TU KAISERSLAUTERN

DOCTORAL THESIS

Algorithms and Tools for Verification and Testing of Asynchronous Programs

A dissertation submitted to

the Department of Computer Science, TU Kaiserslautern

by

Zilong Wang

2015

Abstract

Algorithms and Tools for Verification and Testing of Asynchronous Programs

by Zilong Wang

Software is becoming increasingly concurrent: parallelization, decentralization, and reactivity necessitate asynchronous programming in which processes communicate by posting messages/tasks to others' message/task buffers. Asynchronous programming has been widely used to build fast servers and routers, embedded systems and sensor networks, and is the basis of Web programming using Javascript. Languages such as Erlang and Scala have adopted asynchronous programming as a fundamental concept with which highly scalable and highly reliable distributed systems are built.

Asynchronous programs are challenging to implement correctly: the loose coupling between asynchronously executed tasks makes the control and data dependencies difficult to follow. Even subtle design and programming mistakes on the programs have the capability to introduce erroneous or divergent behaviors. As asynchronous programs are typically written to provide a reliable, high-performance infrastructure, there is a critical need for analysis techniques to guarantee their correctness.

In this dissertation, I provide scalable verification and testing tools to make asynchronous programs more reliable. I show that the combination of counter abstraction and partial order reduction is an effective approach for the verification of asynchronous systems by presenting PROVKEEPER and KUAI, two scalable verifiers for two types of asynchronous systems. I also provide a theoretical result that proves a counter-abstraction based algorithm called expand-enlarge-check, is an asymptotically optimal algorithm for the coverability problem of branching vector addition systems as which many asynchronous programs can be modeled. In addition, I present BBS and LLSPLAT, two testing tools for asynchronous programs that efficiently uncover many subtle memory violation bugs .

Acknowledgements

I would like to thank the many people who have encouraged my PhD studies and made my years at MPI-SWS and TU-KL enjoyable.

I would like to thank my adviser Rupak Majumdar for his many years of exceptional guidance. He has not only provided me the necessary vision, encouragement and advice throughout my graduate school journey but also given me great freedom and trust to pursue independent research.

I thank my committee members and reviewers, Roland Meyer, Pierre Ganty, and Klaus Schneider, for their valuable feedback and being flexible with my constraints.

I thank the many MPI-SWS and TU-KL professors who provided me valuable classes and opinions: Roland Meyer, not only for his excellent concurrent theory and automata courses that led me to the beautiful world of formal methods, but also for his careful guidance and inspiration during the collaboration with him; Deepak Garg, for his classes on type theories and logics; Ruzica Piskac, for her classes on decision procedures and SAT/SMT theories.

I would like to thank the many academic collaborators: Indranil Saha, K.C. Shashidhar, Sai Deep Tetali, and Min Gao (among others) for the many fruitful discussions I have had with them. They have influenced me in more ways than they could imagine.

I had a great time during my internship at Microsoft Research, India. I thank Akash Lal for hosting me there, the careful explanation of the large-scale verification tools used in Microsoft, and his experience about doing research in industry.

I would also like to thank all friends at MPI-SWS for making my day-to-day life entertaining: Ezgi Cicek, Yan Chen, Dmitry Chistikov, Rayna Dimitrova, Susanne van den Elsen, Nancy Estrada, Johannes Kloos, Ori Lahav , Cheng Li, Filip Niksic, Vinayak Prabhu, and Anne-Kathrin Schmuck. I specially thank Reinhard Munz for giving me a lot of free driving to Saarland University. Without his kind help, I would have had to start off at 5am to catch up with a train to Saarbruecken everyday.

Finally, I thank my parents for their unconditional love, patience and understanding. I dedicate this dissertation to them.

Contents

List of Figures

List of Tables

To my Family.

Chapter 1

Introduction

Software is becoming increasingly concurrent: parallelization (e.g., in scientific computations), decentralization (e.g., in web applications), and reactivity (e.g., for GUI and web servers) necessitate asynchronous computations. Although shared-memory implementations are often possible, the burden of preventing unwanted thread interleavings without crippling performance is onerous. Many have instead adopted *asynchronous programming models* in which processes communicate by posting messages/-tasks to others' message/task buffers—Miller et al.[94] discuss why such models provide good programming abstractions. An asynchronous program can involve either a single process or multiple processes: a single-process asynchronous program executes a series of short-lived tasks one-by-one, and each task may potentially buffering additional tasks to be executed later. Since single-process asynchronous models ensure quick response to incoming events (e.g., user input, connection requests), they have been widely used to build fast servers and routers [71, 99], embedded systems and sensor networks [58], and are the basis of Web programming using Javascript. On the other hand, in a multi-process asynchronous program, each process handles messages from its own buffer, and may communicate with other processes by sending messages to others' buffers. Languages such as Erlang and Scala have adopted the multi-process asynchronous programming as a fundamental concept with which highly scalable and highly reliable distributed systems are built.

Asynchronous programs are challenging to implement correctly. Writing correct single-process asynchronous programs is hard since the loose coupling between asynchronously executed tasks makes the control and data dependencies in the programs difficult to follow. Writing correct multi-process asynchronous programs is also hard because of the large amount of nondeterminisms introduced by process interleavings and message delays. Even subtle design and programming mistakes on the programs have the capability to introduce erroneous or divergent behaviors. As asynchronous programs are typically written to provide a reliable, high-performance infrastructure, there is a critical need for analysis techniques to guarantee their correctness.

In this dissertation, I provide scalable verification and testing tools to make asynchronous programs more reliable.

Verification of Asynchronous Programs The goal of verification is that, given an asynchronous program and a property, check whether the property holds in the program. As side effect, a verifier may provide a witness explaining why the property does not hold in the program.

Asynchronous programs are notoriously hard to verify because they may contain infinitely many states due to the unbounded number of contents in buffers. It is impossible for a verifier to naively examine all states to check a property holds, from the algorithmic point of view.

In the dissertation, I show that the combination of *counter abstraction* and *partial order reduction* (CntAbs+POR) is an effective approach to verify asynchronous programs in which contents in buffers are unordered. Counter abstraction bounds the number of contents of the buffers abstractly: given a pre-defined bound k, as long as a buffer has no more than k contents, counter abstraction counts the contents of the buffer precisely. However, if the buffer contains more than k contents, counter abstraction regards it as if it contains infinitely many contents. Therefore, from the verification point view, the size of the buffer becomes finite after counter abstraction: either from 0 to k, or infinity. Counter abstraction provides a finite-state over-approximation of the behaviors of an asynchronous program. Thus if the over-approximation can be easily proved to be correct w.r.t. a property, the original asynchronous program is correct w.r.t. the property, too.

Once the state space becomes finite, partial order reduction comes into play to further reduce the number of states to be examined, intuitively, by analyzing "important" ones only. Partial order reduction is the key to make verification of asynchronous programs from possible to practical: I empirically show that the verification time on realistic asynchronous systems is significantly reduced by partial order reduction techniques.

In the dissertation, I present two applications following the CntAbs+POR approach. I present (1) PROVKEEPER, a verifier for message passing systems, and (2) KUAI, a verifier for software-defined networks (SDNs). PROVKEEPER verifies *provenance*-related properties on message passing systems such as browser extensions, ensuring correct access control and information dissemination. KUAI verifies safety properties on SDNs, such as no packet-forwarding loops, no black holes, and correct enforcement of middlebox policies, etc. I empirically show that both PROVKEEPER and KUAI are scalable to verify realistic asynchronous systems efficiently.

Besides the above practical tools, I also provide a theoretical result about the expand-enlarge-check algorithm (EEC) [47], which is a counter-abstraction based decision procedure for Petri nets [100] in which many asynchronous programs can be modeled [39, 45, 66, 67, 83, 112]. I extend EEC to branching vector addition systems (BVAS) [35], a model that is more expressive than Petri nets, and prove that EEC is an *asymptotically optimal* algorithm for both BVAS and Petri nets.

Testing of Asynchronous Programs When contents in buffers are FIFO-ordered in asynchronous programs, precise algorithmic reasoning such as state-reachability becomes undecidable [16], even when there is only a single finite-state process (posting messages to itself). Thus one cannot expect algorithmic tools that not only keep the FIFO order requirement on contents but also prove the correctness of such asynchronous programs. Instead, I choose to keep the FIFO order requirement but develop efficient and scalable *testing* tools for such asynchronous programs. The goal of testing is that, given an asynchronous program and a property, detect as many bugs as possible that violate the property in the program. Testing is not required to *prove* the property holds in asynchronous programs.

Sequentialization has been shown to be very successful for finding bugs in asynchronous programs [4, 14, 49, 75, 76, 78, 79, 96, 103, 104, 116, 118]. The key idea of sequentialization is to define a *bounding parameter* to translate an asynchronous program into a *sequential program* such that the behaviours of the sequential program is a subset of the ones of the original asynchronous program. In other words, the sequential program is an under-approximation of the original asynchronous program: if the sequential program is buggy, then the asynchronous program is also buggy. Since testing of sequential programs is well-understood, any tools that work for sequential programs can be used for finding bugs in asynchronous programs.

Two factors are considered as the keys to maximize the value of sequentialization. The first factor is to show a bounding parameter is effective in the sense that under a *small* bound, many interesting bugs can be found already in *realistic applications*. This is important because (1) a smaller bound results in a smaller under-approximation which is easier for a tool to analyze, and (2) many bounding parameters can be naively defined in theory but are not scalable to find bugs efficiently in real applications. The second factor is the capability of the underlying tools for sequential programs. After all, it is the tools that perform the work for finding bugs.

In the dissertation, I present two testing tools BBS and LLSPLAT, keeping the above two factors in mind. Bouajjani and Emmi introduced *phase-bounding* sequentialization algorithm [13] for asynchronous programs. However, there was no empirical evaluation to show the practical value of phase-bounding. I implement the phase-bounding algorithm in the tool BBS and use BBS to test TinyOS [46, 58] programs, which are widely used in wireless sensor networks. The empirical results indicate that a variety of subtle memory violation bugs are manifested within a small phase bound (3 in most of the cases). From the evaluation, I conclude that phase-bounding is an effective approach in bug finding for asynchronous programs.

The next contribution is LLSPLAT, a testing tool for sequential programs. LLSPLAT combines concolic testing [52, 111] and bounded model checking [29, 31, 73, 92] together to gain better testing performance. I evaluate LLSPLAT with two state-of-the-art concolic testing tools CREST [19] and KLEE [20] using 36 standard benchmarks. The evaluation shows that (1) for the same time budget (an hour per program), LLSPLAT provides on average 31%, 19%, 20%, 21% higher branch coverage than CREST's four search strategies, and on average 21% higher branch coverage than KLEE, and (2) LLSPLAT achieves higher branch coverage quickly: LLSPLAT starts to outperform CREST and KLEE after at most 3 minutes. In addition, I also evaluate LLSPLAT with the state-of-the-art bounded model checker CBMC [73] using 13 sequentialized SystemC benchmarks. The experiments show that LLSPLAT can find bugs more quickly than CBMC.

1.1 Outline

The rest of my dissertation is organized as follows. Chapter 2–4 present PROVKEEPER, KUAI, and the complexity results of EEC, respectively, which show the power of counter abstraction and partial order reduction for the verification of asynchronous programs. Chapter 5 and Chapter 6 present BBS and LLSPLAT, respectively, which enrich the techniques for testing asynchronous programs. The first four contributions have been published in [86], [88], [90], and [89], respectively. The work about LLSPLAT is currently under submission.

Chapter 2

PROVKEEPER: A Provenance Verifier for Message Passing Programs

2.1 Introduction

Controlled access and dissemination of data is a key ingredient of system security: we do not want secret information to reach untrusted principals and we do not want to receive bad information (indirectly) from untrusted principals. Many organizations receive private information from users and this information is passed around within the organization to carry out business-critical activities. These organizations must ensure that the data is not accidentally disclosed to unauthorized users, as the potential cost of disclosure can be high. Moreover, in many domains, such as healthcare and finance, the control of data is required by regulatory agencies through legislation such as HIPAA and GLBA.

We present an abstract model of information dissemination in message passing systems, and a static analyzer to verify correct dissemination. We model systems as concurrent message passing processes, one process for each principal in the system. Processes communicate by sending and receiving messages via a shared set of channels. Channels are unbounded, but can reorder messages. Sends are non-blocking, but receive actions block until a message is available.

To track information about the origin and access history of a message, we augment messages with *provenance* annotations. Roughly, the provenance of a message is a function of the sequence of principles that have transmitted the message in the past. Depending on the function, we get different provenance annotations. For example, the annotation can simply be the sequence of principals. Whenever a principal sends a message, we append the name of the principal to the current provenance of the message. The *provenance verification problem* asks, given a message passing program, a variable in the program, and a set of allowed provenance annotations, whether the provenance of every message stored in the variable, on every run of the program, belongs to the set of allowed provenances.

Consider a healthcare system in which a patient sends health questions to a secretary or a nurse, who in turn, forwards the question to doctors. An information-dissemination policy may require that every health answer received by the patient has been seen by at least one doctor. That is, the provenance of every message received by the patient must belong to the regular language $\mathsf{Patient}(\mathsf{Secretary} + \mathsf{Nurse})\,\mathsf{Doctor}^+$.

We consider provenance verification for general provenance domains satisfying an algebraic requirement. Static provenance verification is hard because of two sources of unboundedness in the model. First, the provenance information associated with a single message can be unbounded. For example there is no bound on the number of doctors who see a health question before an answer is sent back. Second, the number of pending messages in the system can be unbounded. We tackle these two sources of unboundedness as follows.

We give a reduction from provenance verification problem to coverability in *labeled* Petri nets, where tokens carry (potentially unbounded) provenance data. As a result, we obtain a general decidability result for provenance verification problem, when the domain of provenance annotations is well-structured [1, 41]. Specifically, we show verification is EXPSPACE-complete for the set provenance domain, that tracks the set of principals that have seen a message, as well as for the language provenance domain, in which provenance information is stored as ordered sequences of principals that have seen the message and policies are regular languages. Our proofs combine well-structuredness arguments with symbolic representations; we analyze coverability in a product of a Petri net modeling the system and a symbolic domain encoding the set of allowed provenances.

While our decision procedures reduce the verification problems to problems on

Petri nets, our experiences with a direct implementation of provenance verification based on existing Petri net coverability tools have been somewhat disappointing. Mostly, this is because after the reduction to Petri nets, the coverability tools fail to utilize the structure of message passing programs, in particular potential state-space reductions arising from partial order reduction (POR) [51].

We implemented a coverability checker PROVKEEPER that is tuned for message passing programs on top of the Spin model checker [59]. Our implementation uses the expand-enlarge-check (EEC) paradigm [47]. The EEC algorithm explores a sequence of finite-state approximations of the message passing program. Intuitively, the approximation is obtained by replacing the counters in the Petri net with "abstract" counters that count precisely up to a given parameter k, and then set the count to ∞. Since the induced state space is finite for each approximation, we can use a finite-state reachability engine (such as Spin) to explore its state space. Additionally, we use partial order reduction, already implemented in Spin, to reduce the explored state space, allowing local actions of different processes to commute.

Our choice of a message passing programming model with unbounded but unordered buffers was inspired by the communication model in browser extensions, where several components communicate asynchronously. Specifically, we checked the following property of extensions. Most browsers have a "private mode" that allow users to browse the internet without saving information about pages visited. Browser extensions should respect the private mode and not save user information (or worse, upload user information to remote servers) while the user is browsing in the private mode. We checked this property and found that several widely-used Firefox extensions, including some extensions whose purpose is to improve user privacy, do not properly handle "private mode" settings. Among nine browser extensions using message passing, local storage, and sometimes remote database accesses, we found five extensions store user data even in the private mode. Thus, our experiments demonstrate that a precise static tool can be useful in detecting privacy violations in this domain.

One can view our result as a general compilation procedure from a provenance verification problem for a program P to a safety verification problem for an instrumented program P'. The instrumentation P' adds some counters to P but keeps the

```
patient {
  var p1, p2, p3;
  while (true) {
    choose
P1: [] p1 = HQ; send(ch0, p1);
P2: [] p1 = AR; send(ch0, p1);
P3: [] recv(ch1, p2);
P4:    if (p2 == HA) p3 = p2;
  }
}
```

```
secretary {
  var s1, s2;
  while (true) {
S1: recv(ch0, s1);
S2: if (s1 == HQ)
S3:   send(ch2, s1);
S4: else {
S5:   s2 = AA(s1);
S6:   send(ch1, s2);
    } }
  }
}
```

```
doctor{
  var d1, d2;
  while(true) {
D1: recv(ch2, d1);
D2: d2 = HA(d1);
D3: choose
D4: [] send(ch1, d2);
D5: [] send(ch2, d2);
  }
}
```

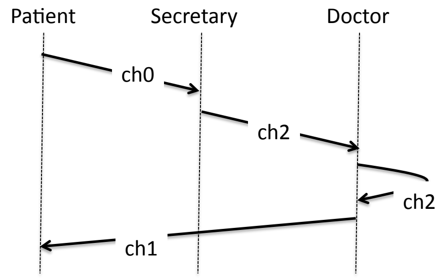


FIGURE 2.1: Medical system example

other features (e.g., complex control flow and data structures) the same: program P' is safe iff P satisfies the provenance properties. After the reduction, we can harness any verification technique that has been developed for the underlying class of programs (e.g., abstract interpretation or software model checking). Our experiments use a simple dataflow abstraction, but other abstract domains could be used for more precision. We chose message passing programs for our presentation as they capture the essence of provenance tracking: concurrency, unbounded provenance information, and unbounded channels. This focus allows us to settle the complexity of provenance verification without mixing it with the complexity of features in the programming model.

2.2 Example

We motivate our results by modeling a simple online health system described in [8], which allows patients to interact with their doctors and other healthcare professionals using a web-based message passing system. In the system, users have different roles, such as Patient, Secretary, and Doctor. Patients can ask health questions and receive answers by exchanging messages with their doctors.

For simplicity of exposition, we describe a subset of the functionality of the system as a message passing program. (In Section 2.6, we modeled the entire system as a case study.) Intuitively, a message passing program is a collection of imperative processes running concurrently, one for each principal in the system. In our example, each role (Patient, Doctor, etc.) is modeled as a different principal. The processes run by the principals have local variables, and in addition, communicate with each other by sending to and receiving from shared channels. We assume shared channels are potentially unbounded, but may reorder messages. Message sends are non-blocking, the execution continues at the control point following the send. Receives are blocking: a process blocks until some message from the channel is received.

Figure 2.1 shows a simple implementation of the system, written in a simple imperative language. We have three principals: Patient (modeling the set of patients using the system), Secretary (modeling secretaries who receive and forward messages), and Doctor (modeling the set of doctors using the system). The `choose` construct nondeterministically chooses and executes one of its branches. A `send` action sends a message to a channel, and a `recv` receives a message from a channel into a local variable.

There are four kinds of messages in the system. The patient can send a health question (HQ) or an appointment request (AR). The healthcare providers can send back a health answer (HA) or an appointment confirmation (AA). The principals communicate through shared channels `ch0`, `ch1`, and `ch2`.

The patient process runs in a loop. In each step, it nondeterministically decides to either send an HQ or an AR to `ch0`, or to receive an answer on channel `ch1`. The secretary process runs a loop. In each step, it receives a message from channel `ch0`. If it is an HQ, the message is forwarded to doctors on channel `ch2`. If it is an AR, the secretary answers the patient directly on channel `ch1`. The doctor process receives health questions on channel `ch2`. It computes a health answer based on the received message (the assignment on line `D2`). It can either reply directly to the patient (on channel `ch1`), or put the answer back to channel `ch2` for further processing.

Figure 2.1 also shows a possible message sequence for a health question, where the patient sends a health question to the secretary, the secretary forwards it to the doctor, and the doctor looks at the message several times before replying with a health answer.

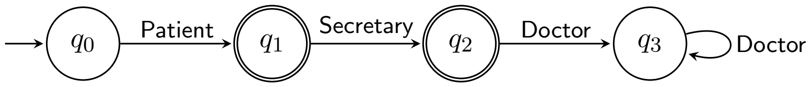


FIGURE 2.2: Complemented finite automaton for provenance property. We omit an accepting sink to which all unspecified edges go.

