifact of our synthesis approaches is a human-readable, operational description of the system.

9.2 Synthesis from Partial or Incomplete Descriptions

The SKETCH system [SLRBE05, STB+06, SAT+07, SLJB08, Sol09] is a program synthesis framework where the correctness requirement is expressed as a — possibly a sub-optimal, but functionally correct — C program. The programmer then expresses the "shape" of the desired program as another C-like program, called the sketch with certain details — called "holes" in the SKETCH parlance — unspecified. The SKETCH system fills in the holes in the sketch such that the completed version of the sketch is functionally equivalent to the sub-optimal C program provided by the user. The SKETCH system has been successfully used to synthesize bit-stream programs for encryption and decryption [SLRBE05], finite state programs [STB+06] and stencil computations [SAT+07]. Perhaps the work that is most closely related to the work described in this manuscript is the PSKETCH system [SLJB08], which synthesizes concurrent data structures. The ideas we present in this manuscript are inspired by SKETCH and share a lot of methodological similarities with SKETCH. However, unlike SKETCH, we focus specifically on distributed reactive synthesis problems.

The more recently proposed storyboard programming approach [SS11, SS12] by the authors of SKETCH also shares several similarities with the techniques presented in this manuscript. Our notion of a scenario is very similar to the notion of a storyboard as used by Singh, et. al. A key difference is that storyboards seem to be more geared towards describing representational transformations over linked data structures, where there is no notion of time, whereas scenarios have an implicit notion of time associated with them.

Our work is also related to recent research on program repair [JGB05, vEJ13].The goal of program repair is to repair a buggy program, such that some objective function is maximized. To this end, techniques like modeling the problem as one of finding a memory-less strategy for a Büchi game [JGB05] and finding a repair such that the repair deviates as little as possible from the buggy program on non-buggy executions [vEJ13] have been proposed. However these techniques are closer in spirit to classical synthesis approaches than to the ones we describe in this manuscript and therefore suffer from many of the same shortcomings.

9.3 Synthesis from Sequence Charts

Specifying a reactive system using example scenarios — in the form of message sequence charts, or live sequence charts — also has a long tradition. In particular, the problem of

deriving an implementation that exhibits at least the behaviors specified by a given set of scenarios is well-studied [AEY03, UKM03, BBO12]. A particularly well-developed approach is *behavioral programming* [HMW12, DH01, HM03] that builds on an extension of message sequence charts, called *live sequence charts* [DH01], and has been shown to be effective for specifying the behavior of a single controller reacting with its environment. It is not clear how requirements in the form of temporal logic specifications can be supported in this framework.

The work in [BKKL10] generalizes Angluin's learning algorithm [Ang87] to synthesize automata from MSCs but does not allow for the specification of requirements and relies on the programmer to answer classification and equivalence queries and is therefore not automatic. The problem of inferring extended finite-state machines has been studied in the context of active learning [CHJS14], but the techniques are again, not automatic, and do not accommodate temporal logic specifications. Scenarios — in the form of "flows" — have also been used in the modular verification of cache coherence protocols [TT08, OTT09].

9.4 Straight-line and Recursive Program Synthesis

The earliest work that we are aware of in the area of synthesizing straight-line program fragments is the extensive work on what was then called "super-optimizations". The original problem was formulated by Massalin [Mas87], and the objective was to deduce the *smallest* possible program that was behaviorally identical to another, possibly longer and less efficient, program. The approach presented by Massalin [Mas87] could only scale to a programs with a very few instructions. Since then, more scalable algorithms have emerged [JNR02, JNZ06] and superoptimizers have also been applied in peephole optimizations and binary translation [BA06, BA08, SSCA15]. More recently, stochastic approaches have been successfully applied to yield scalable superoptimization algorithms [SSA13, SSA14]. Stochastic techniques techniques have also been applied to synthesize loop invariants [SA14].

Significant inroads have been made in the last decade or so in the area of synthesizing small program fragments to perform various tasks, starting from some form of formal specifications. The research on the SKETCH framework [SLRBE05] perhaps reinvigorated research in the area of program synthesis. The idea of using an unoptimized program as a specification for a more optimized version which is to be synthesized was novel. Although the initial system was for synthesis of bit-streaming programs [SLRBE05], the techniques were later adapted to sketching finite programs [STB+06], stencils computations [SAT+07], concurrent

data structures [SLJB08] as well as to synthesize code for data structure manipulations via storyboards [SS11]. Synthesis of data structure manipulation routines has also been explored in other recent work [FCD15, AGK13]. Other recent work has viewed the problem of synthesizing straight-line code as that of component-based synthesis [GJTV11, JGST10]. Enumerative approaches to synthesizing code fragments that are vectorized equivalents of unoptimized code has also been explored in recent work [BCG$^+$13].