We capture the flow of messages through the principals using provenance annotations with each message; the provenance captures the history of all the principals that have forwarded the message. While in Section 2.3 we give a general algebraic definition of a *provenance domain*, for the moment, think of a provenance as a string over the principals. When a message is initially assigned, e.g., on line `P1`, the provenance is the empty string ε. After the patient sends the message, the channel `ch0` contains an HQ message with provenance Patient. When the message is forwarded to channel `ch2`, its provenance becomes Patient Secretary. Finally, when the message is sent back on `ch1`, its provenance is a string in the regular language Patient Secretary Doctor$^+$, indicating that it has been sent originally by the patient, seen by the secretary next, and then seen by the doctor one or more times.

The *provenance verification problem* asks, given the message passing program, a variable `v`, and a regular language R of provenances, whether the content of `v` has a provenance in R along all program executions. In the example, we can ask if the provenance of variable `p3` is in the set

$$\varepsilon + \mathsf{Patient\ Secretary\ Doctor}^+, \tag{2.1}$$

capturing the requirement that any health answer must be initiated by a health question from the patient, and must be seen by a doctor at least once, after it has been seen by a secretary.

Notice that the example is unbounded in two dimensions. First, the channels can contain unboundedly many messages. For example, the patient process can send unboundedly many messages on channel `ch0` before the secretary process receives them. Second, the provenance annotations can be unbounded: a message in channel `ch2` can have an unbounded number of Doctor annotations.

We show the provenance verification problem is decidable. The first observation

```
patient {
  var p1, p2, p3;
  while (true) {
    choose
P1′  [] p1 = ⟨HQ,q0⟩; ⟨ch0, HQ,q1⟩++;
P2′  [] p1 = ⟨AR,q0⟩; ⟨ch0, AR,q1⟩++;
P3₁  [] if ⟨ch1,HQ,q⟩ > 0 (for each q ∈ Q)
          p2 = ⟨HQ,q⟩; ⟨ch1,HQ,q⟩−;
P3₂  [] if ⟨ch1,HA,q⟩ > 0 (for each q ∈ Q)
          p2 = ⟨HA,q⟩; ⟨ch1,HA,q⟩−;
P4′     if (p2 == (HA, ·) ) p3 = p2;
P3₃  [] if ⟨ch1,AA,q⟩ > 0 (for each q ∈ Q)
          p2 = ⟨AA,q⟩; ⟨ch1,AA,q⟩−;
P3₄  [] if ⟨ch1,AR,q⟩ > 0 (for each q ∈ Q)
          p2 = ⟨AR,q⟩; ⟨ch1,AR,q⟩−;
  }
}
```

FIGURE 2.3: Translation of `patient`. We have simplified some statements for readability: the actual translation performs a case split over `p1` in lines `P1′` and `P2′`, and performs the check on line `P4′` after each statement $\texttt{P3}_i$.

is that, if we ignore provenances, we can keep a counter for each channel `ch` and each message type m, that counts the number of messages with value m that are currently in ch. A send action increases the counter, a receive decrements it. We can then show that the transition system of a message passing program is *well-structured* [1, 41]: an action that could be taken in a state can also be taken if there are more messages in the channels. Formally, we give a reduction to Petri nets, an infinite-state well-structured system with good decidability properties.

In the presence of provenances, we have to be more careful. Unlike a normal Petri net, now the "tokens" (the messages in the channels) will carry potentially unbounded provenance annotations. However, given the regular set R, we only need to distinguish two provenance annotations that behave differently with respect to a deterministic finite automaton A for R. So, we keep more counters that are now of the form $\langle \texttt{ch}, m, q\rangle$: one counter for each combination of channel `ch`, message type m, and state q of A. The state of the automaton A remembers where the automaton would go to, starting with its initial state, on seeing the provenance annotation. Similarly, for each variable in the program, we distinguish the contents of the variable based on the message type m as well as the state q of the automaton.

Figure 2.2 shows a deterministic automaton accepting the complement of the language in (2.1). Using this automaton, we describe the reduction to a well-structured system as follows. Let $Q = \{q_0, q_1, q_2, q_3, q_4\}$ be the set of states of the automaton (q_4 is the omitted sink state). We have a set of integer-valued counters $\langle \texttt{ch}i, m, q\rangle$, for $i = 0, 1, 2$, $m \in \{\texttt{HQ}, \texttt{HA}, \texttt{AA}, \texttt{AR}\}$, and $q \in Q$. For example, the counter $\langle \texttt{ch0}, \texttt{HQ}, q_1\rangle$ stores the number of HQs in ch0 for which the automaton is in state q_1. Figure 2.3 shows the translation of the `patient` process. The send actions are replaced by incrementing the appropriate counter. For example, the action `send(ch0,p1)` in line `P1` is replaced with incrementing the counter $\langle \texttt{ch0}, \texttt{HQ}, q_1\rangle$, the state of the automaton is q_1 because the principal Patient takes the automaton from its initial state q_0 to the state q_1. The receive action non-deterministically selects a non-zero counter and decrements it, while storing the message and the state into the local variable.

After the translation, we are left with a well-structured system. Verifying the provenance specification reduces to checking if there is a reachable configuration of the system in which `v` contains a message whose provenance automaton is in a final state. This reachability question can be solved as a *coverability problem* on the well-structured system, which is decidable. In fact, we show a symbolic encoding that gives an optimal algorithm.

2.3 Message Passing Programs

Preliminaries A *multiset* m over a set Σ is a function $\Sigma \to \mathbb{N}$ with finite support (i.e., $m(\sigma) \neq 0$ for finitely many $\sigma \in \Sigma$). By $\mathbb{M}[\Sigma]$ we denote the set of all multisets over Σ. As an example, we write $m = [\![\sigma_1^2, \sigma_3]\!]$ for the multiset $m \in \mathbb{M}[\{\sigma_1,\sigma_2,\sigma_3\}]$ with $m(\sigma_1) = 2, m(\sigma_2) = 0$, and $m(\sigma_3) = 1$. We write $\emptyset$ for the empty multiset, mapping each $\sigma \in \Sigma$ to 0. Two multisets are ordered by $m_1 \leq m_2$ if for all $\sigma \in \Sigma$, we have $m_1(\sigma) \leq m_2(\sigma)$. Let $m_1 \oplus m_2$ (resp. $m_1 \ominus m_2$) be the multiset that maps every element $\sigma \in \Sigma$ to $m_1(\sigma) + m_2(\sigma)$ (resp. $\max\{0, m_1(\sigma) - m_2(\sigma)\}$).

For a set X, a relation $\preceq \subseteq X \times X$ is a *well-quasi-order* (wqo) if it is reflexive, transitive, and such that for every infinite sequence $x_0, x_1, \ldots$ of elements from X, there exists

$i < j$ such that $x_i \preceq x_j$. Given a wqo $\preceq$, we define its *induced equivalence* $\equiv \subseteq X \times X$ by $x \equiv y$ if $x \preceq y$ and $y \preceq x$.

A subset X' of X is *upward closed* if for each $x \in X$, if there is a $x' \in X'$ with $x' \preceq x$ then $x \in X'$. A subset X' of X is *downward closed* if for each $x \in X$, if there is a $x' \in X'$ with $x \preceq x'$ then $x \in X'$. A function $f : X \to X$ is called $\preceq$-*monotonic* if for each $x, x' \in X$, if $x \preceq x'$ then $f(x) \preceq f(x')$.

A *transition system* $TS = (\mathcal{C}, c_0, \to)$ consists of a set $\mathcal{C}$ of configurations, an initial configuration $c_0 \in \mathcal{C}$, and a transition relation $\to \subseteq \mathcal{C} \times \mathcal{C}$. We write $\to^*$ for the reflexive transitive closure of $\to$. A configuration $c \in \mathcal{C}$ is *reachable* if $c_0 \to^* c$. A *well-structured transition system* is a $TS = (\mathcal{C}, c_0, \to)$ equipped with a well-quasi order $\preceq \subseteq \mathcal{C} \times \mathcal{C}$ such that for all $c_1, c_2, c_3 \in \mathcal{C}$ with $c_1 \preceq c_2$ and $c_1 \to c_3$, there exists $c_4 \in \mathcal{C}$ with $c_3 \preceq c_4$ and $c_2 \to c_4$.

2.3.1 Programming Model

Syntax We work in the setting of asynchronous message passing programs. For simplicity, we assume that the programming language has a single finitely-valued datatype $\mathcal{M}$ of messages. A *channel* is a (potentially unbounded) multiset of messages supporting two actions: a *send* action (written $ch!x$) that takes a message stored in variable x and puts it into the channel, and a *receive* action (written $ch?x$) that takes a message m from the channel and copies it to the variable x. Let C be a finite set of channels.

A *control flow graph* (CFG) $G = (X, V, E, v^0)$ consists of a set X of message variables, a set V of control locations including a unique *start location* $v^0 \in V$, and a set E of labeled directed edges between the control locations in V. Every edge in E is labeled with one of the following actions:

- an *assignment* $y := \otimes(x)$, where $x, y \in X$ and $\otimes$ is an uninterpreted unary operation on messages;
- an *assume action* $\mathsf{assume}(x = m)$, where $x \in X$ and $m \in \mathcal{M}$;
- a *send action* $ch!x$, or a *receive action* $ch?x$, where $x \in X$ and $ch \in C$.

A *message passing program* $\mathcal{P} = (\mathit{Prin}, C, \{G_p\}_{p \in \mathit{Prin}})$ consists of a finite set Prin of *principals*, a set C of channels, and for each $p \in \mathit{Prin}$, a control flow graph G_p.

Intuitively, a message passing program consists of a finite set of processes. Each process is owned by a named entity or a principal. The processes have local variables which can be updated using unary operators, and communicate with other processes by asynchronously sending to and receiving messages from the set of channels C.

We shall use the notation $v \xrightarrow{a,p} v'$ to denote that the CFG G_p of principal p has an edge $(v, v') \in E_p$ labeled with the action a. Given the set $\{G_p\}_{p \in Prin}$ of CFGs, we define $X^\star = \uplus\{X_p \mid p \in Prin\}$, $V^\star = \uplus\{V_p \mid p \in Prin\}$, and $E^\star = \uplus\{E_p \mid p \in Prin\}$ as the disjoint unions of local variables, control locations, and control flow edges, respectively.

Semantics We now give a *provenance-carrying* semantics to message passing programs. Let U be a (not necessarily finite) set of *provenances*. We shall associate with each message in a message passing program a provenance from U.

Let $\mathcal{P} = (Prin, C, \{G_p\}_{p \in Prin})$ be a message passing program. A *provenance domain* $\mathcal{U} = (U, \preceq, \psi)$ for $\mathcal{P}$ consists of a set U of provenances, a well-quasi ordering $\preceq$ on U, and for each principal $p \in Prin$ and for each operation $op \in \otimes \cup \{!, ?\}$, a $\preceq$-monotonic function $\psi(p, op) : U \to U$. A provenance domain is decidable if $\preceq$ is a decidable relation and ψ is a computable function. We assume all provenance domains below are decidable.

Since channels are unordered, we represent contents of a channel as a multiset of pairs of messages and provenances. A *configuration* $(\ell, \mathbf{c}, \pi)$ consists of a location function $\ell : Prin \to V^\star$ mapping each principal to a control location; a channel function $\mathbf{c} : C \to \mathbb{M}[\mathcal{M} \times U]$ mapping each channel to a multiset of pairs of messages from $\mathcal{M}$ and provenances from U; and a store function $\pi : X^\star \to \mathcal{M} \times U$ mapping each variable to a message and its provenance.

Define $\ell_0 : Prin \to V^\star$ as the function mapping $p \in Prin$ to the start location $v_p^0 \in V_p$ and $\mathbf{c}_0 : C \to \mathbb{M}[\mathcal{M} \times U]$ as the function mapping each $ch \in C$ to the empty multiset $\emptyset$. Let $\pi_0 : X^\star \to \mathcal{M} \times U$ be a mapping from variables in $X^\star$ to a default initial value m_0 from $\mathcal{M}$ and a default initial provenance ε from U.

The provenance-carrying semantics of a message passing program $\mathcal{P}$ with respect to the provenance domain $(U, \preceq, \psi)$ is defined as the transition system $TS(\mathcal{P}) = (\mathcal{C}, c_0, \to)$

where $\mathcal{C}$ is the set of configurations, the initial configuration $c_0 = (\ell_0, \mathbf{c}_0, \pi_0)$, and the transition relation $\rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is defined as follows.

For a function $f : A \rightarrow B$, $a \in A$, and $b \in B$, let $f[a \mapsto b]$ denote the function that maps a to b and all $a' \neq a$ to $f(a')$. We define $(\ell, \mathbf{c}, \pi) \rightarrow (\ell', \mathbf{c}', \pi')$ if there exists $p \in \mathit{Prin}$ and $(\ell(p), a, \ell'(p)) \in E^{\bigstar}$ such that for all $p' \neq p$, we have $\ell(p') = \ell'(p')$; and

1. if $a \equiv y := \otimes(x)$ and $(m, u) = \pi(x)$ then $\mathbf{c}' = \mathbf{c}$ and $\pi' = \pi[y \mapsto (\otimes(m), \psi(p, \otimes)(u))]$;
2. if $a \equiv \mathsf{assume}(x = m)$ then $\mathbf{c}' = \mathbf{c}$, $\pi' = \pi$, and $\pi(x) = (m, \cdot)$;
3. if $a \equiv ch!x$ then $\pi'{=}\pi$ and if $(m, u){=}\pi(x)$, then $\mathbf{c}' = \mathbf{c}[ch \mapsto \mathbf{c}(ch) \oplus [\![(m, \psi(p, !)(u))]\!]]$;
4. if $a \equiv ch?x$ and there is (m, u) such that $\mathbf{c}(ch)(m, u) > 0$ then $\mathbf{c}' = \mathbf{c}[ch \mapsto \mathbf{c}(ch) \ominus [\![(m, u)]\!]]$ and $\pi' = \pi[x \mapsto (m, \psi(p, ?)(u))]$.

Intuitively, in each step, one of the principals executes a local action. An assignment action $y := \otimes(x)$ transforms the message contained in x by applying the operation $\otimes$ and transforms the provenance of x by applying ψ, storing the new message and its provenance in y. An assume checks that a variable has a specific message. Sends and receives model asynchronous communication to shared channels. Send actions are non-blocking, receive actions are blocking, and a channel can reorder messages.

Let $\mathcal{P}$ be a message passing program and $\mathcal{U} = (U, \preceq, \psi)$ a provenance domain. We consider provenance specifications given by downward closed sets over U. Downward closed sets capture the "monotonicity" property that holds in many domains. For example, a security policy that holds when a given set of trusted principals looks at a message, is also met when fewer principals look at it. Conversely, bad behaviors are captured by upward closed sets.

The *provenance verification problem* asks, given a variable x of $\mathcal{P}$ and a downward closed set $D \subseteq U$, if the provenance of the content of variable x is always in D along all runs of the program. Dually, the specification is violated if there exists a reachable configuration where the provenance of variable x is in the upward closed set $I = U \backslash D$. Such a configuration indicates a violation of security policies. We shall use the dual formulation in our algorithms.

2.3.2 Examples

We now give illustrative examples of provenance domains.

Example 1. [The Language Provenance Domain] *Consider $U = Prin^*$, the set of finite sequences over principals. Let $(Q, Prin, q_0, \delta)$ be a deterministic finite automaton, and let $\preceq$ be defined as $u \preceq v$ iff $\delta(q_0, u) = \delta(q_0, v)$. Let ψ be the function defined as $\psi(p, !)(u) = u \cdot p$, and $\psi(\cdot, \cdot)(u) = u$ for all other operations. Intuitively, the language provenance domain associates a list of principals with each message: the sequence of principals who have sent this message along the current computation.*

A downward closed set D in the language provenance domain is a regular language that prescribes a set $F \subseteq Q$ of final states for the finite automaton A. The corresponding upward closed set I is a regular language that prescribes a set $Q \setminus F$ of final states for the complement automaton $\overline{A}$. The provenance verification problem asks, for example, if the provenance of the message in p3 *always belongs to the regular language* Patient Secretary Doctor$^+$ *along all runs of the program.*

Example 2. [The Set Provenance Domain] *Let $U = 2^{Prin}$, the set of sets of principals. Let $\preceq$ be set inclusion. Since the set of principals is finite, this is a wqo. Let ψ be the function defined as $\psi(p, !)(u) = u \cup \{p\}$, and $\psi(\cdot, \cdot)(u) = u$ for all other operations. The set provenance domain associates a set of principals with each message: the set contains all the principals who have sent this message (potentially multiple times). An upward closed set I corresponds to a set of sets of principals, such that if a set of principals is in I, each of its supersets is also in I. As an example, suppose the set of principals $Prin$ is divided into "trusted" and "untrusted" principals. A downward closed set D specifies the sets all of whose elements are "trusted". As a result, the corresponding upward closed set I captures all sets containing at least one "untrusted" principal. The provenance verification problem asks, given a variable x, if there is a message stored in x along a run that has a provenance which is one of the sets in I.*

2.4 Model Checking

We now give a model checking algorithm for provenance verification by reduction to labeled Petri nets.

2.4.1 Labeled Petri Nets

A *Petri net* (PN) is a tuple $N = \langle S, T, (I, O) \rangle$ where S is a finite set of places, T is a finite set of transitions, and functions $I : T \to S \to \{0, 1\}$ and $O : T \to S \to \{0, 1\}$ encodes pre- and post-conditions of transitions.

A *marking* is a multiset over S. A transition $t \in T$ is *enabled* at a marking μ, denoted by $\mu[t\rangle$, if $\mu \geq I(t)$. An enabled transition t at μ may *fire* to produce a new marking μ', denoted by $\mu[t\rangle\mu'$, where $\mu' = \mu \ominus I(t) \oplus O(t)$. We naturally lift the enabledness and firing notions from one transition to a sequence $\sigma \in T^*$ of transitions.

A PN N and a marking μ_0 define a transition system $TS(N) = (\mathbb{M}[S], \mu_0, \to)$, where $\mu \to \mu'$ if there is a transition t such that $\mu[t\rangle\mu'$.

The encoding of a PN N is given by a list of pairs of lists. Each transition $t \in T$ is encoded by two lists corresponding to $I(t)$ and $O(t)$. Each list $I(t)$ or $O(t)$ is encoded as a bitvector of size $|S|$. The size of N, written $\|N\|$, is the sum of the representations of all the lists.

Let N be a Petri net and μ_0 and μ markings. The *coverability problem* asks if there is $\mu' \geq \mu$ that is reachable from μ_0, so $\mu_0 \to^* \mu' \geq \mu$. In this case, we say μ is coverable from μ_0.

Theorem 1. *[82, 106] The coverability problem for Petri nets is EXPSPACE-complete.*

In the usual definition of Petri nets, tokens are simply uninterpreted "dots" and markings count the number of dots in each place. We now extend the Petri net model with tokens labeled with elements from a decidable provenance domain $\mathcal{U}$. A $\mathcal{U}$-labeled Petri net $N = \langle S, T, (I, O), \Lambda \rangle$ is a Petri net $\langle S, T, (I, O) \rangle$ that is equipped with a labeling function Λ specifying how provenance markings are updated when a transition is fired. Consider a transition $t \in T$. Let $p_1, \ldots, p_k$ be an ordering of all the places in S for which $I(t)(p) = 1$. For each place $p' \in S$ with $O(t)(p') = 1$, the labeling function $\Lambda(t, p')$ is a $\preceq$-monotonic function $U^k \to U$. We assume the labeling function Λ is computable.

A *labeled marking* μ is a mapping from places S to multisets over U, i.e., it labels each token in a marking with an element of U. A labeled marking μ induces a marking $\mathsf{erase}(\mu)$ that maps each $p \in S$ to $\sum_{u \in U} \mu(p)(u)$ obtained by erasing all provenance information carried by tokens. Fix a transition t, and let $p_1, \ldots, p_k$ be an ordering of the places such that $I(t)(p) = 1$. The transition t is enabled at a labeled marking μ if for each $p \in S$ with $I(t)(p) = 1$, we have $\mathsf{erase}(\mu)(p) \geq 1$. An enabled transition t at μ can fire to produce a new labeled marking μ', denoted (by abuse of notation) $\mu[t\rangle\mu'$, defined as follows. To compute μ' from μ, first pick and remove arbitrarily tokens from p_1 to p_k with labels u_1 to u_k respectively. Then, for each p' with $O(t)(p') = 1$, add a token whose label is $\Lambda(t, p')(u_1, \ldots, u_k)$ to p'. All other places remain unchanged. We extend the firing notion to sequences of transitions, as well as notions of transition system, size, reachability, and coverability to labeled Petri nets in the obvious way.

To prove the coverability problem is decidable for $\mathcal{U}$-labeled Petri nets, we argue that their transition systems $(\mathbb{M}[U]^S, \mu_0, \hookrightarrow)$ are *well-structured* in that the labeled markings can be equipped with an order that allows larger labeled markings to mimic the behaviour of smaller ones, i.e. there is a wqo $\ll\ \subseteq \mathbb{M}[U]^S \times \mathbb{M}[U]^S$ that is compatible with the transitions: for all $\mu_1 \hookrightarrow \mu_1'$ and $\mu_1 \ll \mu_2$ there is $\mu_2 \hookrightarrow \mu_2'$ so that $\mu_1' \ll \mu_2'$.

To define a suitable wqo on labeled markings, we first compare the multisets on a place. Intuitively, $\mu(p) \ll \mu'(p)$ with $\mu, \mu' \in \mathbb{M}[U]^S$ and $p \in S$ if for every u in $\mu(p)$ there is an element u' in $\mu'(p)$ such that $u \preceq u'$ in the wqo $\preceq$ of the provenance domain. Hence, $\mu \ll \mu'$ if for each $p \in S$ there is an injective function $f_p : \mu(p) \to \mu'(p)$ so that for each $u \in \mu(p)$, we have $u \preceq f_p(u)$. The result is a wqo by Higman's lemma [57] and the fact that wqos are stable under Cartesian products. The ordering is also compatible with the transitions by the monotonicity requirement on labelings. The following theorem follows using standard results on well-structured transition systems [1, 41].

Theorem 2. *The coverability problem for $\mathcal{U}$-labeled Petri nets is decidable and EXPSPACE-hard for decidable provenance domains $\mathcal{U}$.*

The coverability problem for labeled Petri nets need not be in EXPSPACE, even when the operations on $\mathcal{U}$ are provided by an oracle. For example, nested Petri nets [85] can encode reset nets, for which a non-primitive recursive lower bound is known for coverability [110].

2.4.2 From Message Passing Programs to Labeled Petri Nets

Let $\mathcal{P} = (Prin, C, \{G_p\}_{p \in Prin})$ be a message passing program and $\mathcal{U} = (U, \preceq, \psi)$ a provenance domain. We now give a labeled Petri net semantics to the program.

Define the labeled Petri net $N(\mathcal{P},\mathcal{U}) = \langle S, T, (I, O), \Lambda \rangle$ as follows. There is a place for each program location, for each local variable and message value, and each channel and message value: $S = V^\bigstar \cup (X^\bigstar \times \mathcal{M}) \cup (C \times \mathcal{M})$.

In the definition of labels, we use variable $\mathsf{prov}(p)$ for the token (which is a provenance) in place $p \in S$ that is used for firing. The set T is the smallest set that satisfies the following conditions.

1. For each $e \equiv v \xrightarrow{y:=\otimes(x),p} v'$ in $E^\bigstar$, and for each $m, m' \in \mathcal{M}$, there is a transition t with $I(t) = [\![v, (x, m), (y, m')]\!]$ and $O(t) = [\![v', (x, m), (y, \otimes m)]\!]$. Also, $\Lambda(t, (x, m)) = \mathsf{prov}(x, m)$, $\Lambda(t, (y, \otimes m)) = \psi(p, \otimes)(\mathsf{prov}(x, m))$, and $\Lambda(t, v') = \varepsilon$.
2. For each $e \equiv v \xrightarrow{\mathsf{assume}(x=m),p} v'$ in $E^\bigstar$, there is a transition t with $I(t) = [\![v, (x, m)]\!]$ and $O(t) = [\![v', (x, m)]\!]$. Also, $\Lambda(t, v') = \varepsilon$, and $\Lambda(t, (x, m)) = \mathsf{prov}(x, m)$.
3. For each $e \equiv v \xrightarrow{ch!x,p} v'$ in $E^\bigstar$, and for each $m \in \mathcal{M}$, there is a transition t with $I(t) = [\![v, (x, m)]\!]$, $O(t) = [\![v', (x, m), (ch, m)]\!]$. Also, $\Lambda(t, v') = \varepsilon$, $\Lambda(t, (x, m)) = \mathsf{prov}(x, m)$, and $\Lambda(t, (ch, m)) = \psi(p, !)(\mathsf{prov}(x, m))$.
4. For each $e \equiv v \xrightarrow{ch?x,p} v'$ in $E^\bigstar$, for each $m, m' \in \mathcal{M}$, there is a transition t with $I(t) = [\![v, (x, m), (ch, m')]\!]$ and $O(t) = [\![v', (x, m')]\!]$. Also, $\Lambda(t, v') = \varepsilon$ and $\Lambda(t, (x, m')) = \psi(p, ?)(\mathsf{prov}(ch, m'))$.

To relate $\mathcal{P}$ with its Petri nets semantics $N(\mathcal{P},\mathcal{U})$, we define a bijection ι between configurations and labeled markings: $\iota(\ell, \mathbf{c}, \pi) = \mu$ iff all of the three conditions hold: (1) $\mu(v) = [\![\varepsilon]\!]$ iff there is $p \in Prin$ with $\ell(p) = v$; (2) for all $x \in X^\bigstar$, for all $m \in \mathcal{M}$, and for all $u \in U$, $\mu(x, m) = [\![u]\!]$ iff $\pi(x) = (m, u)$; (3) for all $ch \in C$, for all $m \in \mathcal{M}$, and for all $u \in U$, $\mu(ch, m)(u) = k$ iff $\mathbf{c}(ch)(m, u) = k$. Define the initial labeled marking $\mu_0 = \iota(\ell_0, \mathbf{c}_0, \pi_0)$. The following observation follows from the definition of ι.

Lemma 1. *$TS(\mathcal{P})$ and $TS(N(\mathcal{P},\mathcal{U}))$ are isomorphic.*

Complexity-wise, the problem inherits the hardness of coverability in (unlabeled) Petri nets for any non-trivial provenance domain.

Theorem 3. *Given a message passing program $\mathcal{P}$ and a decidable provenance domain $\mathcal{U} = (U, \preceq, \psi)$, the provenance verification problem is decidable. It is EXPSPACE-hard for any provenance domain with at least two elements.*

Proof. From the construction of the labeled Petri net, Lemma 1, the provenance verification problem is reducible in polynomial time to coverability for labeled Petri nets. Thus, by Theorem 2, provenance verification problem is decidable.

For EXPSPACE-hardness, we reduce Petri net coverability to provenance verification. To simulate a Petri net with a message passing program, we introduce a channel for every place and then serialize the reading of tokens. Consider $N = \langle S, T, (I, O) \rangle$. We construct a message passing program with one principal, one message, and a channel for each place in S. The control flow graph of the only principal has a central node from which loops simulate the Petri net transitions. At each step, the central node picks a transition $t \in T$ non-deterministically and simulates first the consumption and then the production of tokens — one by one. To consume a token from place p with $I(t)(p) = 1$, the principal receives a message from channel p. For the production, it sends a message to the channel p' with $O(t)(p') = 1$. Additionally, the principal non-deterministically checks if the current configuration of channels covers the target marking. If so, it writes a message into a special variable x. The provenance verification problem asks whether x ever contains a message with non-trivial provenance. EXPSPACE-hardness follows from Theorem 1. □

2.5 EXPSPACE Upper Bounds

For set and language provenance domains, we can in fact show a matching upper bound on the complexity. It relies on a fairly general product construction and reduction to Petri nets. We say that a provenance domain $\mathcal{U}$ is of *finite index* if the equivalence induced by $\preceq$ has finitely many classes. We denote this equivalence by $\equiv$. Clearly, any finite provenance domain (thus, the set domain) is of finite index. The language domain is also of finite index: take the equivalence relation induced by the Myhill-Nerode classes of the language. The following lemma characterizes the structural properties of provenance domains of finite index.

Lemma 2. *Consider a Petri net $N = \langle S, T, (I, O), \Lambda\rangle$ that is labelled by $\mathcal{U}$ of finite index. (1) The equivalence classes are closed under Λ: for any tuple $e_1, \ldots, e_k$ of $\equiv$-equivalence classes, the image $\Lambda(e_1, \ldots, e_k)$ is fully contained in another equivalence class e. (2) The upward-closure of any $u \in U$ is a finite union of $\equiv$-classes.*

Let $N = \langle S, T, (I, O), \Lambda\rangle$ be a $\mathcal{U}$-labeled Petri net, and suppose $\mathcal{U}$ is of finite index. We now define a product construction that reduces N to an ordinary Petri net $N' = \langle S', T', (I', O')\rangle$. Intuitively, for each place $p \in S$ and each equivalence class e, there is a place (p, e) in S' that keeps track of all tokens in N at place p and having their label in the equivalence class e. We define $S' = S \times \{[u]_{\equiv} \mid u \in U\}$. Each transition in N is simulated by a family of transitions in T', one for each combination of equivalence classes for the source tokens. More precisely, T' is the smallest set that contains the following family of transitions for each $t \in T$. Let $p_1, \ldots, p_k$ be the places in S with $I(t)(p_i) = 1$. For each sequence $\overline{p} = \langle e_1, \ldots, e_k\rangle$ of k-tuples of $\equiv$-equivalence classes, we have a transition $t_{\overline{p}} \in T'$ such that $I'(t_{\overline{p}})((p_i, e_i)) = 1$ for $i = 1, \ldots, k$ and $I'(t_{\overline{p}})(p) = 0$ for all other places. Moreover, for each $p \in S$ with $O(t)(p) = 1$ labeled with Λ, we have that $O'(t_{\overline{p}})((p, e)) = 1$ with $\Lambda(e_1, \ldots, e_k) \subseteq e$. Note that this inclusion is well-defined by Lemma 2(1). This product construction reduces a labelled coverability query in N to several unlabelled queries in N'. What are the unlabelled queries we need? Consider a token u in a labelled marking $\mu \in \mathbb{M}[U]^S$. We use the equivalence classes that, with Lemma 2(2), characterize the upward closure of u. In the following proposition, we assume that these classes are effectively computable. This is the case for set and language domains.

Proposition 1. *If $\mathcal{U}$ is of finite index, coverability for $\mathcal{U}$-labeled Petri nets is reducible to coverability for Petri nets.*

Proposition 1 provides a 2EXPSPACE upper bound for the set and language domains, which is not optimal. Consider the set domain. Each subset of principals yields an equivalence class of provenances. Hence, there is an exponential number of classes and the above product net is exponential. A similar problem occurs for the language domain if the provenance specification is given by a non-deterministic finite automaton. There are regular languages where this non-deterministic representation is exponentially more succinct than any deterministic one. The deterministic one, however, is needed in the product. To derive an optimal upper bound, we give compact representations of these exponentially many classes.

Theorem 4. *Provenance verification problem is in EXPSPACE for set and language domains.*

Proof. To establish membership in EXPSPACE, we implement the above reduction from labeled to unlabeled coverability in a compact way, so that the size of the resulting Petri net is polynomial in the size of the input. The challenge is to avoid the multiplication between places and equivalence classes, which may be exponential. Instead, we first encode the classes into polynomially many additional places, and maintain the relationship between a place and a class in the marking of the new net. Second, we only keep the provenance information for tokens in the goal marking, and omit the provenance of the remaining tokens.

Let E be the set of equivalence classes of a provenance domain of finite index. Let $\kappa = \lceil \log |E| \rceil$. The symbolic representation of E uses 2κ places. Let the places be $b_0, d_0, \ldots, b_{\kappa-1}, d_{\kappa-1}$. We maintain the invariant that in any reachable marking, exactly one of b_i, d_i contains a single token, for $i = 0, \ldots, (\kappa - 1)$. Intuitively, a token in b_i specifies the bit i is one, and a token in d_i specifies the bit i is zero. Using constructions on (1-safe) Petri nets, one can "copy" a bitvector, remove all tokens from a bitvector, or update a bitvector to a value.

For example, to empty out a bitvector, we introduce $\kappa + 1$ places $p_0, \ldots, p_\kappa$, with an initial token in p_0. Each p_i, $i \in \{0, \ldots, \kappa - 1\}$, has two transitions: they take a token from p_i and from b_i (resp. d_i), and put a token in p_{i+1}. When p_κ is marked, all the bits have been cleared. Similarly, to copy the configuration from places $b_0, d_0, \ldots, b_{\kappa-1}, d_{\kappa-1}$ to empty places $b'_0, d'_0, \ldots, b'_{\kappa-1}, d'_{\kappa-1}$, we use the following gadget. We add additional $\kappa + 1$ places $p_0, \ldots, p_\kappa$, with an initial token on p_0. For each p_i, $i \in \{0, \ldots, \kappa - 1\}$ there are two transitions: one takes a token from p_i and one token from b_i and puts a token in p_{i+1}, one in b_i, and one in b'_i; the other takes a token from p_i and one from d_i and puts a token in p_{i+1}, one in d_i, and one in d'_i. When the place p_κ is marked, the bits in $b_0, d_0, \dots, b_{\kappa-1}, d_{\kappa-1}$ have been copied to $b'_0, d'_0, \dots, b'_{\kappa-1}, d'_{\kappa-1}$.

Now, in the translation of the Petri net, instead of a place (x, m, e) for each variable x, message m, and equivalence class $e \in E$, we keep 2κ places for each place (x, m), encoding the equivalence class e for x and m. If all 2κ places for (x, m) are empty in a marking, it implies that the current content of x is not m; otherwise, the provenance equivalence class $e \in E$ of (x, m) is encoded by the 2κ bits. The transitions of the net are updated with the gadgets to copy the provenance bitvectors in case of assignments.

Moreover, for each channel ch, we maintain the provenance information of one message, and drop the provenance of every other message in the channel. That is, each channel ch is modeled using places (ch, m) for each $m \in \mathcal{M}$, and in addition, $2\kappa \cdot |\mathcal{M}|$ places that encode the provenance equivalence class of one message for each value in $\mathcal{M}$ stored in the channel. Intuitively, tokens in (ch, m) denote messages with value m in the channel ch whose provenance has been "forgotten" and tokens in the bitvectors encode one message (per message type) in the channel whose provenance is encoded using 2κ places. We use non-determinism to guess which messages contribute to the message with provenance in the target. When a message is sent to a channel, we non-deterministically decide to keep its provenance (thus using the bitvectors, moving any tokens already there) or to drop its provenance.

Similarly, when we receive from a channel, we non-deterministically decide to either read from the "special" places for the encoding of an equivalence class, or from the "normal" place.

Now, for the set domain, we use $2|Prin|$ places to encode sets of principals. For the language domain, where the specification is given by a non-deterministic automaton with states Q, we use $2|Q|$ places to encode the subsets of states. The encoding allows us to perform the subset construction on the fly. Each action of the program requires at most a polynomial number of additional places to encode the gadgets. Thus, we get a Petri net that is polynomial in the size of the message passing program and the specification. Thus, using Theorem 1, we get the EXPSPACE upper bound. □

2.6 Implementation and Experiments

We have implemented PROVKEEPER, a verifer for the provenance verification problem for language provenance domains. PROVKEEPER takes as input a message passing program encoded in an extended Promela syntax in which channels are marked asynchronous and have the semantics described in Section 2.3. It reduces the provenance verification problem to Petri net coverability using the algorithm from Section 2.4. We first used state-of-the-art tools for Petri net coverability [44, 93]. Unfortunately, the times taken to verify the provenance properties were high. This is because Petri net coverability tools are optimized for nets with many places that can be unbounded and for high concurrency. Instead, message passing programs only have few places that are unbounded (the channels). Our second observation is that message passing programs have a lot of scope for partial-order reduction, by allowing a process to continue executing until it hits a blocking receive action. To take advantage of these features, we implemented PROVKEEPER that combines expand-enlarge-check (EEC) [47] with partial order reduction [51].

2.6.1 Expand-Enlarge-Check and Partial Order Reduction

The EEC procedure [47] performs *counter abstraction* over a Petri net. We observe that only the places representing shared channels can have more than one token in our Petri nets. Instead of counting the exact number of messages in a channel, we fix a parameter $k \geq 0$ and count precisely up to k. If at any point, the number of messages in a channel exceeds k, we replace the number by ∞. Once the count goes to ∞, we do not decrease the count even when messages are removed from the channel. For example, if $k = 0$, the abstraction of a channel distinguishes two cases: either the channel has no messages or it has an arbitrary number of messages.

The abstraction is sound, in that if a marking is coverable in the original net, it is also covered in the abstraction. However, the abstraction can add spurious counterexamples, in that a marking can be considered coverable in the abstraction, even though it is not coverable in the original net. By concretely simulating a specific counterexample path, we can decide if the counterexample is genuine or spurious. In case the counterexample is spurious, we increase the parameter k and continue. This abstraction-refinement process is guaranteed to terminate, by either finding a genuine path that covers a given marking, or by proving that the target marking is not coverable for some parameter k in the abstraction [47]. We have found that $k = 1$ is usually sufficient to soundly abstract the state space and to prove a provenance property; this is consistent with other uses of counter abstractions in verification [64, 101].

Additionally, we note that once the parameter k is fixed, the state space of the system is finite, since each channel can have at most $k + 2$ messages ($\{0, \ldots, k\} \cup \{\infty\}$). Thus, for each k, we can perform reachability analysis using a finite-state reachability engine. The implementation of ProvKeeper uses the Spin model checker [59] to perform reachability analysis in every iteration where k is fixed. In Spin models, for each channel, each message type, and each state of the provenance automaton, we have a variable that takes $k + 2$ values, implementing the k-abstraction.

Additionally, message passing programs have the potential for partial order reduction. For example, each process in the program can be executed until it reaches a blocking receive action, and the local actions of different processes commute. Since Spin already implements partial order reduction, we get the benefits of partial order reduction for free.

2.6.2 Case Studies: Message Passing Benchmarks

We first describe our evaluation on a set of three message passing systems (see Table 2.1). The example MyHealth Portal is described in [8]. We checked if the provenance of a variable is always in the regular language $\mathsf{Patient}\ (\mathsf{Secretary} + \varepsilon)\ \mathsf{Nurse}\ \mathsf{Doctor}^{+} + \varepsilon$. The bug tracking system [63] manages software bug reports. It has five principals and eight types of messages (bug report, closed, fix-again, fix, must-fix, more-information, pending, and verified). The provenance specification, given as an automaton with nine states, encodes the flow of events leading from a bug report to a bug fix. We found that the original system violated the specification because a message was sent to an incorrect channel. After fixing the bug, we were able to prove the property for the new system. The Service Incident Exchange Standard (SIS) specifies a

Example	Principals	Messages	Channels	Automaton
Health Care	4	4	5	6
Bug Tracking	5	8	5	9
SIS	16	9	18	2

TABLE 2.1: Message passing benchmarks. "Principals" is the number of principals, "Messages" the possible values of messages, "Channels" is the number of shared channels, and "Automaton" is the number of states in the provenance automaton.

system to share service incident data and facilitate resolutions. The standard envisages interactions between service requesters and providers. We took the system model from [23], which consists of 16 principals, 18 channels, and 9 message types. The property to check is once a service request is terminated, it is never reopened.

Results Table 2.2 lists the analysis results. All experiments were performed on a 2 core Intel Xeon X5650 CPU machine with 64GB memory and 64bit Linux (Debian/Lenny). We compare state-of-the-art Petri net coverability tools (Mist2 [44] and Petruchio [93]) with PROVKEEPER. We run Petruchio and three different options of Mist2 and report the best times. A timeout indicates that all the tools timed out. The "Markings" row indicates the number of coverability checks required to prove correctness. The time denotes the sum of the times for all the coverability checks to finish, where for each check, we take the best time by any tool.

For PROVKEEPER, we report the parameter k for which either a genuine counterexample was found, or the system was proved correct. We compare the results with and without partial order reduction. For each run, we give three numbers: the number of states and transitions explored by our checker and the time taken. There is a significant reduction when partial order reduction is turned on. Moreover, PROVKEEPER is orders of magnitude faster than the Petri net coverability tools.