More recently, the FlashFill algorithm [Gul11] was one of the first to leverage the notion of an *inductive* specification, which has been described in Chapter 5. The original FlashFill algorithm was designed for synthesizing string transformations in spreadsheets based on a few input-output examples demonstrating the desired transformation [Gul11]. However, since then, the techniques have been applied to a variety of different domains [KG15, BGHZ15, LG14, GKT11, SG12, PGGP14, PGBG12]. A framework called FlashMeta [PG15], which unifies the domain-specific inductive synthesis algorithms implemented in the rest of the Flash algorithms using a common abstract algorithm has also been recently developed.

Program Synthesis techniques have also recently been used to synthesize loop invariants. The ICE [GLMN14] and Alchemist [SGM15] are prime examples, along with algorithms that use a stochastic search [SA14]. A tool based on the Alchemist [SGM15] algorithm participated in the 2015 SyGuS competition in the invariant synthesis track. Decision trees based learners have also been explored recently for SyGuS solvers [GNMR15], where they have been primarily used to learn thresholds for affine classifiers. Type directed approaches to program synthesis from input-output examples have also recently been a subject of study [OZ15, Ose15, FOWZ16].

10

Conclusions

This chapter concludes this dissertation by first providing a brief summary of the research that has been described in this dissertation, followed by an orthogonal exploration of the themes that have been prevalent throughout this dissertation. We then highlight some avenues along which the work described in this dissertation can be improved and extended, and conclude with the author's opinions and outlook about research in the area of verification and program synthesis.

10.1 Summary of the Dissertation

This dissertation approached the problem of synthesizing a distributed reactive synthesis from the direction of *completing* an incomplete description of the protocol. Apart from the inherent difficulty of developing such protocols, our primary motivation for this approach was that it was not clear if describing the protocol purely using a temporal logic is necessarily easier than describing it operationally. Furthermore, the complexity of distributed reactive synthesis from temporal logic descriptions made it all the more appealing to view the synthesis problem as a fruitful interaction between a synthesis tool and a programmer.

We formalized the problem of protocol completion, and described our experience with using a theoretically elegant, but practically ineffective, symbolic algorithm to solve the protocol completion problem.

We then described a tool called TRANSIT where the programmer would symbolically codify the parts of the protocol that are well understood. The programmer would then describe fixes to counterexamples presented by the tool TRANSIT using *concolic snippets*, which were a mixture of symbolic constraints and constraints involving concrete values, the latter of which

is intended to be derived from a concrete erroneous execution. The programmer is a part of the synthesis loop in TRANSIT. Our prototype of TRANSIT was able to assist the programmer in describing a complex industrial cache coherence protocol, demonstrating the scalability of the proposed techniques.

We then made a brief digression to describe the SyGuS problem that came about as a generalization of the core computational problem solved within TRANSIT. The SyGuS effort was successful and annual SyGuS contests are conducted with participation growing each year. We described an enumerative strategy to solve instances of the SyGuS problems, studied the limitations of purely enumerative approaches, and proposed an improved algorithm that is enumerative in spirit, but demonstrates enhanced scalability. We empirically evaluated a tool based on this algorithm, called EUSOLVER, and found it to be able to solve a set of benchmarks that no existing SyGuS solver had been able to solve, to the best of our knowledge.

We then concluded our excursion into the world of syntax-guided synthesis and developed algorithms for distributed protocol synthesis that eliminated the programmer from the synthesis loop by automatically analyzing counterexamples and suitably constraining future solution candidates. We evaluated these algorithms on a variety of benchmarks, and observed that while they scaled to moderately complex protocols, their scalability was nonetheless lower than that of TRANSIT. We also described a model checking and synthesis framework, called KINARA, that we developed as part of this effort, and which has now been released as an open-source project.

10.2 Themes Explored in this Dissertation

Two themes have been pervasive throughout this dissertation. The first has been about the interplay between the amount of programmer involvement and the scalability of the synthesis algorithms. The second has been about the use of alternative and, hopefully more intuitive and convenient techniques, to specify programmer intent. We now discuss, in some detail, how each of these themes, has been explored in the research described in this dissertation.

10.2.1 Interplay between Programmer Involvement and Scalability

The TRANSIT system required the programmer to be a part of the synthesis loop. This resulted in the tool being scalable enough to assist a programmer to develop a large industrial cache coherence protocol. The implementation of TRANSIT described in this dissertation made

use of an enumerative algorithm that is simplistic in comparison with the decision tree based algorithm presented in Chapter 6. The scalability of TRANSIT is restricted only by the scalability of the expression inference algorithm. So, we can expect that TRANSIT could scale to being able to assist a programmer in designing even more complex protocols if coupled with better algorithms for expression inference, such as the one described in Chapter 6.

In contrast to TRANSIT, the work described in Chapters 7 and 8 was aimed at being fully automatic. While they could scale to reasonably complex protocols, we were unable to get them to scale to the industrial SGI-Origin cache coherence protocol that TRANSIT proved to be useful on. While, in general, more programmer inputs and involvement should imply an easier synthesis problem that the algorithms are required to solve, our experience with the use of scenarios in the work described in Chapter 7 was counter-intuitive: more programmer inputs, in the form of providing more scenarios resulted in larger incomplete state machines, which in turn led to poor scalability. However, an important point to note here is that in specifying the scenarios the programmer made minimal use of the state labeling techniques described in Section 7.2. A more extensive use of these techniques could have resulted in more compact incomplete state machines, and thus enhanced the scalability of the algorithms. The fact that the constraints were expressed as integer linear programs also contributed greatly to the scalability of the techniques presented in Chapter 7. This was something that we could not leverage when we allowed ESMs and ESM-SKs with state variables in Chapter 8, and we had to replace a rather lean ILP solver with a relatively heavy-weight SMT solver as the constraint solving engine.