2.6.3 Private Mode and Firefox Extensions

We performed a larger case study on provenance in browser extensions. Modern browsers provide a "private mode" that deletes cookies, forms, and browsing history at the end of each browsing session. Browsers also provide an extension mechanism, through which third-party developers can add functionality to browsers. Extensions

PN tools	Health Care	Bug Tracking (1)	Bug Tracking (2)	SIS
Markings	12	1	40	127
Time	125.6s	2308.940s	timeout	1152.07s

ProvKeeper	Health Care	Bug Tracking (1)	Bug Tracking (2)	SIS
k	0	1	0	1
States (No POR)	6351	39	4905516	3738754
States (POR)	2490	39	995468	893786
Trans (No POR)	23357	39	24850365	17274836
Trans (POR)	4249	39	1707682	1736062
Time (No POR)	0.04s	0.01s	38.6s	58.7s
Time (POR)	0.01s	0.01s	3.37s	6.10s

TABLE 2.2: Results of the message passing benchmarks. Bug Tracking (1) is the buggy version.

can communicate between their front- and back-ends by asynchronous messages passing, and between each other via temporary files. Moreover, Firefox lets extension developers manage SQLite databases in user machines by invoking a service called *mozIStorageService*. It provides a set of asynchronous APIs for extensions to communicate with databases through SQL queries. If extension developers do not properly handle the private mode, user data may be stored in the database while the user is browsing in private mode.

It is expected that browser extensions should respect the private mode. Unfortunately, browsers do not restrict an extension's capability in private mode, and it is the responsibility of developers not to record user data in private mode. In the second set of case studies, we check if extension developers for Firefox obey the privacy concerns when the user is browsing in private mode.

Our goal is to check if extensions using *mozIStorageService* can store user data while in private mode. We formulate the problem of tracking information flow in private mode as a provenance verification problem. Consider a set of browser extensions cooperating with each other, and a principal Db modelling a database. For each extension A, we introduce two principals NormA and PrivA that represent two instances of A running in the normal and in the private mode, respectively. For each extension A that saves data to the database, there are two channels $ch_{\mathsf{Db}}, ch'_{\mathsf{Db}}$ for NormA and PrivA to interact with Db. Moreover, for each pair of extensions (A, B) where A sends data to B, for instance, by writing and reading files, there are four combinations:

$(\mathsf{NormA}, \mathsf{NormB})$, $(\mathsf{PrivA}, \mathsf{NormB})$, $(\mathsf{NormA}, \mathsf{PrivB})$, and $(\mathsf{PrivA}, \mathsf{PrivB})$. For each case, we introduce a channel ch to model the message flow from A to B. The property we check is whether some PrivA directly or indirectly updates the database. Note that it is not sufficient to ensure every write to the database is guarded by a check that the browser is not in private mode. There can be indirect flows where data is stored in a temporary file in private mode, or communicated to a different extension, and later stored in the database.

We use Firefox 13.0.1 in our experiments. We selected nine popular extensions from Firefox's extension repository, by filtering them based on the keywords *form*, *history*, and *shopping*, and then filtering based on their use of *mozIStorageService*. The extensions we chose have about 50000 users on average.

The workflow of the verification is as follows. We first use JSure [38], a Javascript parser and static analyzer, to obtain the control flow from the extension source code, and to produce a message passing program in Promela syntax. As the access to a database is either via calling the *mozIStorageService* APIs directly or via helper extensions, we capture along the control flow the information about when an extension calls these APIs to update the database, and the information about when extensions communicate with each other by writing and reading temporary files. Our front end abstracts away complex data structures in the program. In particular, we do not track the contents inserted into the database. This may lead to false positives in the analysis. We then run PROVKEEPER to verify the message passing program.

Table 2.3 lists the results. Five out of the nine examples are found to store user information even in private mode. All examples can be verified efficiently (in a few milliseconds) because usually a small portion of code is related to database accesses and extension communications, and complex data structures are abstracted out. For all unsafe cases, we have successfully replayed executions that violate the private mode in Firefox.

2.7 Related Work

Provenance annotation on data has been studied extensively in the database community [18, 33, 54], both for annotating query results and for tracking information through workflows. Provenance information is usually tracked for a fixed database and a fixed query in a declarative query language. Seen as a program, the query has exactly one "execution path." The connection between provenance tracking and dependency analysis in (sequential) programs was made in [26]. A provenance-tracking semantics for asynchronous π-calculus was given in [115], but the static analysis problem was not considered. Most previous work focused on dynamic tracking and enforcement along one execution path, and the static meet-over-all-paths solution was not considered. In contrast, we provide algorithms to track provenances in concurrent message passing programs, and give algorithms to check provenance queries over all execution paths of programs. We were inspired by the algebraic framework of provenance semirings [54] to give a similar algebraic description of provenance domains.

Our algorithm for provenance verification generalizes algorithms for explicit information flow studied in the context of sequential programs [107], e.g., through taint analysis. Taint analysis problems [60, 84] classify methods as *sources*, *sinks*, and *sanitizers*, and require that any data flow from sources to sinks must go through one or more sanitizers. In our model, this property can be formulated by requiring that the provenance of every message received by a sink must conform to the regular specification $(source^+ \; sanitizer^+)^*$. We are able to verify such properties for message passing programs, where the source, sanitizer, and sink can be concurrently executing processes sharing unbounded channels, and with other intermediary processes as well. Previous work, too numerous to enumerate here, either dealt with dynamic enforcement or provided imprecise static checks for these domains. We show *precise* static analysis remains decidable!

2.8 Extensions

We have described a general algebraic model of provenance in concurrent message passing systems and an algorithm for statically verifying provenance properties. For these expressive programs, only dynamic checks or imprecise static checks had been studied so far. While the complexity may seem high, reachability analysis in message passing programs is already EXPSPACE-complete, so provenance verification does not incur an extra cost.

Our decidability results continue to hold under some extensions to the programming model. For example, our decidability results also hold when programs can test the provenance of a message against an upward closed set in a conditional, or in the presence of a spawn instruction that dynamically generates a new thread of execution. Informally, to decide provenance verification in the presence of provenance-tests, we extend the product construction to track the membership in each upward closed set appearing syntactically in some conditional. To handle spawn, we modify the reduction to Petri nets to keep a place for each spawned instance (that is, each tuple of control location and valuation to local variables).

On the other hand, many other extensions are easily seen to be undecidable. For example, if each principal executes a recursive program, or if messages come from an unbounded domain such as the natural numbers, or if channels preserve the order of messages, the provenance verification problem becomes undecidable by simple reductions from known undecidable problems [95].

Name	LOC	Leak	Usage	Leak Details	Time
Amazon Price History and More 4.1.4	8124	Yes	Provide comparative pricing for searched products. Inform pricing drops for searched products.	Records shopping history while in private mode.	57ms
Facebook Chat History Manager 1.5	2798	Yes	Help users organize conversations by time and names of persons.	Records the person to whom users talk, the conversation content, and the time in private mode.	60ms
FVD Speed Dial with Online Sync 4.0.3	21278	Yes	Provide a dashboard holding favorite websites of users. Cross-platform bookmark synchronization.	Keeps counting how often users look at the websites on heir Speed Dial in private mode and lists them.	57ms
Privad 1.0	17593	Yes	Uses differential privacy to prevent ad targeting.	Records user browsing history while in private mode.	60ms
Shopping Assist 3.2.4.6	15263	Yes	Provide comparative pricing for searched products.	Records shopping history while in private mode.	57ms
Form History Control 1.2.10.3	16560	No	Autosave text on forms, search bar history, for crash recovery.		63ms
History Deleter 2.4	3027	No	Utilities to delete history automatically by user defined rules.		90ms
Lazarus: Form Recovery 2.3	10839	No	Autosave text on forms, search bar history, for crash recovery.		64ms
Session Manager 0.7.9	14010	No	Autosave sessions by time for crash recovery.		104ms

TABLE 2.3: Experimental results for Firefox extensions.

Chapter 3

KUAI: A Model Checker for Software-defined Networks

3.1 Introduction

Software-defined networking (SDN) is a novel networking architecture in which a centralized software controller dynamically updates the packet processing policies in network switches based on observing the flow of packets in the network [40, 62]. SDNs have been used to implement sophisticated packet processing policies in networks, and there is increasing industrial adoption [62, 91].

We consider the problem of verifying that an SDN satisfies a network-wide safety property. Since the controller code in an SDN can dynamically change how packets flow in the network, a bug in the controller code can lead to hard-to-analyze network errors at run time. We describe the design of , a distributed enumerative model checker for SDNs. The input to is a model of an SDN consisting of two parts. The first part is the controller, written in a simplified guarded-command language similar to Murphi. The second part is the description of a network, consisting of a fixed finite set of switches, a fixed set of client nodes, and the topology of the network (i.e., the connections between the ports of the clients and the switches). Given a safety property of the network, explores the state space of the SDN to check if the property holds on all executions.

Figure 3.1 shows a simple SDN. It consists of two switches sw_1 and sw_2 connected to two clients c_1 and c_2. Each client has a port and each switch has two ports to send and receive packets, and the figure shows how the ports are connected to each other. Each connection between ports represents a bi-directional communication channel that

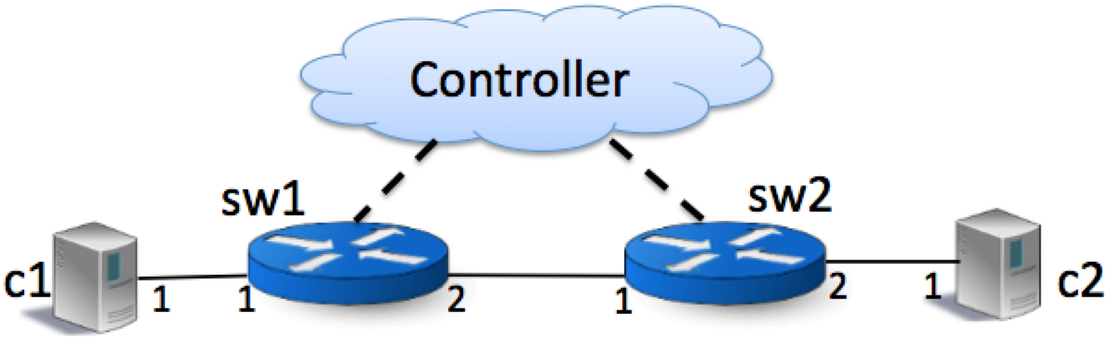


FIGURE 3.1: SSH Example

```
def pktIn(pkt)
  (sw,pt) = pkt.loc
  if pkt.prot = SSH:
    drop(pkt)
  else:
    dest = 2 if pt = 1 else 1
    fwd(pkt, [|dest|], sw)
  rule r1 = (5,{prot=SSH},[||])
  rule r2 = (1,{port=1},[|2|])
  rule r3 = (1,{port=2},[|1|])
  message cm1 = add(r1)
  message cm2 = add(r2)
  message cm3 = add(r3)
  for sw in [sw1, sw2]:
    send_message(cm1, sw)
    send_message(cm2, sw)
    send_message(cm3, sw)
```

LISTING 3.1: Controller for SSH

may reorder packets. Moreover, the switches are connected to a controller through dedicated links. Packets are routed in the network using *flow tables* in switches. A flow table is a collection of prioritized forwarding *rules*. A rule consists of a priority, a pattern on packet headers, and a list of ports. A switch processes an incoming packet based on its flow table. It looks at the highest priority rule whose pattern matches the packet and forwards the packet to the list of ports specified in the rule, and drops the packet if the list of ports in the rule is empty. In case no rule matches a packet, the switch forwards the packet to the controller using a request queue and waits for a reply from the controller on a forward queue. The controller replies with a list of ports to which the packet should be forwarded, and optionally sends *control messages* to the control queue of one or more switches to update their flow tables. A control message can add or delete a rule in a switch.

By specifying the rules to be added or deleted, a controller can dynamically control the behaviors of all switches in an SDN network. For example, suppose we want to implement the policy that all SSH packets are dropped. The controller can update the switches with a rule that states that no SSH packets are forwarded, and another that states all non-SSH packets are forwarded. List 3.1 shows a possible controller that implements this policy. Essentially, the controller drops SSH packets, and adds three rules on the switches: $r1$ to drop SSH packets, $r2$ to forward packets from port 1 to port 2, and $r3$ to forward packets from port 2 to port 1. Since dropping SSH packets (rule $r1$) has higher priority, it will match SSH packets, and rules $r2$ and $r3$ will only match (and forward) non-SSH packets. The controller has a subtle bug. It turns out that a switch can implement rules in arbitrary order. Thus, the switches may end up adding rules $r2$ and $r3$ before adding $r1$, thus violating the policy. Our model checker KUAI confirms the bug A possible fix in this case is to implement a *barrier* after line 15, to ensure that rule $r1$ is added before the other rules. Our model checker confirms the policy holds in the fixed version.

The verification of SDNs is challenging due to several reasons. First, even when the topology is fixed with a finite set of clients and switches, the state space is still unbounded, as clients may generate unboundedly many packets and these packets could be simultaneously progressing through the network. For example, client c_1 may send a packet to sw_1 at any point, and an unbounded number of packets can be in the network before sw_1 processes them. Similarly, there may be an unbounded number of control messages (i.e., messages sent from the controller to a switch) between the controller and the switches. While there may be a physical limit on the number of packets and control messages imposed by packet buffers in the switches, the sizes of these buffers can be large (of the order of megabytes) and precise modeling of buffers will blow up the state space.

Second, the packets may be processed in arbitrary interleaved orders, and the processing of one packet may influence the processing of subsequent ones because the controller may update flow tables based on the first packet. Similarly, control messages between the controller and the switches may be processed in arbitrary order and this may lead to potential bugs, including the bug pointed to above.

KUAI handles these challenges in the following way. First, instead of modeling unbounded multisets for packet queues, we implement a *counter abstraction* where we track, for each possible packet, whether zero or arbitrarily many instances of the packet are waiting in a multiset. This abstraction enables us to apply finite-state model checking approaches.

Second, we implement a set of partial-order reduction techniques that are specific to the SDN domain. For example, we note that while in principle a switch only processes one packet at a time, we do not lose behaviors by processing all packets at the packet queue of a switch atomically. Similarly, using the semantics of the barrier message [91], we show that a switch can atomically execute all control messages up to the last barrier in its control queue. Specifically, this optimization enables the model checker to bound the size of control queues. Additionally, we show that whenever there is a packet in a client's packet queue, the client can receive and process it immediately, so that sends from switches can be atomically processed with receives at clients. Finally, we show that we can eagerly serve requests to the controller, that is, we do not lose behaviors if we restrict the controller's request queue to size one and service these requests as soon as they appear.

We empirically demonstrate that our set of partial order reduction techniques significantly reduces the state spaces of SDN benchmarks, often by many orders of magnitude. For the simple SSH example, the number of explored states is approximately 2 million without partial order reductions, but only 13 with reductions!

To handle large state spaces, our model checker KUAI distributes the model checking over a number of nodes in a cluster, using the PReach distributed model checker [10] (based on Murphi [37]) as its back end. The large-scale distribution enables KUAI to model check large state spaces quickly.

3.2 Software-defined Networks

Preliminaries. A *multiset* m over a set Σ is a function $\Sigma \to \mathbb{N}$ with finite support (i.e., $m(\sigma) \neq 0$ for finitely many $\sigma \in \Sigma$). By $\mathbb{M}[\Sigma]$ we denote the set of all multisets over

Σ. We shall write $m = [\![\sigma_1^2, \sigma_3]\!]$ for the multiset $m \in \mathbb{M}[\{\sigma_1,\sigma_2,\sigma_3\}]$ with $m(\sigma_1) = 2, m(\sigma_2) = 0$, and $m(\sigma_3) = 1$. We write $\emptyset$ for an empty multiset, mapping each $\sigma \in \Sigma$ to 0. We write $\{\}$ for an empty set. Two multisets are ordered by $m_1 \leq m_2$ if for all $\sigma \in \Sigma$, we have $m_1(\sigma) \leq m_2(\sigma)$. Let $m_1 \oplus m_2$ (resp. $m_1 \ominus m_2$) be the multiset that maps every element $\sigma \in \Sigma$ to $m_1(\sigma) + m_2(\sigma)$ (resp. $\max\{0, m_1(\sigma) - m_2(\sigma)\}$).

Given a set of states, a *(guarded) action* α is a pair (g, c) where g is a *guard* that evaluates the states to a boolean and c is a *command*. A action α is *enabled* in a state s if the guard of α evaluates s to true. If α is enabled in s, the command of α can execute and lead to a new state s', denoted by $s \xrightarrow{\alpha} s'$. We write $\alpha(s) = s'$ if $s \xrightarrow{\alpha} s'$. A *transition system* TS is a tuple $(S, A, \rightarrow, s_0, AP, L)$ where S is a set of states, A is a set of actions, $\rightarrow \subseteq S \times A \times S$ is a transition relation, $s_0 \in S$ is the initial state, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labeling function. We write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$. A state s' is *reachable* from s if $s \rightarrow^* s'$. We write $s \rightarrow^+ s'$ if there is a state t such that $s \rightarrow t \rightarrow^* s'$. For a state s, let $A(s)$ be the set of actions enabled in s; we assume $A(s) \neq \emptyset$ for each $s \in S$. The *trace* of an infinite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots$ is defined as $\mathit{trace}(\rho) = L(s)L(s_1)\dots$. The trace of a finite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n$ is defined as $\mathit{trace}(\rho) = L(s)L(s_1)\dots L(s_n)$. An execution is *initial* if it starts in s_0. Let $\mathit{Traces}(TS)$ be the set of traces of initial executions in TS. We define invariants and invariant satisfaction in the usual way.

Syntax of Software-defined Networks We model an SDN as a network consisting of *nodes*, *connections*, and a *controller* program. Nodes come from a finite set $\mathit{Clients}$ of *clients* and a (disjoint) finite set $\mathit{Switches}$ of *switches*. Each node n has a finite set of *ports* $\mathit{Port}(n) \subseteq \mathbb{N}$ which are connected to ports of other nodes. A *location* (n, pt) is a pair of a node and a port $pt \in \mathit{Port}(n)$. Let Loc be the set of locations. A connection is a pair of locations. A network is well-formed if there is a bijective function $\lambda : \mathit{Loc} \rightarrow \mathit{Loc}$, called the *topology function*, such that $\{((n, pt), \lambda(n, pt)) \mid (n, pt) \in \mathit{Loc}\}$ is the set of connections and no two clients are connected directly.

We model a *packet* pkt in the network as a tuple $(a_1, \dots, a_k, loc)$, where $(a_1, \dots, a_k) \in \{0, 1\}^k$ models an abstraction of the packet data and $loc \in \mathit{Loc}$ indicates the location of pkt. Let Packet be the set of all packets.

Each switch contains a set of rules that determine how packets are forwarded. A *rule* is a tuple $(priority, pattern, ports)$, where $priority \in \mathbb{N}$ determines the priority of the rule, $pattern$ is a proposition over $Packet$, and $ports$ is a multiset of ports. We write $Rule$ to denote the set of all rules. Intuitively, a packet matches a rule if it satisfies $pattern$. A switch forwards a packet along $ports$ for the highest priority rule that matches.

Rules are added or deleted on a switch by the controller through a set of *control messages* $CM = \{\mathsf{add}(r), \mathsf{del}(r) \mid r \in Rule\}$. Additionally, the controller uses a *barrier* message b to synchronize.

```
type client {
  Port : set of nat
  pq : multiset of packets
}
rule "send(c, pkt)"
  true ==> send(c, pkt)
end
rule "recv(c,pkt,pkts)"
  exist(pkt:c.pq, true) ==> recv(c,pkt,pkts)
end
```

LISTING 3.2: Client

A client $c \in Clients$ is modeled as in List 3.2. It consists of a finite set $Port$ of ports and a *packet queue* $pq \in \mathbb{M}[Packet]$ containing a multiset of packets which have arrived at the client. We use (guarded) actions to model behaviors of clients. An action is written as "**rule** *name guard* $\Longrightarrow$ *command* **end**." Predicate $exist(i : X, \varphi)$ asserts that there is an element i in the set (or multiset) X such that the predicate φ holds. Additionally, if $exist(i : X, \varphi)$ holds, then the variable i is bound to an element of X that satisfies φ and can be used later in the command part. In each step, a client c can (1) send a non-deterministically chosen packet pkt along some ports (rule send), or (2) receive a packet pkt from its packet queue and (optionally) send a multiset of packets $pkts$ on some ports (rule recv).

```
type switch {
  Port : set of nat
  ft : set of rules
  pq : multiset of packets
  cq : list of barriers and
      multisets of control messages
  fq : set of forward messages
  wait : boolean
}
rule "match(sw,pkt,r)"
  !sw.wait & noBarrier(sw) &
  exist(pkt:sw.pq,
    exist(r:sw.ft, bestmatch(sw,r,pkt))) ==>
  match(sw,pkt,r)
end
rule "nomatch(sw,pkt)"
  !sw.wait & noBarrier(sw) & !RqFull(controller) &
  exist(pkt:sw.pq,
    !exist(r:sw.ft,bestmatch(sw,r,pkt))) ==>
  nomatch(sw,pkt)
end
rule "add(sw,r)"
  !sw.wait & noBarrier(sw) &
  exist(add(r):sw.cq[0],true) ==>
  add(sw,r)
end
rule "delete(sw,r)"
  !sw.wait & noBarrier(sw) &
  exist(del(r):sw.cq[0],true) ==>
  delete(sw,r)
end
rule "fwd(sw,pkt,pts)"
  sw.wait & noBarrier(sw) &
  exist((pkt,pts):fq, true) ==>
  fwd(sw,pkt,pts)
end
rule "barrier(sw)"
  !noBarrier(sw) ==>
  barrier(sw)
end
```

LISTING 3.3: Switch

A switch sw is modeled as in List 3.3. It consists of a set of ports, a *flow table* $ft \subseteq Rule$, a packet queue pq containing packets arriving from neighboring nodes, a *control queue* cq containing control messages or barriers from the controller, a *forward queue* fq consisting of at most one pair $(pkt, ports)$ through which the controller tells the switch to forward packet pkt along the ports $ports$, and a boolean variable $wait$. Predicate $noBarrier(sw)$ asserts $sw.cq$ does not contain a barrier. Predicate $bestmatch(sw, r, pkt)$ asserts that r is the highest priority rule whose pattern matches the packet pkt in switch sw's flow table.

Intuitively, a switch has a normal mode and a waiting mode determined by the $wait$ variable. When the switch is in the normal mode, as long as there is no barrier in its control queue, it can either attempt to forward a packet from its packet queue based on its flow table, or update its flow table according to a control message in its control queue. When the switch cannot find a matching rule in its flow table for a packet, it can initiate a request to the controller, change to the waiting mode, and wait for a forward message from the controller telling it how to forward the packet. Once it receives a forward message (pkt, pts) and there is no barrier in the control queue, it forwards the pending packet pkt to the ports in pts, and changes back to the normal mode. If the control queue contains one or more barriers, the switch dequeues all control messages up to the first barrier from its control queue and updates its flow table.

```
type controller {
  CS : set of control states
  cs0 : CS                    cs : CS
  rq : set of packets         κ : ℕ+
  pktIn : function
}
rule "ctrl(pkt,cs)"
  exist(pkt:controller.rq, true) ==>
  ctrl(pkt,controller.cs)
end
```

LISTING 3.4: Controller

A controller $controller$ is modeled as in List 3.4. It is a tuple $(CS, cs_0, cs, rq, \kappa, pktIn)$ where CS is a finite set of *control states*, $cs_0 \in CS$ is the *initial control state*, cs is the *current control state*, rq is a finite *request queue* of size $\kappa \geq 1$ consisting of packets forwarded to the controller from switches, and $pktIn$ is a function that takes a packet

pkt and a control state cs_1, and returns a tuple $(\eta, (pkt, pts), cs_2)$ where η is a function from $Switches$ to $(\mathbb{M}[CM] \cup \{b\})^*$, (pkt, pts) is a forward message, and cs_2 is a control state. Intuitively, in each step, the controller removes a packet pkt from rq and executes $pktIn(pkt, controller.cs)$. Based on the result $(\eta, (pkt, pts), cs')$, it sends back to the source of the packet the forward message (pkt, pts) that specifies pkt should be forwarded along pts, and goes to a new control state cs'. Further, for each switch sw in the network it appends $\eta(sw)$ to sw's control queue.

Semantics of Software-defined Networks The semantics of an SDN is given as a transition system. Let $\mathcal{N} = (Clients, Switches, \lambda, Packet, Rule, controller)$ be an SDN, where each component is as defined above.

A *state* s of the SDN $\mathcal{N}$ is a quadruple (π, δ, cs, rq), where π is a function mapping each client $c \in Clients$ to its packet queue pq and δ is a function mapping each switch $sw \in Switches$ to a tuple $(pq, cq, fq, ft, wait)$ consisting of its packet queue, control queue, forward queue, flow table, and the wait variable.

For a non-empty list $l = [x_1, x_2, \ldots, x_n]$, define $l.hd{=}x_1$, $l.tl{=}[x_2, \ldots, x_n]$, and $l[i]$ as the i-th element in l. Given two lists l_1 and l_2, let $l_1@l_2$ be the concatenation of l_1 and l_2. For two non-empty lists $l_1 = [x_1, \ldots, x_m]$ and $l_2 = [y_1, \ldots, y_n]$ in $(\mathbb{M}[CM] \cup \{b\})^*$, define $l_1 + l_2$ be the list $[x_1, \ldots, x_{m-1}, x_m \oplus y_1, y_2, \ldots, y_n]$ if $x_m \neq b$ and $y_1 \neq b$; $l_1@l_2$ otherwise.

Given a flow table ft and a list $l \in (\mathbb{M}[CM] \cup b)^*$, let $update(ft, l)$ be a procedure that updates ft based on l as follows. It dequeues the head of l and sets l to $l.tl$. If the head is a barrier b, then ignore it. If the head is a multiset m, it nondeterministically chooses a *fetching order* p and based on p, removes a control message cm with $m(cm) > 0$ from m. If cm is $\mathsf{add}(r)$, then add the rule r to ft, or if cm is $\mathsf{del}(r)$, then delete r from ft. It keeps updating ft based on p until m becomes empty. It repeats the above instructions on l until l becomes empty. Then it returns the resulting flow table ft.

For a function $f\colon X \to Y$, $x \in X$, and $y \in Y$, let $f[x \mapsto y]$ denote the function that maps x to y and all $x' \neq x$ to $f(x')$. Let $f[x_1 \mapsto y_1; x_2 \mapsto y_2; \ldots; x_n \mapsto y_n]$ denote the function $f[x_1 \mapsto y_1][x_2 \mapsto y_2] \ldots [x_n \mapsto y_n]$. Given a subset $X' = \{x_1, \ldots, x_n\} \subseteq X$, let $f[\text{foreach } x_i \in X' : x_i \mapsto y_i]$ be the function $f[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$ where $1 \leq i \leq n$. Given a tuple $t = (f_1, \ldots, f_n)$, let $t.f_i$ be the field f_i, for $1 \leq i \leq n$. By abuse of notation, we write $t[f_i \mapsto v]$ to be the tuple such that $t[f_i \mapsto v].f_i = v$ and for any $j \neq i$, $t[f_i \mapsto v].f_j = t.f_j$.

We define the following *basic operations* over δ and π:

1. Add or delete packets in switches or in clients. Given a set $X \subseteq \mathit{Switches} \times \mathit{Packet}^{\mathbb{N}}$, define $\mathit{addPkt}(\delta, X) = \delta[\text{foreach}\ (sw, pkt^k) \in X, sw \mapsto \delta(sw)[pq \mapsto \delta(sw).pq$
$\oplus [\![pkt^k]\!]]]$. Given a set $Y \subseteq \mathit{Clients} \times \mathit{Packet}^{\mathbb{N}}$, define $\mathit{addPkt}(\pi, Y) = \pi[\text{foreach}\ (c, pkt^k) \in Y, c \mapsto \pi(c) \oplus [\![pkt^k]\!]]$. We define $\mathit{delPkt}(\delta, X)$ and $\mathit{delPkt}(\pi, Y)$ analogously by replacing $\oplus$ with $\ominus$ above.

2. Set the $wait$ bit of a switch sw to true or false. Define $setWait(\delta, sw) = \delta[sw \mapsto \delta(sw)[wait \mapsto \mathit{true}]]$ and $unsetWait(\delta, sw) = \delta[sw \mapsto \delta(sw)[wait \mapsto \mathit{false}]]$.

3. Add or delete a rule r in the flow table of a switch sw. Define $\mathit{addRule}(\delta, sw, r) = \delta[cq \mapsto [\delta(sw).cq.hd \ominus [\![\mathsf{add}(r)]\!]]; sw \mapsto \delta(sw)[\mathit{ft} \mapsto \delta(sw).\mathit{ft} \cup \{r\}]]$. Define $\mathit{delRule}(\delta, sw, r) = \delta[cq \mapsto [\delta(sw).cq.hd \ominus [\![\mathsf{del}(r)]\!]]; sw \mapsto \delta(sw)[\mathit{ft} \mapsto \delta(sw).\mathit{ft} \backslash \{r\}]]$.

4. Add or delete a forward message msg in a switch sw. Define $\mathit{addFwdMsg}(\delta, sw, msg) = \delta[sw \mapsto \delta(sw)[\mathit{fq} \mapsto \delta(sw).\mathit{fq} \cup \{msg\}]]$ and $\mathit{delFwdMsg}(\delta, sw, msg) = \delta[sw \mapsto \delta(sw)[\mathit{fq} \mapsto \delta(sw).\mathit{fq} \backslash \{msg\}]]$.

5. Flush and run all control messages up to the first barrier in a switch. Define $\mathit{flush}(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l; \mathit{ft} \mapsto update(\delta(sw).\mathit{ft}, [m, b])]]$ where $l = [\emptyset]$, if $\delta(sw).cq = [m, b]$; $l = l'$, if $\delta(sw).cq = [m, b]@l'$ and l' is not an empty list.

6. Flush and run all control messages up to the last barrier in a switch. Define $\mathit{flushall}(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l_1; \mathit{ft} \mapsto update(\delta(sw).\mathit{ft}, l_2)]]$ where $l_1 = [\emptyset]$ and $l_2 = \delta(sw).cq$ if the last element of $\delta(sw).cq$ is a barrier. Otherwise, let $\delta(sw).cq = l@[m]$. Then $l_1 = [m]$ and $l_2 = l$.

7. Add control messages and barriers to the control queues of the switches. Given a total function $f : \mathit{Switches} \to (\mathbb{M}[CM] \cup \{b\})^*$, define $\mathit{addCtrlCmd}(\delta, f) = \delta[\text{foreach}\ sw \in \mathit{Switches} : sw \mapsto \delta(sw)[cq \mapsto \delta(sw).cq + f(sw)]]$.

For a switch sw, a packet pkt, and a multiset of ports pts, let $FwdToC(sw, pkt, pts)$ be a set $\{(c, pkt'^k) \mid \exists pt \in sw.Port.\ pts(pt) = k \wedge \lambda(sw, pt) = (c, pt') \wedge c \in Clients \wedge pkt' = pkt[loc \mapsto (c, pt')]\}$ and $FwdToSw(sw, pkt, pts)$ be a set $\{(sw', pkt'^k) \mid \exists pt \in sw.Port.\ pts(pt) = k \wedge \lambda(sw, pt) = (sw', pt') \wedge sw' \in Switches \wedge pkt' = pkt[loc \mapsto (sw', pt')]\}$. Intuitively, when sw is about to forward pkt on its ports pts, these two sets summarize how many packets should be forwarded to its connected clients and switches.

For an SDN $\mathcal{N}$, let $Send = \{send(c, pkt) \mid c \in Clients \wedge pkt \in Packet\}$ be the set of *send actions*. We define analogously the set of *receive actions* $Recv$, the set of *match actions* $Match$, the set of *no-match actions* $NoMatch$, the set of *add actions* Add, the set of *delete actions* Del, the set of *forward actions* $Forward$, the set of *barrier actions* $Barrier$, and the set of *control actions* $Ctrl$.

Let $\pi_0 = \lambda c \in Clients.\emptyset$ and $\delta_0 = \lambda sw \in Switches.(\emptyset, [\emptyset], \{\}, \{\}, false)$. The semantics of an SDN $\mathcal{N}$ is given by a transition system $TS(\mathcal{N}) = (S, A, \rightarrow, s_0, AP, L)$. Here, S is the set of states, $s_0 = (\pi_0, \delta_0, cs_0, \{\})$ is the initial state, and $A = Send \cup Recv \cup Match \cup NoMatch \cup Add \cup Del \cup Forward \cup Barrier \cup Ctrl$. The transition relation $s \xrightarrow{\alpha} s'$ is defined as follows.

1. $\alpha = send(c, pkt)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = addPkt(\delta, \{(sw, pkt)\})$ and $sw = pkt.loc.n$.

2. $\alpha = recv(c, pkt, pkts)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = delPkt(\pi, \{(c, pkt)\})$, $\delta' = addPkt(\delta, X)$ and $X = \{(sw, pkt'^k) \mid pkts(pkt') = k \wedge pkt'.loc.n = sw\}$.

3. $\alpha = match(sw, pkt, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = addPkt(\pi, FwdToC(sw, pkt, r.ports))$ and $\delta' = addPkt(\delta, FwdToSw(sw, pkt, r.ports))$.

4. $\alpha = nomatch(sw, pkt)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq')$ where $rq' = rq \cup \{pkt\}$, $\delta'' = delPkt(\delta, \{(sw, pkt)\})$, and $\delta' = setWait(\delta'', sw)$.

5. $\alpha = add(sw, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = addRule(\delta, sw, r)$.

6. $\alpha = del(sw, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = delRule(\delta, sw, r)$.

7. $\alpha = \mathit{fwd}(sw, pkt, pts)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = \mathit{addPkt}(\pi, \mathit{FwdToC}(sw, pkt, pts))$, $\delta_1 = \mathit{delFwdMsg}(\delta, sw, (pkt, pts))$, $\delta_2 = \mathit{addPkt}(\delta_1, \mathit{FwdToSw}(sw, pkt, pts))$, and $\delta' = \mathit{unsetWait}(\delta_2, sw)$.

8. $\alpha = \mathit{barrier}(sw)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = \mathit{flush}(\delta, sw)$.

9. $\alpha = \mathit{ctrl}(pkt, cs)$. Let $\mathit{pktIn}(pkt, cs) = (\eta, msg, cs')$ and $sw = pkt.loc.n$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs', rq')$ where $rq' = rq \backslash \{pkt\}$, $\delta'' = \mathit{addFwdMsg}(\delta, sw, msg)$, and $\delta' = \mathit{addCtrlCmd}(\delta'', \eta)$.

An atomic proposition $p \in AP$ is an assertion over packet fields or over control states. Define an SDN specification as a safety property $\Box\phi$ where ϕ is a formula over AP and $\Box$ is the "globally" operator of linear-temporal logic. The *model checking problem for an SDN* asks, given an SDN $\mathcal{N}$ and an SDN specification $\Box\phi$, if $TS(\mathcal{N})$ satisfies $\Box\phi$. For example, blocking SSH packets can be specified as $\Box \bigwedge_{pkt \in Packet}(pkt.loc.n \in \mathit{Clients} \wedge pkt.src \in \mathit{Clients} \wedge pkt.loc.n \neq pkt.src \Rightarrow pkt.prot \neq \mathrm{SSH})$.

3.3 Optimizations

We now describe partial-order reduction and abstraction techniques that reduce the state space. These techniques use the structure of SDNs and, as we demonstrate empirically, are crucial in making the model checking scale to non-trivial examples. We state the correctness theorems; the proofs are in Section 3.5.

Partial Order Reduction Let $TS = (S, A, \rightarrow, s_0, AP, L)$ be an action-deterministic transition system, i.e., $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ implies $s' = s''$. Given two actions $\alpha, \beta \in A$ with $\alpha \neq \beta$, α and β are *independent* if for any $s \in S$ with $\alpha, \beta \in A(s)$, $\beta \in A(\alpha(s))$, $\alpha \in A(\beta(s)))$, and $\alpha(\beta(s)) = \beta(\alpha(s))$. The actions α and β are dependent if α and β are not independent. An action $\alpha \in A$ is a *stutter action* if for each transition $s \xrightarrow{\alpha} s'$ in TS, we have $L(s) = L(s')$.

For $i \in \{1, 2\}$, let $TS_i = (S_i, A_i, \rightarrow_i, s_0^i, AP, L_i)$ be transition systems. Infinite executions ρ_1 of TS_1 and ρ_2 of TS_2 are *stutter-equivalent*, denoted $\rho_1 \triangleq \rho_2$, if there is an infinite sequence $A_0 A_1 A_2 \ldots$ with $A_i \subseteq AP$, and natural numbers

$n_0, n_1, n_2, \ldots, m_0, m_1, m_2, \ldots \geq 1$ such that

$$trace(\rho_1) = \underbrace{A_0 \ldots A_0}_{n_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{n_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{n_2 \text{ times}} \ldots$$

$$trace(\rho_2) = \underbrace{A_0 \ldots A_0}_{m_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{m_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{m_2 \text{ times}} \ldots$$

TS_1 and TS_2 are *stutter equivalent*, denoted $TS_1 \triangleq TS_2$, if $TS_1 \trianglelefteq TS_2$ and $TS_2 \trianglelefteq TS_1$, where $\trianglelefteq$ is defined by: $TS_1 \trianglelefteq TS_2$ iff for all $\rho_1 \in \mathit{Traces}(TS_1)$. $\exists \rho_2 \in \mathit{Traces}(TS_2)$. $\rho_1 \triangleq \rho_2$.

3.3.1 Barrier Optimization

Intuitively, barrier optimization uses the observation that for any state, we can always flush out control queues of switches until there are no barriers in them. This implies that after a control action is executed, one can immediately update flow tables of switches whose control queue has barriers added by the controller. Hence a control action and successive barrier actions can be merged. We prove its correctness by viewing it as an instance of partial order reduction.

For an SDN $\mathcal{N}$, note that $TS(\mathcal{N})$ is not action-deterministic due to barrier actions. With different fetching orders, $barrier(sw)$ may lead to multiple states. Define $b(s, sw)$ as the number of transitions of the form $s \xrightarrow{barrier(sw)} s'$. Note that a barrier action from any s leads to at most $2^{|Rule|}$ states. Hence for each transition $s \xrightarrow{barrier(sw)} s_i$ where $1 \leq i \leq b(s, sw)$, we can append the action with the index i, i.e., $s \xrightarrow{barrier(sw)_i} s_i$. In the following, we redefine the set $Barrier = \{barrier(sw)_i \mid sw \in Switches \wedge 1 \leq i \leq 2^{|Rule|}\}$, and assume that $TS(\mathcal{N})$ is action-deterministic by renaming barrier actions.

A switch sw has a barrier iff there is a barrier in sw's control queue. A state s has a barrier, denoted $hasb(s)$, iff some switch $sw \in Switches$ has a barrier in s. Define the *ample set* for every state s in $TS(\mathcal{N})$ as follows: if s has a barrier, then $ample(s) = \{barrier(sw)_i \mid 1 \leq i \leq b(s, sw) \wedge sw$ has a barrier in $s\}$, that is, all barrier actions enabled in s. If s does not have a barrier, then $ample(s) = A(s)$.

Given $TS(\mathcal{N})$, we now define a transition system $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$ where $\hat{S} = S$ is the set of states, and the transition relation $\Rightarrow$ is defined as: if $s \xrightarrow{\alpha} s'$ and

$\alpha \in ample(s)$, then $s \overset{\alpha}{\Rightarrow} s'$.

Theorem 5. *Let $TS(\mathcal{N})$ be an action-deterministic transition system. $TS(\mathcal{N}) \triangleq \widehat{TS}$.*

Intuitively, Theorem 5 holds because any barrier action is independent of other actions and is a stutter action. Hence for an infinite execution $s \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{barrier(sw)} t$ in $TS(\mathcal{N})$ where s has a barrier and α_i is not a barrier action for all $1 \leq i \leq n$, we can permute $barrier(sw)$ forward until s and obtain a stutter-equivalent execution in $\widehat{TS}$.

Since Theorem 5 holds, we can merge a control action and successive barrier actions into a single transition $s \xrightarrow{ctrl(pkt,cs)}_2 s'$ where we define the new semantics of $ctrl(pkt, cs)$ under the transition relation $\rightarrow_2$. Formally, Let $(\eta, (pkt, pts), cs') = pktIn(pkt, cs)$ and $sw = pkt.loc.n$.