The interplay between the amount of information provided by the programmer and scalability is also apparent in the work described in Chapter 8. There, had the programmer not specified a small set of transitions that were candidates for synthesis, the algorithm would not have scaled. By narrowing the search space using an expert's intuition, the programmer in effect enables the synthesis algorithms to be useful in providing assistance.

10.2.2 Use of Alternative Techniques to Specify Intent

Traditionally, research on reactive synthesis has focused on synthesis starting from temporal logic specifications or requirements. In addition to being computationally difficult, it is not clear that writing formal temporal logic specifications is necessarily easier, simpler or better than writing an operational description of the model, from a software engineering perspective.

The work presented in this dissertation, on the other hand, uses other techniques to specify the programmer intent. The TRANSIT system used *concolic snippets* added by the programmer over the course of developing a protocol. Program synthesis techniques were then used to obtain a candidate protocol which was consistent with the snippets provided by the programmer. A point to note here is that the set of concolic snippets may be *ambiguous*. The expression inference algorithm in TRANSIT uses the principle of Occam's razor and deems the *simplest* or the smallest expression to be most likely to explain the under-specified set of constraints.

Chapter 7 demonstrated how scenarios could be coupled with synthesis techniques in the construction of distributed protocols. We believe that scenarios are a more intuitive form of specification for distributed protocols, than specification of the protocol using a state machine-like abstraction. Although the problem definition in Chapter 8 demanded an incomplete protocol in the state machine abstraction, the incomplete protocol was in fact constructed from a set of scenarios.

10.3 Avenues for Future Work

There are several directions in which the work described in this dissertation can be extended and improved upon. We highlight what we consider to be the most fruitful directions in this section.

Use of Inductive Specifications in Protocol Completion

The scalability of the algorithms described in Chapter 8 leaves something to be desired. It is apparent from the summary of experimental results shown in Table 8.1 that the scalability is limited by the performance of the SMT solver. But as we have mentioned earlier, the constraints obtained on the uninterpreted functions are essentially *inductive specifications*. The only terms appearing in such constraints are the uninterpreted functions applied to concrete constant values, and concrete constant values. However, each constraint may involve disjunctions and may also refer to multiple unknown functions. In other words, these specifications are not *separable*, where separability is a concept we have defined in Section 6.2.1. Recent work [PG15] has studied how such inductive specifications can be leveraged to develop scalable synthesis algorithms. Although the work describes techniques for synthesis with disjunctive and non-separable constraints, it is not clear how they perform: all the instantiations of the algorithms seem to be for separable inductive specifications. Applying or adapting these techniques for use

in the context of constraints obtained from the automatic protocol completion problem might be an interesting area of future work. Efficient algorithms for solving constraints of this form would have an immediate impact on the scalability of the algorithms presented in Chapter 8, as it would obviate the need for the SMT solver

Synthesis Algorithms for Non-separable Specifications

The algorithm for solving the SyGuS problem described in Chapter 6 dealt only with separable specifications. The original ESOLVER however, could handle non-separable specifications and even specifications involving multiple functions, albeit with reduced scalability. Investigating if the techniques that make EUSOLVER scalable on separable specifications can be adapted for use on non-separable specifications would be a useful endeavor. Indeed, such specifications do occur in practice as shown in Chapter 8; in addition the SyGuS benchmark suite also has a small number of benchmarks with non-separable specifications.

10.4 Reflections on Verification and Program Synthesis

The problem of synthesizing a program or a circuit from a formal description of the behavior of the program or circuit is often mentioned as (one of) the holy grail of computer science, starting from Church's problem, presented in 1957 [Chu57]. The number of scholarly articles published on program synthesis in the past decade vindicates my[19] opinion that synthesis is a technology whose time has come. Someone who is even slightly less than optimistic might be of the opinion that it is premature: After all, we haven't been able to get verification techniques to scale beyond programs with which have in the order of a hundred thousand lines. Mission critical software systems like automotive control and avionics software are still largely without formal proofs of correctness. Why then attack a new problem when there are already so many unsolved problems?