Ctrl. $(\pi, \delta, cs, rq) \xrightarrow{ctrl(pkt,cs)}_2 (\pi, \delta', cs', rq')$ where $rq' = rq\backslash\{pkt\}$. Define $\delta'' = addFwdMsg(\delta, sw, (pkt, pts))$, and $\delta''' = addCtrlCmd(\delta'', \eta)$. Let $\{sw_1, \ldots, sw_n\}$ be the set of all switches whose control queue has barriers in δ'''. Let $\delta_0 = \delta'''$ and $\delta_i = flushall(\delta_{i-1}, sw_i)$ for all $1 \leq i \leq n$. Define $\delta' = \delta_n$.

Given $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$, define a transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$ where $S_2 \subseteq \hat{S}$ is a set of states reachable by $\rightarrow_2$, A_2 is $A\backslash Barrier$, $AP_2 = AP$, $L_2 = L$, and $\rightarrow_2$ is defined inductively as

$$\frac{s_0 \overset{\alpha}{\Rightarrow} s'}{s_0 \xrightarrow{\alpha}_2 s'} \qquad \frac{s_0 \rightarrow_2^+ s \overset{\alpha}{\Rightarrow} s' \wedge \alpha \notin Ctrl}{s \xrightarrow{\alpha}_2 s'}$$

$$\frac{s_0 \rightarrow_2^+ s \overset{\alpha}{\Rightarrow} t \Rightarrow^* s' \wedge \alpha \in Ctrl \wedge \neg hasb(s')}{s \xrightarrow{\alpha}_2 s'}$$

Since we only remove barrier actions which are stutter actions, we have $TS_2 \triangleq \widehat{TS} \triangleq TS(\mathcal{N})$. Hence we have the following theorem:

Theorem 6. *Given an SDN $\mathcal{N}$ and a safety property $\Box\phi$, $TS(\mathcal{N})$ satisfies $\Box\phi$ iff TS_2 satisfies $\Box\phi$.*

3.3.2 Client Optimization

Given transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$, we further reduce the state space by observing that any receive action of a client is a stutter action and is independent of other actions. Formally, we define $ample(s)$ for each state $s \in S_2$ as follows: if there is a client in s such that its packet queue is not empty, then $ample(s) = \{recv(c, pkt, pkts) \mid pkt$ is in $c.pq$ at $s\}$, that is, all receive actions enabled in s. Otherwise, $ample(s) = A(s)$. We now define a transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$ where $S_3 = S_2$, $A_3 = A_2$, $AP_3 = AP_2$, $L_3 = L_2$, and where the transition relation $\rightarrow_3$ is defined as: if $s \xrightarrow{\alpha}_2 s'$ and $\alpha \in ample(s)$, then $s \xrightarrow{\alpha}_3 s'$.

Theorem 7. *(1) $TS_2 \triangleq TS_3$. (2) Given a safety property $\Box\phi$, TS_2 satisfies $\Box\phi$ iff TS_3 satisfies $\Box\phi$.*

3.3.3 $(0, \infty)$ Abstraction

The $(0, \infty)$ abstraction bounds the size of packet queues and the multiset in each control queue. The idea is as follows. One can regard a multiset as a counter that counts the number of elements in it exactly. Instead, $(0, \infty)$ abstraction abstracts a multiset so that for each element e, it either does not contain e (i.e. 0) or contains unboundedly many copies of e (i.e. ∞). Then the size of an abstracted multiset is bounded. Note that for any state s in TS_3, any switch's control queue contains exactly one multiset. Hence, the abstraction bounds the length of control queues.

Let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ be the extension of the natural numbers with infinity. We naturally extend the addition operation by assuming that $\infty + \infty = \infty$ and $\infty + c = \infty$ for all $c \in \mathbb{Z}$. Given a multiset $m \in \mathbb{M}[D]$ for some finite set D, define an *extended multiset* $\mathsf{over}(m)$ such that for each element $d \in D$, $\mathsf{over}(m)(d) = 0$ if $m(d) = 0$, and $\mathsf{over}(m)(d) = \infty$ otherwise. Define $\mathbb{M}[D]^\infty$ as the set of all extended multisets and multisets over D. Given a control queue cq with length n, let $\mathsf{over}(cq)$ be such that for $1 \leq i \leq n$, $\mathsf{over}(cq)[i] = \mathsf{over}(cq[i])$ if $cq[i] \neq b$; $\mathsf{over}(cq)[i] = b$ otherwise. For $m_1, m_2 \in \mathbb{M}[D]^\infty$, we write $m_1 \leq_e m_2$ iff for all $d \in D$, $m_1(d) \leq m_2(d)$ or $m_2(d) = \infty$. Given two control queues cq, cq' of same length n, define $cq \leq_e cq'$ iff for each $1 \leq i \leq n$, $(cq[i] = b \leftrightarrow cq'[i] = b) \wedge (cq[i] \neq b \rightarrow cq[i] \leq_e cq'[i])$.

Given an SDN and the transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$, Define a transition system $TS_4 = (S_4, A_4, \rightarrow_4, s_0, AP_4, L_4)$ where $S_4 = \{\mathsf{over}(s) \mid s \in S_3\}$, $A_4 = A_3$, $AP_4 = AP_3$, and $L_4 = L_3$. The definition of $\rightarrow_4$ is given in detail in Section 3.5.3. We provide the intuition of $\rightarrow_4$ here: $\rightarrow_4$ is defined so that (1) whenever a packet pkt is added $k \geq 1$ times into a packet queue pq, we set pq to $\mathsf{over}(pq \oplus [\![pkt^k]\!])$, and (2) whenever $\eta(sw)$ is added into switch sw's control queue cq, we set cq to $\mathsf{over}(cq + \eta(sw))$. The following lemma claims that TS_4 simulates TS_3, which leads to Theorem 8.

Lemma 3. *For any infinite initial execution $s_0 \xrightarrow{\beta_1}_3 s_1 \xrightarrow{\beta_2}_3 s_2 \ldots$ in TS_3, there is an infinite initial execution $t_0 \xrightarrow{\beta_1}_4 t_1 \xrightarrow{\beta_2}_4 t_2 \ldots$ in TS_4 such that for all $i \geq 0$, $s_i = (\pi_i, \delta_i, cs_i, rq_i)$ and $t_i = (\pi'_i, \delta'_i, cs'_i, rq'_i)$ satisfy the following condition: for all $c \in Clients$, $\pi_i(c) \leq_e \pi'_i(c)$ and for all $sw \in Switches$, $\delta_i(sw).pq \leq_e \delta'_i(sw).pq$, $\delta_i(sw).cq \leq_e \delta'_i(sw).cq$, $\delta_i(sw).fq = \delta'_i(sw).fq$, $\delta_i(sw).ft = \delta'_i(sw).ft$, and $\delta_i(sw).wait = \delta'_i(sw).wait$, and $cs_i = cs'_i$, and $rq_i = rq'_i$.*

Theorem 8. *Given a safety property $\Box\phi$, if TS_4 satisfies $\Box\phi$ then TS_3 satisfies $\Box\phi$.*

3.3.4 All Packets in One Shot Abstraction

So far, a switch processes a single packet at a time. We can further reduce the reachable state space by forcing a switch to process all packets matched by some rule at a time. The intermediate states produced by successive match actions in a switch are removed. Let $TS_4 = (S_4, A_4, \rightarrow_4, s_0, AP_4, L_4)$. Define a transition system $TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5)$ where $S_5 = S_4$, $AP_5 = AP_4$, $L_5 = L_4$, A_5 is the union of the new "multiple" match actions and A_4 excluding the old "single" match actions, and $\rightarrow_5$ is defined as:

$$\frac{s \xrightarrow{\alpha}_4 s' \wedge \alpha \text{ is not a match action}}{s \xrightarrow{\alpha}_5 s'}$$

and if $pkt\_lst = [pkt_1, \ldots, pkt_n]$ and $r\_lst = [r_1, \ldots, r_n]$

$$\frac{s \xrightarrow{match(sw,pkt_1,r_1)}_4 s_1 \ldots s_{n-1} \xrightarrow{match(sw,pkt_n,r_n)}_4 s'}{s \xrightarrow{match(sw,pkt\_lst,r\_lst)}_5 s'}$$

We prove TS_5 simulates TS_4. We define a relation $R \subseteq S_4 \times S_5$ such that $((\pi, \delta, cs, rq), (\pi', \delta', cs', rq')) \in R$ iff for all $pkt \in Packet$, for all $c \in Clients$, $\pi(c)(pkt) = \infty \rightarrow \pi'(c)(pkt) = \infty$ and for all $sw \in Switches$, $\delta(sw).pq(pkt) = \infty \rightarrow \delta'(sw).pq(pkt) = \infty$, $\delta(sw).cq = \delta'(sw).cq$, $\delta(sw).fq = \delta'(sw).fq$, $\delta(sw).ft = \delta'(sw).ft$, and $\delta(sw).wait = \delta'(sw).wait$, and $cs = cs'$, and $rq = rq'$.

Theorem 9. *(1)The relation R is a simulation relation. (2)For a safety property $\Box\phi$, if TS_5 satisfies $\Box\phi$, then TS_4 satisfies $\Box\phi$.*

3.3.5 Controller Optimization

We consider a restricted class of SDNs in which the size κ of the controller's request queue is one. Under this restriction, we can define a new transition system TS_6 that is stutter equivalent to TS_5 and has fewer reachable states. The idea is to observe that a no-match action is a stutter action and is independent of any actions before a corresponding control action is executed. Formally, given $TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5)$, we define a new transition relation $\rightarrow_6$ inductively:

$$\frac{s_0 \xrightarrow{\alpha}_5 s'}{s_0 \xrightarrow{\alpha}_6 s'} \qquad \frac{s_0 \rightarrow_6^+ s_1 \xrightarrow{nomatch(sw,pkt)}_5 s_2 \xrightarrow{ctrl(pkt,cs)}_5 s'}{s_1 \xrightarrow{nomatch\_ctrl(sw,pkt,cs)}_6 s'}$$

$$\frac{s_0 \rightarrow_6^+ s_1 \xrightarrow{\alpha}_5 s' \land \alpha \text{ is not a no-match action}}{s_1 \xrightarrow{\alpha}_6 s'}$$

where a new action $nomatch\_ctrl(sw, pkt, cs)$ merges $nomatch(sw, pkt)$ and $ctrl(pkt, cs)$ actions. We define a transition system $TS_6 = (S_6, A_6, \rightarrow_6, s_0, AP_6, L_6)$, where $S_6 = S_5$ is the set of states, A_6 is the union of all $nomatch\_ctrl(sw, pkt, cs)$ actions and $A_5 \backslash (NoMatch \cup Ctrl)$, $AP_6 = AP_5$, and $L_6 = L_5$.

Theorem 10. *Given an SDN $\mathcal{N}$ where the size of the request queue of the controller is one, and a safety property $\Box\phi$. (1) $TS_5 \triangleq TS_6$. (2) TS_5 satisfies $\Box\phi$ iff TS_6 satisfies $\Box\phi$.*

3.4 Implementation and Evaluation

KUAI[1] is implemented on top of PReach [10], a distributed enumerative model checker built on Murphi. We model switches, clients, and the controller as concurrent Murphi processes which communicate using message passing, with the queues modeled as multisets. We manually abstract IP packets using predicates used in the controller. We implement $(0, \infty)$-counter abstraction as a library on top of Murphi multisets.

KUAI takes as input topology information such as the number of switches, clients, and their connections, (manually) abstracted packets, and the controller code written as a Murphi process, and invariants written in Murphi syntax. We found it fairly straightforward to port POX [102] controllers due to the imperative features of Murphi. Murphi allows arbitrary first order logic formulas as invariants and it is easy to specify safety properties. KUAI compiles them into a single Murphi file and the model checking effort is then distributed across several machines using PReach. Finally the output of the tool is an error trace if the program invariant fails, or *success* otherwise.

We have evaluated KUAI on a number of real world OpenFlow benchmarks. The experiments were performed on a cluster of 5 Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 x 16GB Quad Rank RDIMMs memory and 174GB storage. Our experiments had access to a total of 150 cores and had access to 4TB of RAM.

Table 3.1 shows a summary of experimental results and compares against model checking without the optimizations from Section 3.3. Empty rows indicate model checking did not terminate in 1 hour or ran out of memory. Figure 3.2 shows the scalability of model checking with increasing distribution on the three largest examples. We noticed that the performance of the distributed model checker plateaued around 70 Erlang processes on these and other large examples. Thus, times (in table 3.1) are provided for configurations that use 70 Erlang processes. As we introduced abstractions, it is possible that we get false positives. We verified the existence of all bugs reported by KUAI manually and there were no false positives.

Besides the table, we plot the MAC learning example in Figure 3.3, which shows

[1]The tool is can be downloaded at https://github.com/t-saideep/kuai

Program	Bytes/ state	w/o optimizations States	w/o optimizations Time	w/ optimizations States	w/ optimizations Time
SSH 2×2	304	2,283,527	23.52s	13	6.40s
ML 3×3	320	9,109,456	89.99s	5308	6.39s
ML 6×3	748			23,926,202	604.07s
ML 9×2	1276			18,615,767	793.84s
FW(S) 1×2	332	2,110,986	26.89s	3645	5.45s
FW(M) 2×4	448			45,507	8.03s
FW(M) 3×4	560			512,439	55.06s
FW(M) 4×4	676			5,360,871	475.54s
RS 4×4	764			4998	6.60s
RS 4×5	764			590,570	82.82s
RS 4×6	764			5,112,013	327.39s
SIM 5×6	632			167	6.23s
SIM 5×8	632			167	6.34s
SIM 5×12	1108			167	6.85s

TABLE 3.1: Experimental results. Omitted entries indicate that model checking did not terminate. The number X×Y in the Program column means that there are X switches and Y clients in the example.

how significantly our optimization techniques reduce the state space. Though we still suffer from the state-space explosion problem, our optimizations delay it and enable us to verify SDNs with much larger configurations.

We now describe the benchmarks in detail.

SSH We run KUAI on the SSH controller from Listing 3.1. It finds the control message reordering bug in 0.1 seconds. By adding a barrier after line 15, KUAI proves the correctness in 6.4 seconds by exploring 13 states. In contrast, the unoptimized version explores over 2 million states.

MAC Learning Controller (ML) This is based on the POX [102] implementation of the standard ethernet discovery protocol. We checked there are no forwarding loops (similar to [114]), i.e., a packet should not reach a switch more than once. Packets are augmented with a bit for each switch which gets set when the switch processes that packet. The invariant is specified using these visit-bits (called *reached*): $\Box\ \forall sw \in Switches.\ \forall pkt \in sw.pq.\ (\neg pkt.reached(sw))$.

A cycle in the topology will lead to forwarding loops as the controller does not compute the minimum spanning tree. We discover the bug in a cyclic topology of 3 switches 3 clients in 0.47 seconds. We re-ran the example on a topology containing the minimum spanning tree of the original cyclic topology and the tool is able to prove

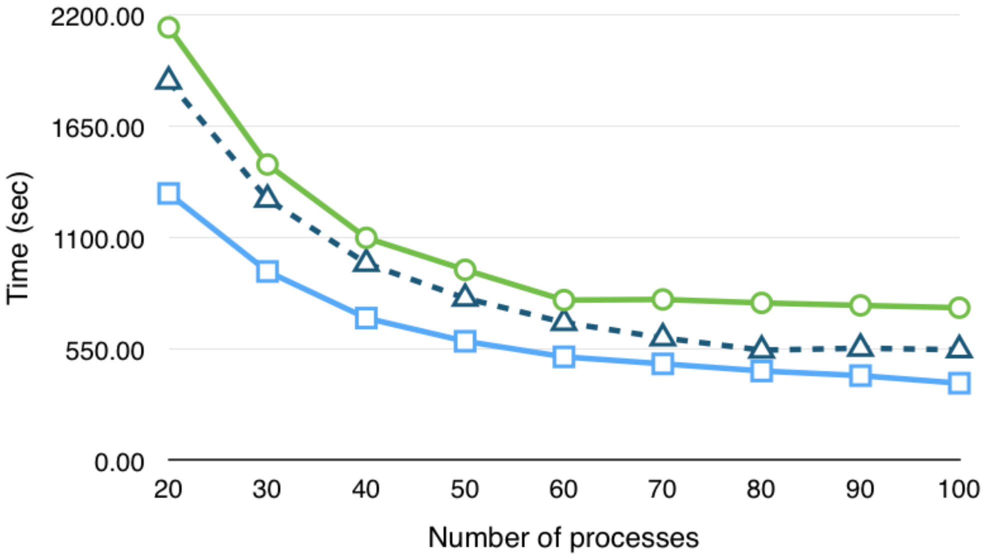


FIGURE 3.2: Verification time vs processes ○ ML 9×2 △ ML 6×3 □ FW(M) 4×4

that there were no forwarding loops in 6.39 seconds. We scale the example by adding more switches. We notice that while the verification on topology with 9 switches and 2 clients has fewer states than the one with 6 switches and 3 clients, each state in the latter case is bigger than the former and hence the memory and communication overheads are higher.

Single Switch Firewall (FW(S)) This is based on an advanced GENI assignment [48] on building an OpenFlow based firewall. The controller takes as input a simple configuration file which is a list of tuples of the form (client1, port1, client2, port2). This specifies that packets originating from client1 on port1 can be forwarded to client2 on port2. We abbreviate the tuples as $(client1\colon port1 \rightarrow client2\colon port2)$. Any flow not explicitly allowed is forbidden. The flows are uni-directional and the above flow will reject traffic initiated by client2 on port2 towards client1 on port1. However, once client1 initiates a flow, the firewall should allow client2 to reply back, making the flow bi-directional until client1 closes the connection.

The naive implementation of the controller is as follows: on receiving a packet $(c1\colon p1 \rightarrow c2\colon p2)$, check if there is a tuple matching the flow in the policy. If it does, add rules $(c1\colon p1 \rightarrow c2\colon p2)$ and $(c2\colon p2 \rightarrow c1\colon p1)$ and forward the packet to $c2$. Otherwise add a rule to drop packets of the form $(c1\colon p1 \rightarrow c2\colon p2)$. The invariant to verify here is to ensure the policy of the firewall, i.e.,

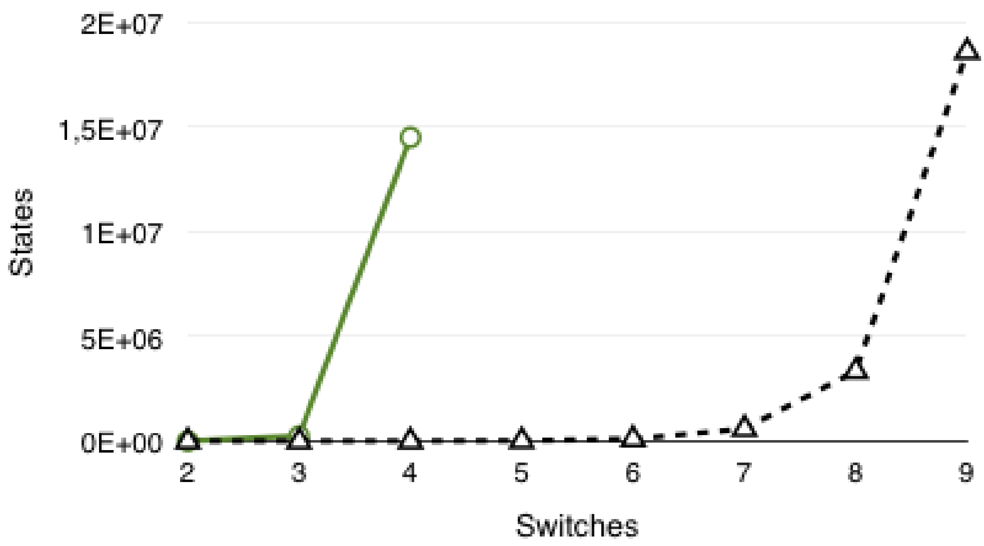


FIGURE 3.3: State space of MAC learning controller.
Δ: optimized, $\circ$: unoptimized

a packet from $c1\colon p1$ should be forwarded to $c2\colon p2$ if and only if $(c1\colon p2 \rightarrow c2\colon p2)$ exists in the firewall policy or if $(c2\colon p2 \rightarrow c1\colon p1)$ exists in the policy and $c2$ has already initiated the corresponding flow. The following formula specifies that allowed packets should not be dropped: $\Box \forall p \in Packet.\ on\_dropped(p) \Rightarrow \neg flows[p.src][packet.src\_port][packet.dest][p.dest\_port]$, where *on_dropped(p)* is set if a packet-drop transition is fired on packet p (and reset at the beginning of every transition). *flows* is an auxiliary variable in the controller which keeps track of allowed flows based on the firewall policy and initiating client.

We ran the experiment on a topology with 2 clients and a firewall. We found an interesting bug in our implementation which is caused by not assigning proper priorities to rules. For example, when $(c1\colon p1 \rightarrow c2\colon p2)$ is present in the policy but not $(c2\colon p2 \rightarrow c1\colon p1)$, the rule to drop flows should have a lower priority than the rules to allow flows. Otherwise, the following bug would occur. If $c2$ initiates the flow $(c2\colon p2 \rightarrow c1\colon p1)$ then the controller adds a rule to drop packets matching that flow. Later on, if $c1$ initiates $(c1\colon p1 \rightarrow c2\colon p2)$ and the controller adds the corresponding rules to allow the flow on both directions, the switch now has two conflicting rules of the same priority. One to allow and the other to drop $(c2\colon p2 \rightarrow c1\colon p1)$. The switch may non-deterministically choose to drop the packet. Once we fixed the bug, the tool could prove the invariant in 5.45 seconds.

Multiple Switch Firewalls (FW(M)) We extend the above example to include multiple replicated firewalls for load balancing. We now allow the clients to send packets to all of these firewalls. We augment the implementation of the single switch controller to add the same rules on all firewalls. However, this implementation no longer ensures the invariant in the multi-switch setting.

Consider the case with two firewalls, $f1$ and $f2$. The tool reports the following bug: $c1$ initiates $(c1\colon p1 \rightarrow c2\colon p2)$ on firewall $f1$. The controller adds the corresponding rules to allow flows in both directions to $f1$ and $f2$ but only sends a barrier to $f1$. Now $f2$ delays the installation of $(c2\colon p2 \rightarrow c1\colon p1)$ and $c2$ replies back to $c1$ through $f2$ which forwards the packet to the controller. The controller then drops the packet.

The fix here is to add the rules along with barriers on all switches and not just the switch from which the packet originates. With this fix the tool is able to prove the property in 8 seconds. In order to test the scalability, we tested the tool on increasing number of firewalls in the topology.

Resonance (RS) Resonance [97] is a system for ensuring security in large networks using OpenFlow switches. When a new client enters the network, it is assigned *registration* state and is only allowed to communicate with a web portal. The portal either authenticates a client by sending a signal to the controller (and the controller assigns the client an *authenticated* state), or sets the client to *quarantined* state. In the authenticated state, the client is only allowed to communicate with a scanner. The scanner ensures that the client is not infected and sends a signal to the controller and lets the controller assign it an *operational* state. If an infection is detected, it is assigned a *quarantined* state. The clients in operational state are periodically scanned and moved to the quarantined state if they are infected. Quarantined clients cannot communicate with other clients.

In our model, the web portal non-deterministically chooses to authenticate or quarantine a client and the scanner non-deterministically marks a client operational or quarantined. We check the invariant that packets from quarantined clients should not be forwarded: $\Box \forall p \in \mathit{Packet}.\ \mathit{on\_forward}(p) \Rightarrow (\mathit{state}(p.src) \neq \mathit{Quarantined})$. Similar to *on_drop*, *on_forward* is set when packet-forward transition is fired and reset before the beginning of every transition. The controller follows the Resonance algorithm [97].

We ran the experiment on a topology of two clients, one portal, one scanner and four switches. The topology is the same as in Figure 2 of [97] without DHCP and DNS clients. KUAI proves the invariant in 6.6 seconds. We scale up the example by increasing the number of clients.

Simple (SIM) Simple [105] is a policy enforcement layer built on top of OpenFlow to ensure efficient middlebox traffic steering. In many network settings, traffic is routed through several middleboxes, such as firewalls, loggers, proxies, etc., before reaching the final destination. Simple takes a middlebox policy as input and translates this to forwarding rules to ensure the policy holds. The invariant ensures that all source packets to a client will be received and forwarded by the middleboxes specified in a given policy before the packet reaches its destination.

We ran the experiment on a topology of two clients, two firewalls, one IDS, one proxy and five switches (see Figure 1 of [105]). KUAI can prove the invariant in 6.48 seconds.

We scale up the example by fixing the destination client and increasing the number of source clients that can send packets to it. Because of our "all packets in one shot" optimization (section 3.3.4), no matter how many packets get queued initially, they are all forwarded in lock-step as the controller forwarding rule applies to all incoming packets.

3.5 Proof Details

3.5.1 Proofs for Barrier Optimization

To ease the proof of Theorem 5, we first provide several lemmas. Lemmas 4 and 5 provide two properties of a barrier action.

Lemma 4. *Let $TS(\mathcal{N}) = (S, A, \rightarrow, s_0, AP, L)$ be an action-deterministic transition system. For each $1 \leq i \leq 2^{|Rule|}$, for all $sw \in Switches$, $barrier(sw)_i$ is independent of $A \backslash Barrier$.*

Proof. It is straightforward to check the correctness of this lemma by using the definition of independence between actions. □

Lemma 5. *Let $TS(\mathcal{N}) = (S, A, \rightarrow, s_0, AP, L)$ be an action-deterministic transition system and an SDN specification $\Box\phi$. For each $1 \leq i \leq 2^{|Rule|}$ and $sw \in Switches$, $barrier(sw)_i$ is a stutter action w.r.t. $\Box\phi$.*

Proof. ϕ is a proposition over packets that have been forwarded by some switch at least once, or over control states. Since $barrier(sw)_i$ does not change packets or control states, $barrier(sw)_i$ is a stutter action. □

Lemma 6 shows the definition of ample set in $TS(\mathcal{N})$ satisfies three conditions.

Lemma 6. *$ample(s)$ satisfies the following conditions.*

1. *$\emptyset \neq ample(s) \subseteq A(s)$.*
2. *Let $s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ be a finite execution in $TS(\mathcal{N})$. If $\alpha \in A \backslash ample(s)$ depends on $ample(s)$, $\beta_i \in ample(s)$ for some $0 < i \leq n$.*
3. *If $ample(s) \neq A(s)$ then any $\alpha \in ample(s)$ is a stutter action.*

Proof. Conditions (1) and (3) are straightforward to verify.

Let us prove condition (2) by contradiction. Suppose (2) does not hold. Then there is a finite execution $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ in $TS(\mathcal{N})$ such that for any $1 \leq i \leq n$, $\beta_i \notin ample(s)$ and α depends on $ample(s)$.

If $ample(s) = A(s)$, then $\beta_1 \in ample(s)$, which leads to a contradiction. Otherwise $ample(s) = \{barrier(sw)_i \mid 1 \leq i \leq b(s, sw) \wedge sw$ has a barrier in $s\}$. Since α depends on $ample(s)$, by Lemma 4, α can only be a barrier action. Since for any $1 \leq i \leq n$, $\beta_i \notin ample(s)$, β_i is not a barrier action. Hence $\alpha \in A(s)$. By the definition of $ample(s)$, $\alpha \in ample(s)$, which leads to a contradiction. Therefore condition (2) holds. □

Lemma 6 implies the following three lemmas from 7 to 9.

Lemma 7. *Let s be a state in $TS(\mathcal{N})$. If $\alpha \in ample(s)$, then α is independent of $A(s)\backslash ample(s)$.*

Proof. Suppose not. Then there is an action $\beta \in A(s)\backslash ample(s)$ such that α and β are dependent. Since $\beta \in A(s)$, then $s \xrightarrow{\beta} s_1$ is a finite execution in $TS(\mathcal{N})$. However it violates the condition (2) in Lemma 6. □

Lemmas 8 and 9 explain two ways to obtain a stutter equivalent execution.

Lemma 8. *Let ρ be a finite execution in $TS(\mathcal{N})$ of the form $s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$ where $\beta_i \notin ample(s)$, for $0 < i \leq n$, and $\alpha \in ample(s)$. There exists a finite execution ρ' of the form $s \overset{\alpha}{\Rightarrow} t_0 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{n-1}} t_{n-1} \xrightarrow{\beta_n} t$ and $\rho \triangleq \rho'$.*

Proof. We prove it by induction on $i \geq 1$.

Base case ($i = 1$): Then $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\alpha} t$. Since $\beta_1 \notin ample(s)$ and $\alpha \in ample(s)$ by Lemma 7, we have β_1 and α are independent. Hence we can permute them and get $\rho' = s \xrightarrow{\alpha} t_1 \xrightarrow{\beta_1} t$. Since $\alpha \in ample(s)$, we have $\rho' = s \overset{\alpha}{\Rightarrow} t_1 \xrightarrow{\beta_1} t$. Moreover, since α is a barrier action and it is a stutter action, we have $\rho \triangleq \rho'$.

Induction step ($i = n$): Let $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\beta_{n+1}} s_{n+1} \xrightarrow{\alpha} t$. Since $\beta_{n+1} \notin ample(s)$ and $\alpha \in ample(s)$, by Lemma 7, β_{n+1} and α are independent. Hence we have $\hat{\rho} = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t_n \xrightarrow{\beta_{n+1}} t$. Since α is a stutter action, $\rho \triangleq \hat{\rho}$. Let $\hat{\rho}(n) = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t_n$. By induction hypothesis, there is a $\rho'(n) = s \overset{\alpha}{\Rightarrow} t_0 \xrightarrow{\beta_1} t_1 \ldots \xrightarrow{\beta_{n-1}} t_{n-1} \xrightarrow{\beta_n} t_n$ such that $\hat{\rho}(n) \triangleq \rho'(n)$. Then we have $\rho' = s \overset{\alpha}{\Rightarrow} t_0 \xrightarrow{\beta_1} t_1 \ldots \xrightarrow{\beta_{n-1}} t_{n-1} \xrightarrow{\beta_n} t_n \xrightarrow{\beta_{n+1}} t$ and $\rho' \triangleq \hat{\rho} \triangleq \rho$. $\square$

Lemma 9. *Let $\rho = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots$ be an infinite execution in $TS(\mathcal{N})$ where $\beta_i \notin ample(s)$, for $i > 0$. There exists an execution ρ' of the form $s \overset{\alpha}{\Rightarrow} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots$ where $\alpha \in ample(s)$ and $\rho \triangleq \rho'$.*

Proof. Since for all $i > 0$, $\beta_i \notin ample(s)$ and $\alpha \in ample(s)$, by Lemma 7, β_i and α are independent. Hence we have $\rho' = s \overset{\alpha}{\Rightarrow} t_0 \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \ldots$ where for each $i > 0$, $\alpha(s_i) = t_i$. Since α is a stutter action, for each $i > 0$, $L(s_i) = L(t_i)$ and $L(s) = L(t_0)$. Hence $\rho \triangleq \rho'$. $\square$

The transition system $\widehat{TS}$ has the following property in Lemma 10.

Lemma 10. *For any infinite execution ρ in $\widehat{TS}$, there are infinitely many state s in ρ such that $ample(s) = A(s)$.*

Proof. Suppose not. Without loss of generality, assume that from the k-th state s_k on, all the states after s_k in ρ are such that $ample(s) \neq A(s)$. Then we have for all $i > k$, the action taken from s_i is a barrier action. However, s_k has finitely many barriers, which implies that ρ cannot be infinite. Contradiction. $\square$

Finally, we prove our main theorem for the barrier optimization:

Theorem 11. *Let $TS(\mathcal{N})$ be an action-deterministic transition system. $TS(\mathcal{N}) \triangleq \widehat{TS}$.*

Proof. By the definition of $\Rightarrow$, we know that every execution in $\widehat{TS}$ is also an execution in $TS(\mathcal{N})$, and hence $\widehat{TS} \trianglelefteq TS(\mathcal{N})$.

We now prove that $TS(\mathcal{N}) \trianglelefteq \widehat{TS}$, that is, for any initial infinite execution ρ in $TS(\mathcal{N})$, there is an initial infinite execution ρ' in $\widehat{TS}$ such that $\rho \triangleq \rho'$. The idea is the following. Let ρ be an infinite initial execution in $TS(\mathcal{N})$ that is not in $\widehat{TS}$. Let l be the minimal index in ρ such that for all $1 \leq i \leq l$, the transition $s_{i-1} \xrightarrow{\mu_i} s_i$ is also a transition $s_{i-1} \xRightarrow{\mu_i} s_i$ in $\widehat{TS}$, that is,

$$\rho = s_0 \xRightarrow{\mu_1} \ldots \xRightarrow{\mu_l} \underbrace{s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} s_2 \ldots}_{\rho_0}$$

Let ρ_0 be an execution in $TS(\mathcal{N})$ which starts in state s and is induced by the action sequence $\beta_1\beta_2\beta_3\ldots$ where $\beta_1 \notin ample(s)$. The execution ρ_0 is successively replaced with stutter-equivalent executions ρ_m, $m = 1, 2, 3, \ldots$, by means of the transformations indicated in Lemmas 8 and 9. Each of these executions ρ_m starts in state s and is based on an action sequence of the form $\alpha_1 \ldots \alpha_m \beta_1 \gamma_1 \gamma_2 \gamma_3 \ldots$ The action sequence $\alpha_1 \ldots \alpha_m$ contains the actions of the ample sets, which are newly inserted according to Lemma 9, and all actions β_n, which were shifted forward according to Lemma 8. $\gamma_1\gamma_2\gamma_3\ldots$ denotes the remaining subsequence of $\beta_1, \beta_2, \beta_3, \ldots$. Thus, ρ_m is of the form $s \xRightarrow{\alpha_1} t_1 \xRightarrow{\alpha_2} \ldots \xRightarrow{\alpha_m} t_m \xrightarrow{\beta_1} t_0^m \xrightarrow{\gamma_1} t_1^m \xrightarrow{\gamma_2} t_2^m \xrightarrow{\gamma_3} \ldots$ where $\alpha_1, \ldots \alpha_m$ are stutter actions. By Lemma 10, $\beta_1 \in ample(t_m)$ for some $m \geq 1$. Then $s \xRightarrow{\alpha_1} t_1 \xRightarrow{\alpha_2} \ldots \xRightarrow{\alpha_m} t_m \xRightarrow{\beta_1} t_0^m$ becomes an execution in $\widehat{TS}$. By repeating this reasoning to the rest of the execution $t_0^m \xrightarrow{\gamma_1} t_1^m \xrightarrow{\gamma_2} t_2^m \xrightarrow{\gamma_3} \ldots$, we obtain an execution ρ_0' in $\widehat{TS}$ (as the "limit" of $\rho_m, \rho_{m+1}, \ldots$), where the induced action sequence contains all actions that occur in ρ_0 (in TS).

Let us assume that ρ has the form $s_0 \xrightarrow{\xi_1} s_1 \xrightarrow{\xi_2} \ldots$ and let $0 = k_0 < k_1 < k_2 < \ldots$ such that $\xi_{k_1}, \xi_{k_2}, \ldots$ results from $\xi_1, \xi_2, \ldots$ by omitting all stutter actions in $\xi_1, \xi_2, \ldots$. Then, $trace(\rho)$ has the form $A_0^+ A_1^+ A_2^+ \ldots$, where A_i is the label $L(s_k)$ of all states s_k with $k_i \leq k < k_{i+1}$. Since each of the nonstutter actions ξ_{k_i} is eventually processed, when generating the executions $\rho_1, \rho_2, \rho_3, \ldots$, for each index k_i there is some finite sequence

w_i of the form $A_0^+ A_1^+ \dots A_i^+$ and some index l_i such that the traces of the executions ρ_j for all $j \geq l_i$ start with w_i. In particular, w_i is a proper prefix of w_{i+1} and w_i are prefixes of the trace associated with the limit execution ρ'. Hence, $trace(\rho')$ has the form $A_0^+ A_1^+ A_2^+ \dots$, and $\rho \triangleq \rho'$. □

Theorem 12. *Given an SDN $\mathcal{N}$ and a safety property $\Box\phi$, $TS(\mathcal{N})$ satisfies $\Box\phi$ iff TS_2 satisfies $\Box\phi$.*

Proof. If $TS(\mathcal{N})$ does not satisfy $\Box\phi$, then there is an execution $s_0 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_n$ in $TS(\mathcal{N})$ such that $L(s_n)$ does not satisfy ϕ. Since $TS(\mathcal{N}) \trianglelefteq TS_2$, there is an execution $s_0 \xrightarrow{\beta_1}_2 \dots \xrightarrow{\beta_m}_2 t_m$ in TS_2 such that $L(t_m) = L(s_n)$. Hence $L(t_m)$ does not satisfy ϕ either. Hence TS_2 does not satisfy $\Box\phi$.

We can prove the other direction analogously. □

3.5.2 Proofs for Client Optimization

The proofs for client optimization mimic the ones in the barrier optimization above. We first show that a receive action is independent of any other actions in Lemma 11 and is a stutter action in Lemma 12.

Lemma 11. *Let $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$ be an action-deterministic transition system. Any receive action $recv(c, pkt, pkts)$ is independent of $A_2 \backslash \{recv(c, pkt, pkts)\}$.*

Proof. It is straightforward to check the correctness of this lemma by using the definition of independence between actions. □

Lemma 12. *Let $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$ be an action-deterministic transition system and an SDN specification $\Box\phi$. Any receive action $recv(c, pkt, pkts)$ is a stutter action w.r.t. $\Box\phi$.*

Proof. ϕ is a proposition over packets that have been forwarded by some switch at least once or over control states. Since any packet sent to the network by a receive action has not been forwarded yet, and a receive action does not change control states, any receive action is a stutter action. □

Lemma 13 shows the definition of ample set in TS_2 satisfies three conditions.

Lemma 13. *$ample(s)$ satisfies the following conditions.*

1. *$\emptyset \neq ample(s) \subseteq A(s)$.*

2. *Let $s \xrightarrow{\beta_1}_2 s_1 \xrightarrow{\beta_2}_2 \ldots \xrightarrow{\beta_n}_2 s_n \xrightarrow{\alpha}_2 t$ be a finite execution in TS_2. If $\alpha \in A \backslash ample(s)$ depends on $ample(s)$, $\beta_i \in ample(s)$ for some $0 < i \leq n$.*

3. *If $ample(s) \neq A(s)$ then any $\alpha \in ample(s)$ is a stutter action.*

Proof. Conditions (1) and (3) are straightforward to verify.

Let us prove condition (2) by contradiction. Suppose (2) does not hold. Then there is a finite execution $\rho = s \xrightarrow{\beta_1}_2 s_1 \xrightarrow{\beta_2}_2 \ldots \xrightarrow{\beta_n}_2 s_n \xrightarrow{\alpha}_2 t$ in TS_2 such that for any $1 \leq i \leq n$, $\beta_i \notin ample(s)$ and α depends on $ample(s)$.

If $ample(s) = A(s)$, then $\beta_1 \in ample(s)$, which leads to a contradiction. Otherwise $ample(s)$ contains only receive actions. By Lemma 11, α is a receive action. Since for all $1 \leq i \leq n$, β_i is not a receive action, $\alpha \in A(s)$. Hence $\alpha \in ample(s)$, which leads to a contradiction. Therefore condition (2) holds. □

The transition system TS_3 has the following property in Lemma 14.

Lemma 14. *For any infinite execution ρ in TS_3, there are infinitely many state s in ρ such that $ample(s) = A(s)$.*

Proof. Suppose not. Without loss of generality, assume that from the k-th state s_k on, all the states after s_k in ρ are such that $ample(s) \neq A(s)$. Then we have for all $i > k$, the action taken from s_i is a receive action. However, there are finitely many packets in packet queues of clients in s_k, which implies that ρ cannot be infinite. Contradiction. □

Finally, we prove our main theorem for the client optimization:

Theorem 13. *(1) $TS_2 \triangleq TS_3$. (2) Given a safety property $\Box\phi$, TS_2 satisfies $\Box\phi$ iff TS_3 satisfies $\Box\phi$.*

Proof. Since Lemmas 13 and 14 hold, we can mimic the proof for Theorem 11 and prove claim (1). By claim (1), claim (2) holds immediately. □

3.5.3 Proofs for $(0, \infty)$ Abstraction

Semantics

Given a flow table ft and a list l in $(\mathbb{M}[CM]^{\infty} \cup b)^*$, let $update^e(ft, l)$ be a procedure that updates ft based on l as follows. It dequeues the head of l and sets l to $l.tl$. If the head is a barrier, then ignore it. If it is an extended multiset m, it nondeterministically chooses a fetching order p and based on p, removes a control message cm with $m(cm) = \infty$ from m and set $m(cm) = 0$. If cm is $\mathsf{add}(r)$, then add the rule r to ft, or if cm is $\mathsf{del}(r)$, then delete r from ft. It keeps updating ft based on p until m becomes empty. It repeats the above instructions on l until l becomes empty. Then it returns the resulting flow table ft.