I believe that research in program synthesis techniques could in fact prove beneficial to research in program verification techniques, and most certainly the other way around as well. My opinion is that the reason that program verification techniques have not seen the uptake that can be considered desirable is simply the relatively high entry barrier, coupled with ineffective

[19]Disclaimer: The views and opinions expressed in this section are solely those of the author, and are not endorsed by the dissertation supervisor, the dissertation committee, or any of the author's collaborators. Statements and predictions in this section may also not be strongly backed by evidence, empirical or otherwise, and serve solely to express the author's point-of-view.

feedback techniques when verification fails. The entry barrier is most commonly in the form of having to write annotations in the form of loop-invariants and heap separability assumptions — usually in some variant of formal first-order logic — for existing code bases, which are often large to begin with. While there exists no silver bullet and programmers will eventually have to bite the bullet and provide these annotations if they desire verified code, a lot can be done to make it less unpleasant to write these annotations. For example, my personal experience with the VCC framework [CDH+09] has been that the feedback provided when verification fails is relatively unhelpful. I am simply presented with an execution that invalidates the annotations that were provided, leaving me with little information about *how* to correct my annotations, and *what* a correct loop invariant is.

I believe that program synthesis techniques could help in remedying this situation by either (1) automatically trying to synthesize a loop invariant, or (2) providing me with *suggestions* that differ from the existing, proposed loop invariant in minor ways and are more likely to be correct. It delights me to see that program synthesis techniques are already being applied to synthesize invariants [SGH+13, SA14]. The second possibility has also been explored in the context of programs and reactive systems [vEJ13, vEJ15]. Such techniques would greatly reduce the entry barrier to using program verification to prove real-world systems correct.

While program synthesis techniques can help adoption of formal methods, I also believe that research in the area of human computer interaction has a large role to play in this arena in the near future. While formal methods researchers are most comfortable working with abstract objects in first order or other, more esoteric logics, the average programmer is unlikely to appreciate the succinctness and the precision of such formalisms. Research in natural language descriptions of such objects is likely to encourage adoption of formal methods far more than anything else. There has been work in using natural language for program synthesis [RGM15] and teaching and grading assignments for a course on automata theory [ADG+13, DKA+15], but I predict that the future will see more of such work, which will ultimately make formal methods more accessible.

Lastly, while research in the areas mentioned in this section can help in encouraging adoption of formal methods as an integral part of software development, the most powerful motivator is likely to be financial. So, the ultimate thrust must come from within an organization, making it an organizational policy to use and apply verification and program synthesis techniques. This is likely to happen only when researchers in this discipline actively collaborate with industrial

partners to find out what problems matter to them, and attempt to solve the — possibly un-fashionable and also difficult — problems that matter.

Bibliography

[ABJ+13] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided Synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20–23, 2013*, pages 1–8, 2013. [Cited on page 147.]

[ACR15] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. Synthesis Through Unification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II*, pages 163–179, 2015. [Cited on pages 77, 78, 79, 84, 85, 86, and 105.]

[ADG+13] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated Grading of DFA Constructions. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3–9, 2013*, 2013. [Cited on page 158.]

[AEY03] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of Message Sequence Charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003. [Cited on page 151.]

[AFSSL14] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Syntax-guided Synthesis Competition (SyGuS-COMP). `http://www.sygus.org`, 2014. [Cited on pages 49 and 71.]

[AGK13] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings*, pages 934–950, 2013. [Cited on page 152.]

[AII+13] Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michal Moskal, and Nikhil Swamy. Calibrating Research in Program

Synthesis Using 72,000 Hours of Programmer Time. Technical report, MSR, 2013. [Cited on page 86.]

[AMR+14] Rajeev Alur, Milo M. K. Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing Finite-State Protocols from Scenarios and Requirements. In *Hardware and Software: Verification and Testing - Proceedings of the 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18–20, 2014*, pages 75–91, 2014. [Cited on page 107.]

[Ang87] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. [Cited on page 151.]

[Ant95] Marco Antoniotti. *Synthesis and Verification of Discrete Controllers for Robotics and Manufacturing Devices with Temporal Logic and the Control-D System*. PhD thesis, New York University, New York, NY, USA, 1995. [Cited on page 149.]

[ARS+15] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic Completion of Distributed Protocols with Symmetry. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II*, pages 395–412, 2015. [Cited on page 121.]

[ATW06] Christoph Schulte Althoff, Wolfgang Thomas, and Nico Wallmeier. Observations on Determinization of Büchi Automata. In *Implementation and Application of Automata*, volume 3845 of *Lecture Notes in Computer Science*, pages 262–272. Springer Berlin Heidelberg, 2006. [Cited on page 149.]

[BA06] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21–25, 2006*, pages 394–403, 2006. [Cited on page 151.]

[BA08] Sorav Bansal and Alex Aiken. Binary Translation Using Peephole Superoptimizers. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8–10, 2008, San Diego, California, USA*, pages 177–192, 2008. [Cited on page 151.]

[BBO12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding Choreography Realizability. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2012, pages 191–202, New York, NY,

USA, 2012. ACM. [Cited on page 151.]

[BCG+13] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From Relational Verification to SIMD Loop Synthesis. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013, Shenzhen, China, February 23–27, 2013*, pages 123–134, 2013. [Cited on pages 71 and 152.]

[BGHZ15] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets using Examples. In *Proceedings of the* 36^{th} *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, Portland, OR, USA, June 15–17, 2015*, pages 218–228, 2015. [Cited on pages 75 and 152.]

[BJP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) Designs. *J. Comput. Syst. Sci.*, 78(3), 2012. [Cited on pages 1 and 149.]

[BKKL10] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering*, 36(3):390–408, May 2010. [Cited on page 151.]

[BKRS12] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to Büchi Automata Translation: Fast and More Deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*, pages 95–109, 2012. [Cited on pages 32 and 33.]

[BP14] Nikolaj Bjørner and Anh-Dung Phan. νZ - Maximal Satisfaction with Z3. In *SCSS 2014*, volume 30 of *EPiC Series*, pages 1–9. EasyChair, 2014. [Cited on page 147.]

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *DAC*, pages 40–45, 1990. [Cited on page 35.]

[Bry85] Randal E. Bryant. Symbolic Manipulation of Boolean Functions using a Graphical Representation. In *Proceedings of the* 22^{nd} *ACM/IEEE Conference on Design Automation, DAC 1985, Las Vegas, Nevada, USA, 1985.*, pages 688–694, 1985. [Cited on page 35.]

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. [Cited on page 35.]

[BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.smt-lib.org`, 2010. [Cited on page 71.]

[BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010. [Cited on page 71.]

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An Open-source Tool for Symbolic Model Checking. In *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer Berlin Heidelberg, 2002. [Cited on page 129.]

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin Heidelberg, 2009. [Cited on page 158.]

[CHJS14] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning Extended Finite State Machines. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, volume 8702 of *Lecture Notes in Computer Science*, pages 250–264. Springer International Publishing, 2014. [Cited on page 151.]

[Chu57] Alonzo Church. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Summaries of Talks Presented at the Summer Institute for Symbolic Logic, Cornell University, 1957*, pages 3–50, 1957. [Cited on page 157.]

[CJEF96] Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996. [Cited on page 46.]

[CMP04] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 382–398. Springer Berlin Heidelberg, 2004. [Cited on page 139.]

[DGV99] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved Automata Gener-

ation for Linear Temporal Logic. In *Computer Aided Verification,* 11^{th} *International Conference, CAV 1999, Trento, Italy, July 6–10, 1999, Proceedings*, pages 249–260, 1999. [Cited on page 32.]

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001. [Cited on pages 48 and 151.]

[Dij74] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, November 1974. [Cited on page 138.]

[Dil96] David L. Dill. The Murφ Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV 1996, pages 390–393, London, UK, UK, 1996. Springer-Verlag. [Cited on pages 4, 46, 129, 130, 133, 134, and 135.]

[DKA+15] Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan, and Björn Hartmann. How Can Automatic Feedback Help Students Construct Automata? *ACM Trans. Comput.-Hum. Interact.*, 22(2):9:1–9:24, 2015. [Cited on page 158.]

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008. [Cited on pages 61, 98, 102, and 145.]

[Dur14] Alexandre Duret-Lutz. LTL translation improvements in Spot 1.0. *IJCCBS*, 5(1/2):31–54, 2014. [Cited on pages 32 and 33.]

[EH00] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi Automata. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22–25, 2000, Proceedings*, pages 153–167, 2000. [Cited on page 32.]

[Ehl11] Rüdiger Ehlers. UNBEAST: Symbolic Bounded Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin Heidelberg, 2011. [Cited on page 149.]

[Ehl12] Rüdiger Ehlers. Symbolic Bounded Synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012. [Cited on page 149.]

[EK14] Javier Esparza and Jan Křetínský. From LTL to Deterministic Automata: A Safraless Compositional Approach. In *Computer Aided Verification*, volume 8559 of *Lecture*

Notes in Computer Science, pages 192–208. Springer International Publishing, 2014. [Cited on page 149.]

[ES97] E. Allen Emerson and A. Prasad Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, 1997. [Cited on pages 46, 130, 133, 134, 135, and 136.]

[EW03] E. Allen Emerson and Thomas Wahl. On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking. In *Correct Hardware Design and Verification Methods,* 12^{th} *IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21–24, 2003, Proceedings*, pages 216–230, 2003. [Cited on page 46.]

[EW05] E. Allen Emerson and Thomas Wahl. Dynamic Symmetry Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems,* 11^{th} *International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005, Proceedings*, pages 382–396, 2005. [Cited on page 46.]

[FCD15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the* 36^{th} *ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015*, PLDI 2015, pages 229–239, 2015. [Cited on page 152.]

[FJR11] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and Compositional Algorithms for LTL Synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011. [Cited on page 149.]

[FJR13] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Exploiting Structure in LTL Synthesis. *STTT*, 15(5-6):541–561, 2013. [Cited on page 149.]

[FOWZ16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the* 43^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 – 22, 2016*, pages 802–815, 2016. [Cited on page 152.]

[FS05] Bernd Finkbeiner and Sven Schewe. Uniform Distributed Synthesis. In *IEEE*

Symposium on Logic in Computer Science, pages 321–330, 2005. [Cited on pages 1 and 149.]

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded Synthesis. *Software Tools for Tchnology Transfer*, 15(5-6):519–539, 2013. [Cited on page 149.]

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011*, pages 62–73, 2011. [Cited on pages 71, 78, 85, and 152.]

[GKT11] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing Geometry Constructions. In *Proceedings of the* 32nd *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011*, pages 50–61, 2011. [Cited on page 152.]

[GLMN14] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification -* 26th *International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings*, pages 69–87, 2014. [Cited on page 152.]

[GNMR15] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. Learning Invariants using Decision Trees and Implication Counterexamples, *Technical Report*. `http://web.engr.illinois.edu/~garg11/papers/dt-ice.pdf`, 2015. [Cited on page 152.]

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18–22, 2001, Proceedings*, pages 53–65, 2001. [Cited on pages 32 and 33.]

[GT14] Adrià Gascón and Ashish Tiwari. Synthesis of a Simple Self-stabilizing System. In *Proceedings of the 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23–24, 2014.*, pages 5–16, 2014. [Cited on page 138.]

[Gul11] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-output Examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011*, pages 317–330, 2011. [Cited on pages 71, 75, and 152.]

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming*

Using LSCs and the Play-Engine. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. [Cited on page 151.]

[HMW12] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012. [Cited on page 151.]

[Hol97] Gerard J. Holzmann. The Model Checker Spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. [Cited on pages 4 and 129.]

[HR76] Laurent Hyafil and Ronald L. Rivest. Constructing Optimal Binary Decision Trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976. [Cited on page 89.]

[ID96] C. Norris Ip and David L. Dill. Better Verification through Symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996. [Cited on pages 4, 29, 30, 46, 129, 133, 134, and 135.]

[ITU96] ITU Telecommunication Standardization Sector. *ITU-R recommendation Z.120, Message Sequence Charts (MSC 1996)*, May 1996. [Cited on page 48.]

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program Repair as a Game. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005, Edinburgh, Scotland, UK, July 6–10, 2005*, pages 226–238, 2005. [Cited on page 150.]

[JGST10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010*, pages 215–224, 2010. [Cited on pages 71, 78, 85, and 152.]

[JNR02] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A Goal-directed Superoptimizer. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17–19, 2002*, pages 304–314, 2002. [Cited on page 151.]

[JNZ06] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6):967–989, 2006. [Cited on page 151.]

[JRU13] Garvit Juniwal, Mukund Raghothaman, and Abhishek Udupa. SyGuS Solver Implementations. `https://github.com/rishabhs/sygus-comp14`, 2013. [Cited

on page 78.]

[KG15] Dileep Kini and Sumit Gulwani. FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015*, pages 776–783, 2015. [Cited on pages 75 and 152.]

[KP08] Gal Katz and Doron Peled. Model Checking-Based Genetic Programming with an Application to Mutual Exclusion. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, LNCS 4963, pages 141–156, 2008. [Cited on page 149.]

[KP09] Gal Katz and Doron Peled. Synthesizing Solutions to the Leader Election Problem Using Model Checking and Genetic Programming. In *Haifa Verification Conference*, pages 117–132, 2009. [Cited on page 149.]

[KPRS06] Yonit Kesten, Amir Pnueli, Li-on Raviv, and Elad Shahar. Model Checking with Strong Fairness. *Formal Methods in System Design*, 28(1):57–84, 2006. [Cited on pages 35, 36, 38, 39, 41, and 42.]

[KPV06] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless Compositional Synthesis. In *Proceedings of the 18th International Conference on Computer Aided Verification, CAV 2006, Seattle, WA, USA, August 17–20, 2006*, pages 31–44, 2006. [Cited on page 149.]

[KR09] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5$^{\text{th}}$ edition, 2009. [Cited on pages 108 and 111.]

[Kup95a] Orna Kupferman. Augmenting Branching Temporal Logics with Existential Quantification over Atomic Propositions. In *Proceedings of the 7th International Conference on Computer Aided Verification, CAV 1995, Liège, Belgium, July, 3–5, 1995*, pages 325–338, 1995. [Cited on page 149.]

[Kup95b] Orna Kupferman. *Model checking for Branching-time Temporal Logics*. PhD thesis, Technion, Haifa, Israel, 1995. [Cited on page 149.]

[KV96] Orna Kupferman and Moshe Y. Vardi. Module Checking. In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV 1996, New Brunswick, NJ, USA, July 31 – August 3, 1996*, pages 75–86, 1996. [Cited on page 149.]

[KV05] Orna Kupferman and Moshe Y. Vardi. Safraless Decision Procedures. In *Proceedings*

of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23–25 October 2005, Pittsburgh, PA, USA, pages 531–542, 2005. [Cited on page 149.]

[LG14] Vu Le and Sumit Gulwani. FlashExtract: A Framework for Data Extraction by Examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom - June 09–11, 2014*, page 55, 2014. [Cited on pages 75 and 152.]

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2–4, 1997*, pages 241–251, 1997. [Cited on pages 15, 55, 56, 68, and 69.]

[LP85] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 97–107, 1985. [Cited on page 32.]

[LT00] Hichem Lamouchi and John Thistle. Effective Control Synthesis for DES Under Partial Observations. In *39th IEEE Conference on Decision and Control*, pages 22–28, 2000. [Cited on pages 1 and 149.]

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. [Cited on page 23.]

[Mas87] Henry Massalin. Superoptimizer - A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5–8, 1987.*, pages 122–126, 1987. [Cited on page 151.]

[MNS16] Parthasarathy Madhusudan, Daniel Neider, and Shambwaditya Saha. Synthesizing Piece-wise Functions by Learning Classifiers. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, Netherlands, April 2 – 8, 2016. Proceedings*, 2016. [Cited on pages 77, 79, and 82.]

[MPS95] Oded Maler, Amir Pnueli, and Joseph Sifakis. On the Synthesis of Discrete Controllers for Timed Systems. In *Symposium on Theoretical Aspects of Computer Science*

(STACS), pages 229–242, 1995. [Cited on page 149.]

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005. [Cited on page 66.]

[Mur98] Sreerama K. Murthy. Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, December 1998. [Cited on page 89.]

[Org05] SMT-COMP Organizers. Satisfiability Modulo Theories Competition (SMT-COMP). `http://www.smtcomp.org`, 2005. [Cited on page 71.]

[Ose15] Peter-Michael Osera. *Program Synthesis with Types*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2015. [Cited on page 152.]

[OTT09] John W. O'Leary, Murali Talupur, and Mark R. Tuttle. Protocol Verification using Flows: An Industrial Experience. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15–18 November 2009, Austin, Texas, USA*, pages 172–179, 2009. [Cited on page 151.]

[OZ15] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed Program Synthesis. In *Proceedings of the* 36$^{\text{th}}$ *ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015*, pages 619–630, 2015. [Cited on page 152.]

[PG15] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25–30, 2015*, pages 107–126, 2015. [Cited on pages 75, 76, 152, and 156.]

[PGBG12] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed Completion of Partial Expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China - June 11–16, 2012*, pages 275–286, 2012. [Cited on page 152.]

[PGGP14] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI 2014, Edinburgh, United Kingdom - June 09–11, 2014, pages 408–418, 2014. [Cited on page 152.]

[PR89] Amir Pnueli and Roni Rosner. On the Synthesis of a Reactive Module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, 1989. [Cited on pages 1 and 148.]

[PR90] Amir Pnueli and Roni Rosner. Distributed Reactive Systems Are Hard to Synthesize. In *31st Annual Symposium on Foundations of Computer Science*, pages 746–757, 1990. [Cited on pages 1 and 149.]

[Qui86] J. Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986. [Cited on page 89.]

[Qui87] J. Ross Quinlan. Simplifying Decision Trees. *International Journal of Man-Machine Studies*, 27(3):221–234, 1987. [Cited on page 89.]

[Qui96] J. Ross Quinlan. Learning Decision Tree Classifiers. *ACM Computing Survey*, 28(1):71–72, 1996. [Cited on page 89.]

[RDK+15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27$^{\text{th}}$ International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part II*, pages 198–216, 2015. [Cited on pages 77, 78, 79, 83, 86, 103, and 105.]

[RGM15] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015*, pages 792–800, 2015. [Cited on page 158.]

[Ros92] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992. [Cited on page 148.]

[RU14] Mukund Raghothaman and Abhishek Udupa. Language to Specify Syntax-guided Synthesis Problems. *CoRR*, abs/1405.5590, 2014. [Cited on pages 71 and 78.]

[RW89] Peter J. G. Ramadge and W. Murray Wonham. The Control of Discrete Event Systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989. [Cited on pages 1 and 149.]

[SA14] Rahul Sharma and Alex Aiken. From Invariant Checking to Invariant Inference

Using Randomized Search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 88–105, 2014. [Cited on pages 151, 152, and 158.]

[Saf88] Shmuel Safra. On the Complexity of ω-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24–26 October 1988*, pages 319–327, 1988. [Cited on pages 148 and 149.]

[SAT+07] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching Stencils. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10–13, 2007*, pages 167–178, 2007. [Cited on pages 46, 74, 150, and 151.]

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi Automata from LTL Formulae. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15–19, 2000, Proceedings*, pages 248–263, 2000. [Cited on pages 32, 37, and 41.]

[SC85] A. Prasad Sistla and Edmund M. Clarke. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3):733–749, 1985. [Cited on page 115.]

[SG12] Rishabh Singh and Sumit Gulwani. Synthesizing Number Transformations from Input-Output Examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7–13, 2012 Proceedings*, pages 634–651, 2012. [Cited on pages 75 and 152.]

[SG15] Rishabh Singh and Sumit Gulwani. Predicting a Correct Program in Programming by Example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I*, pages 398–414, 2015. [Cited on page 76.]

[SGE00] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. SMC: A Symmetry-based Model Checker for Verification of Safety and Liveness Properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000. [Cited on pages 46 and 129.]

[SGH+13] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *Programming Languages and Systems - 22nd European Symposium on Programming,*

ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings, pages 574–592, 2013. [Cited on page 158.]

[SGM15] Shambwaditya Saha, Pranav Garg, and P. Madhusudan. Alchemist: Learning Guarded Affine Functions. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I*, pages 440–446, 2015. [Cited on page 152.]

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011. [Cited on pages 67 and 68.]

[SLJB08] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching Concurrent Data Structures. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2008, 2008. [Cited on pages 74, 150, and 152.]

[SLRBE05] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and K. Ebcioğlu. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation, PLDI 2005*, 2005. [Cited on pages 2, 46, 74, 82, 150, and 151.]

[Sol09] Armando Solar-Lezama. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems, APLAS 2009, Seoul, Korea, December 14–16, 2009*, pages 4–13, 2009. [Cited on pages 46, 74, and 150.]

[Som15] Fabio Somenzi. CUDD: CU Decision Diagram Package Release 2.5.0. `http://vlsi.colorado.edu/~fabio/CUDD`, 2015. [Cited on page 42.]

[SS11] Rishabh Singh and Armando Solar-Lezama. Synthesizing Data Structure Manipulations from Storyboards. In *SIGSOFT/FSE 2011 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC 2011: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5–9, 2011*, pages 289–299, 2011. [Cited on pages 150 and 152.]

[SS12] Rishabh Singh and Armando Solar-Lezama. SPT: Storyboard Programming Tool. In *Proceedings of the 24th International Conference on Computer Aided Verification, CAV 2012, Berkeley, CA, USA, July 7–13, 2012*, pages 738–743, 2012. [Cited on

page 150.]

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA - March 16 – 20, 2013*, pages 305–316, 2013. [Cited on pages 71, 78, 85, and 151.]

[SSA14] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09–11, 2014*, page 9, 2014. [Cited on page 151.]

[SSCA15] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Conditionally Correct Superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25–30, 2015*, pages 147–162, 2015. [Cited on page 151.]

[STB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12^{th} International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21–25, 2006*, pages 404–415, 2006. [Cited on pages 46, 74, 150, and 151.]

[Tar72] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. [Cited on page 137.]

[TB13] Emina Torlak and Rastislav Bodík. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, 2013. [Cited on page 74.]

[TB14] Emina Torlak and Rastislav Bodík. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the* 35^{th} *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2014, pages 530–541, 2014. [Cited on page 74.]

[THB95] Serdar Taşıran, Ramin Hojati, and Robert K. Brayton. Language Containment of Non-deterministic ω-automata. In *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME 1995,*

Frankfurt/Main, Germany, October 2–4, 1995, volume 987 of *Lecture Notes in Computer Science*, pages 261–277. Springer Berlin Heidelberg, 1995. [Cited on page 149.]

[Tri04] Stavros Tripakis. Undecidable Problems of Decentralized Observation and Control on Regular Languages. *Information Processing Letters*, 90(1):21–28, April 2004. [Cited on pages 1 and 149.]

[Tse83] Grigorii Samuilovich Tseitin. On the Complexity of Derivation in Propositional Calculus. *Symbolic Computation*, pages 466–483, 1983. [Cited on page 81.]

[TT08] Murali Talupur and Mark R. Tuttle. Going with the Flow: Parameterized Verification using Message Flows. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17–20 November 2008*, pages 1–8, 2008. [Cited on pages 48, 49, 139, and 151.]

[TW94a] John G. Thistle and W. Murray Wonham. Control of Infinite Behavior of Finite Automata. *SIAM J. Control Optim.*, 32(4):1075–1097, July 1994. [Cited on page 149.]

[TW94b] John G. Thistle and W. Murray Wonham. Supervision of Infinite Behavior of Discrete-Event Systems. *SIAM Journal on Control and Optimization*, 32(4):1098–1113, 1994. [Cited on page 149.]

[UKM03] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, February 2003. [Cited on page 151.]

[URD+13] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16–19, 2013*, pages 287–296, 2013. [Cited on pages 48 and 51.]

[vEJ13] Christian von Essen and Barbara Jobstmann. Program Repair without Regret. In *Proceedings of the 25th International Conference on Computer Aided Verification, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013*, pages 896–911, 2013. [Cited on pages 150 and 158.]

[vEJ15] Christian von Essen and Barbara Jobstmann. Program Repair Without Regret. *Formal Methods in System Design*, 47(1):26–50, 2015. [Cited on page 158.]

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning About Infinite Computations. *Inf. Comput.*, 115(1):1–37, 1994. [Cited on page 32.]

[WBE08] Thomas Wahl, Nicolas Blanc, and E. Allen Emerson. SVISS: Symbolic Verification of Symmetric Systems. In *Tools and Algorithms for the Construction and Analysis of Systems,* 14^{th} *International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 – April 6, 2008. Proceedings*, pages 459–462, 2008. [Cited on page 46.]

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about Infinite Computation Paths (Extended Abstract). In 24^{th} *Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7–9 November 1983*, pages 185–194, 1983. [Cited on page 32.]