We define the following operations for over-approximation:

1. Add packets in switches or clients. Given a set $X \subseteq Switches \times Packet^{\mathbb{N}}$, define $addPkt^e(\delta, X) = \delta[\text{ foreach } (sw, pkt^k) \in X, sw \mapsto \delta(sw)[pq \mapsto \mathsf{over}(\delta(sw).pq \oplus [\![pkt^k]\!])]]$. Given a set $Y \subseteq Clients \times Packet^{\mathbb{N}}$, define $addPkt^e(\pi, Y) = \pi[\text{ foreach } (c, pkt^k) \in Y, c \mapsto \mathsf{over}(\pi(c) \oplus [\![pkt^k]\!])]$.

2. Flush and run all control messages up to the last barrier in a switch. Define $flushall^e(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l_1; ft \mapsto update^e(\delta(sw).ft, l_2)]]$ where $l_1 = [\emptyset]$ and $l_2 = \delta(sw).cq$ if the last element in $\delta(sw).cq$ is a barrier. Otherwise, let $\delta(sw).cq = l@[m]$. Then $l_1 = [m]$ and $l_2 = l$.

3. Add control messages and barriers to the control queues of the switches. Given a total function $f : Switches \to RE$, define $addCtrlCmd^e(\delta, f) = \delta[\text{ foreach } sw \in Switches : sw \mapsto \delta(sw)[cq \mapsto \mathsf{over}(\delta(sw).cq + f(sw))]]$.

For an SDN and the transition system $TS_3 = (S_3, A_3, \to_3, s_0, AP_3, L_3)$, define a transition system $TS_4 = (S_4, A_4, \to_4, s_0, AP_4, L_4)$ where $S_4 = \{\mathsf{over}(s) \mid s \in S_3\}$, $A_4 = A_3$, $AP_4 = AP_3$, $L_4 = L_3$, and for $t, t' \in S_4$, $t \xrightarrow{\alpha}_4 t'$ is defined as

1. $\alpha = send(c, pkt)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi, \delta', cs, rq)$ where $\delta' = addPkt^e(\delta, \{sw, pkt\})$ and $sw = pkt.loc.n$.

2. $\alpha = \mathit{recv}(c, \mathit{pkt}, \mathit{pkts})$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi', \delta', cs, rq)$ where $\pi' = \mathit{delPkt}(\pi, \{c, \mathit{pkt}\})$, $\delta' = \mathit{addPkt}^e(\delta, X)$ and $X = \{(sw, \mathit{pkt}'^k) \mid \mathit{pkts}(\mathit{pkt}') = k \wedge \mathit{pkt}'.loc.n = sw\}$.

3. $\alpha = \mathit{match}(sw, \mathit{pkt}, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi', \delta', cs, rq)$ where $\pi' = \mathit{addPkt}^e(\pi, \mathit{FwdToC}(sw, \mathit{pkt}, r.\mathit{ports}))$ and $\delta' = \mathit{addPkt}^e(\delta, \mathit{FwdToSw}(sw, \mathit{pkt}, r.\mathit{ports}))$.

4. $\alpha = \mathit{nomatch}(sw, \mathit{pkt})$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi, \delta', cs, rq')$ where $\delta'' = \mathit{delPkt}(\delta, \{sw, \mathit{pkt}\})$, $\delta' = \mathit{setWait}(\delta'', sw)$, and $rq' = rq \cup \{\mathit{pkt}\}$.

5. $\alpha = \mathit{add}(sw, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi, \delta', cs, rq)$ where $\delta' = \mathit{addRule}(\delta, sw, r)$.

6. $\alpha = \mathit{del}(sw, r)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi, \delta', cs, rq)$ where $\delta' = \mathit{delRule}(\delta, sw, r)$.

7. $\alpha = \mathit{fwd}(sw, \mathit{pkt}, \mathit{pts})$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi', \delta', cs, rq)$ where $\pi' = \mathit{addPkt}^e(\pi, \mathit{FwdToC}(sw, \mathit{pkt}, \mathit{pts}))$, $\delta_1 = \mathit{delFwdMsg}(\delta, sw, (\mathit{pkt}, \mathit{pts}))$, $\delta_2 = \mathit{addPkt}^e(\delta_1, \mathit{FwdToSw}(sw, \mathit{pkt}, \mathit{pts}))$, and $\delta' = \mathit{unsetWait}(\delta_2, sw)$.

8. $\alpha = \mathit{ctrl}(\mathit{pkt}, cs)$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha}_4 (\pi, \delta', cs', rq')$ where $rq' = rq \backslash \{\mathit{pkt}\}$. Let $\mathit{pktIn}(\mathit{pkt}, cs) = (\eta, msg, cs')$ and $sw = \mathit{pkt}.loc.n$. Define $\delta'' = \mathit{addFwdMsg}(\delta, sw, msg)$, and $\delta''' = \mathit{addCtrlCmd}^e(\delta'', \eta)$. Let $\{sw_1, \ldots, sw_n\}$ be the set of all switches whose control queue has barriers in δ'''. Let $\delta_0 = \delta'''$ and $\delta_i = \mathit{flushall}^e(\delta_{i-1}, sw_i)$ for all $1 \le i \le n$. Define $\delta' = \delta_n$.

Proofs

Lemma 15. *Given a flow table ft and a multiset m of control messages. For any fetching order p chosen by $\mathit{update}(\mathit{ft}, [m])$ to process m, there is a fetching order p' chosen by $\mathit{update}^e(\mathit{ft}, [\mathsf{over}(m)])$ to process $\mathsf{over}(m)$ such that $\mathit{update}(\mathit{ft}, [m]) = \mathit{update}^e(\mathit{ft}, [\mathsf{over}(m)])$.*

Proof. Let $R = \{r \mid m(\mathsf{add}(r)) > 0 \vee m(\mathsf{del}(r)) > 0\}$ be the set of rules that are manipulated by m. Fix a fetching order p for update. We construct a fetching order p' for update^e such that for each $r \in R$, if p adds r to *ft*, then p' also adds r to *ft*; and if p deletes r from *ft*, then p' deletes r from *ft* too. As a result, $\mathit{update}(\mathit{ft}, [m]) = \mathit{update}^e(\mathit{ft}, [\mathsf{over}(m)])$.

Fix a rule $r \in R$. If p adds r to *ft*, then there are two cases to consider.

1. $m(\mathsf{add}(r)) > 0$ and $m(\mathsf{del}(r)) = 0$. Then we have $\mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $\mathsf{over}(m)(\mathsf{del}(r)) = 0$. Then any fetching order of $update^e(ft, [\mathsf{over}(m)])$ including p' adds r to ft.

2. $m[\mathsf{add}(r)] > 0$ and $m[\mathsf{del}(r)] > 0$. Then we have $\mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $\mathsf{over}(m)(\mathsf{del}(r)) = \infty$. We require that p' fetch $\mathsf{del}(r)$ first and then $\mathsf{add}(r)$. Hence p' adds r to ft as well.

If p deletes r from ft, then there are two cases to consider.

1. $m(\mathsf{add}(r)) = 0$ and $m(\mathsf{del}(r)) > 0$. Then we have $\mathsf{over}(m)(\mathsf{add}(r)) = 0$ and $\mathsf{over}(m)(\mathsf{del}(r)) = \infty$. Then any fetching order of $update^e(ft, [\mathsf{over}(m)])$ including p' deletes r from ft.

2. $m(\mathsf{add}(r)) > 0$ and $m(\mathsf{del}(r)) > 0$. Then we have $\mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $\mathsf{over}(m)(\mathsf{del}(r)) = \infty$. We require that p' fetch $\mathsf{add}(r)$ first and then $\mathsf{del}(r)$. Hence p' deletes r from ft as well.

Consequently, if $update(ft, [m])$ chooses p, then let $update^e(ft, [\mathsf{over}(m)])$ choose p', and we have $update(ft, [m]) = update^e(ft, [\mathsf{over}(m)])$. □

We can naturally extend the notion of fetching order from a single multiset to a list l in $(\mathbb{M}[CM] \cup b)^*$. Given l, let l' be the list of multisets $[m_1, \ldots, m_n]$ obtained by removing all barriers from l. Let p_i be a fetching order for m_i, where $1 \leq i \leq n$. Define a fetching order of $update(ft, l)$ to be $p_1, p_2, \ldots, p_n$, that is, the composition of individual fetching orders one by one. Given a list l in $(\mathbb{M}[CM]^\infty \cup b)^*$, we define a fetching order of $update^e(ft, l)$ analogously. We now can extend Lemma 15 to a list l in $(\mathbb{M}[CM] \cup b)^*$ as a corollary.

Corollary 1. *Given a flow table ft and a list l in $(\mathbb{M}[CM] \cup b)^*$. for any fetching order p of $update(ft, l)$, there is a fetching order p' of $update^e(ft, \mathsf{over}(l))$ such that $update(ft, l) = update^e(ft, \mathsf{over}(l))$.*

The following lemma claims two properties between finite initial executions in TS_3 and TS_4.

Lemma 16. *Let* $s_0 \xrightarrow{\alpha_1}_3 s_1 \xrightarrow{\alpha_2}_3 s_2 \ldots \xrightarrow{\alpha_n}_3 s_n$ *be an initial execution in* TS_3 *and* $t_0 \xrightarrow{\alpha_1}_4 t_1 \xrightarrow{\alpha_2}_4 t_2 \ldots \xrightarrow{\alpha_n}_4 t_n$ *be an initial execution in* TS_4 *for some* $n \geq 0$*. For all* $0 \leq i \leq n$*, let* $s_i = (\pi_i, \delta_i, cs_i, rq_i)$ *and* $t_i = (\pi'_i, \delta'_i, cs'_i, rq'_i)$*. The following two properties hold:*

1. *For all* $0 \leq i \leq n$*,* $sw \in Switches$*, and rule* $r \in Rule$*, if* $\delta_i(sw).cq = [m_i] \wedge m_i(\mathsf{add}(r)) = 0 \wedge m_i(\mathsf{del}(r)) = 0 \wedge \delta'_i(sw).cq = [m'_i] \wedge m'_i(\mathsf{add}(r)) = \infty \wedge m'_i(\mathsf{del}(r)) = 0$*, then* $r \in \delta_i(sw).ft$ *and* $r \in \delta'_i(sw).ft$*.*

2. *For all* $0 \leq i \leq n$*,* $sw \in Switches$*, and rule* $r \in Rule$*, if* $\delta_i(sw).cq = [m_i] \wedge m_i(\mathsf{add}(r)) = 0 \wedge m(\mathsf{del}(r)) = 0 \wedge \delta'_i(sw).cq = [m'_i] \wedge m'_i(\mathsf{del}(r)) = \infty \wedge m'_i(\mathsf{add}(r)) = 0$*, then* $r \notin \delta_i(sw).ft$ *and* $r \notin \delta'_i(sw).ft$*.*

Proof. We now prove property 1 by induction on n.

Base case ($n = 0$): the property 1 holds trivially.

Induction step: suppose for all $0 \leq i \leq n$, property 1 holds. We now prove that property 1 also holds for two executions of length $n+1$. Suppose we have two initial executions $s_0 \xrightarrow{\alpha_1}_3 s_1 \xrightarrow{\alpha_2}_2 s_2 \ldots \xrightarrow{\alpha_n}_3 s_n \xrightarrow{\alpha_{n+1}}_3 s_{n+1}$ and $t_0 \xrightarrow{\alpha_1}_4 t_1 \xrightarrow{\alpha_2}_4 t_2 \ldots \xrightarrow{\alpha_n}_4 t_n \xrightarrow{\alpha_{n+1}}_4 t_{n+1}$. Without loss of generality, fix a switch sw and a rule r. Assume that $\delta_{n+1}(sw).cq = [m_{n+1}] \wedge m_{n+1}(\mathsf{add}(r)) = 0 \wedge m_{n+1}(\mathsf{del}(r)) = 0 \wedge \delta'_{n+1}(sw).cq = [m'_{n+1}] \wedge m'_{n+1}(\mathsf{add}(r)) = \infty \wedge m'_{n+1}(\mathsf{del}(r)) = 0$ holds. Our goal is to prove $r \in \delta_{n+1}(sw).ft$ and $r \in \delta'_{n+1}(sw).ft$. We prove it by case analysis on the action α_{n+1}.

1. α_{n+1} is in $Send \cup Recv \cup Match \cup NoMatch \cup Forward$, or is of the form $add(sw', r')$ or $del(sw', r')$ where $sw' \in Switches$ and $r' \neq r$. Then we have $\delta_n(sw).cq = [m_n] \wedge m_n(\mathsf{add}(r)) = 0 \wedge m_n(\mathsf{del}(r)) = 0 \wedge \delta'_n(sw).cq = [m'_n] \wedge m'_n(\mathsf{add}(r)) = \infty \wedge m'_n(\mathsf{del}(r)) = 0$. By induction hypothesis, $r \in \delta_n(sw).ft$ and $r \in \delta'_n(sw).ft$. Since α_{n+1} does not add or delete r, we have $r \in \delta_{n+1}(sw).ft$ and $r \in \delta'_{n+1}(sw).ft$.

2. α_{n+1} is the action $add(sw, r)$. Then $r \in \delta_{n+1}(sw).ft$ and $r \in \delta'_{n+1}(sw).ft$ trivially hold because r is added by α_{n+1}.

3. α_{n+1} is the action $del(sw, r)$. This case is impossible because this implies that $\delta'_{n+1}(sw).cq = [m'_{n+1}]$ and $m'_{n+1}(\mathsf{del}(r)) = \infty$.

4. α_{n+1} is a control action $ctrl(pkt, cs)$. There are three cases that α_{n+1} may modify the control queue cq of the switch sw.

(1) α_{n+1} appends nothing to cq. Then by the same argument in (1), we have $r \in \delta_{n+1}(sw).ft$ and $r \in \delta'_{n+1}(sw).ft$.

(2) α_{n+1} appends a multiset m'' to cq. Since $m_{n+1}(\mathsf{add}(r)) = m_{n+1}(\mathsf{del}(r)) = 0$ at s_{n+1}, we have $m''(\mathsf{add}(r)) = m''(\mathsf{del}(r)) = 0$, and $\delta_n(sw).cq = [m_n] \wedge m_n(\mathsf{add}(r)) = 0 \wedge m_n(\mathsf{del}(r)) = 0$. Since $m'_{n+1}(\mathsf{add}(r)) = \infty$ and $m'_{n+1}(\mathsf{del}(r)) = 0$ at t_{n+1} and $m''(\mathsf{add}(r)) = 0$, we have $\delta'_n(sw).cq = [m'_n] \wedge m'_n(\mathsf{add}(r)) = \infty \wedge m'_n(\mathsf{del}(r)) = 0$. By induction hypothesis, $r \in \delta_n(sw).ft$ and $r \in \delta'_n(sw).ft$. Since α_{n+1} does not change any flow table, we have $r \in \delta_{n+1}(sw).ft$ and $r \in \delta'_{n+1}(sw).ft$.

(3) α_{n+1} appends more than one multiset to cq. This case is impossible because a barrier must be appended in cq. Then $m'_{n+1}(\mathsf{add}(r)) = \infty$ at t_{n+1} implies that $m_{n+1}(\mathsf{add}(r)) \neq 0$ at s_{n+1}, which is a contradiction.

Since the proof of property 2 is analogous to the proof of property 1 above, we skip it now. □

We now prove TS_4 simulates TS_3 in the following lemma.

Lemma 17. *For any infinite initial execution $s_0 \xrightarrow{\beta_1}_3 s_1 \xrightarrow{\beta_2}_3 s_2 \ldots$ in TS_3, there is an infinite initial execution $t_0 \xrightarrow{\beta_1}_4 t_1 \xrightarrow{\beta_2}_4 t_2 \ldots$ in TS_4 such that for all $i \geq 0$, $s_i = (\pi_i, \delta_i, cs_i, rq_i)$ and $t_i = (\pi'_i, \delta'_i, cs'_i, rq'_i)$ satisfy the following condition: for all $c \in Clients$, $\pi_i(c) \leq_e \pi'_i(c)$ and for all $sw \in Switches$, $\delta_i(sw).pq \leq_e \delta'_i(sw).pq$, $\delta_i(sw).cq \leq_e \delta'_i(sw).cq$, $\delta_i(sw).fq = \delta'_i(sw).fq$, $\delta_i(sw).ft = \delta'_i(sw).ft$, and $\delta_i(sw).wait = \delta'_i(sw).wait$, and $cs_i = cs'_i$, and $rq_i = rq'_i$.*

Proof. Induction on the length k of an initial execution in TS_3.

Base case ($k = 0$): it holds because $s_0 = t_0$.

Induction step: Suppose the theorem holds for $k = n$. Consider an initial execution $s_0 \xrightarrow{\beta_1}_3 s_1 \xrightarrow{\beta_2}_3 s_2 \ldots \xrightarrow{\beta_n}_3 s_n \xrightarrow{\beta_{n+1}}_3 s_{n+1}$ in TS_3. By induction hypothesis, there is an initial execution $t_0 \xrightarrow{\beta_1}_4 t_1 \xrightarrow{\beta_2}_4 t_2 \ldots \xrightarrow{\beta_n}_4 t_n$ in TS_4 such that if for $0 \leq i \leq n$, s_i and t_i satisfy the condition. Hence β_{n+1} is enabled at t_n. Suppose $\beta_{n+1}(t_n) = t_{n+1}$. Our goal is to prove s_{n+1} and t_{n+1} also satisfy the condition. We prove it by case analysis on β_{n+1}. It is easy to check if all actions except control actions are taken from s_n and get s_{n+1}, the same action can also taken from t_n and get t_{n+1}, and s_{n+1} and t_{n+1} satisfy the condition.

Let us consider the case where a control action β_{n+1} is taken from s_n to s_{n+1}. By induction hypothesis, $rq'_n = rq_n$ and $cs'_n = cs_n$. Hence $rq'_{n+1} = rq_{n+1}$ and $cs'_{n+1} = cs_{n+1}$. Without loss of generality, it is sufficient to analyze one switch sw and show $\delta_{n+1}(sw).cq \leq_e \delta'_{n+1}(sw).cq$ and $\delta_{n+1}(sw).ft = \delta'_{n+1}(sw).ft$.

Denote l be the list in $(\mathbb{M}[CM] \cup b)^*$ that is added to the switch sw by the controller. If l is empty, then cq and ft do not change after β_{n+1} is taken, that is, $\delta_{n+1}(sw).cq = \delta_n(sw).cq \leq_e \delta'_n(sw).cq = \delta'_{n+1}(sw).cq$, and $\delta_{n+1}(sw).ft = \delta_n(sw).ft = \delta'_n(sw).ft = \delta'_{n+1}(sw).ft$. If l is not empty, we consider three cases:

1. $l = [m]$. Since l has no barriers, $\delta'_{n+1}(sw).ft = \delta'_n(sw).ft = \delta_n(sw).ft = \delta_{n+1}(sw).ft$. In addition, $\delta_{n+1}(sw).cq = [\delta_n(sw).cq.hd \oplus m] \leq_e [\delta'_n(sw).cq.hd \oplus \mathsf{over}(m)] = \delta'_{n+1}(sw).cq$

2. $l = [m]@l_2@[b, \hat{m}]$ or $[m]@l_2@[b]$. Then $\delta_{n+1}(sw).cq \leq_e \delta'_{n+1}(sw).cq$. Suppose $\delta_n(sw).cq = [m_0]$ and $\delta'_n(sw).cq = [m'_0]$. we show that for any fetching order p of $update(\delta_n(sw).ft, m_0 \oplus m)$ that leads to $\hat{ft}$, there is a fetching order p' of $update^e(\delta'_n(sw).ft, m'_0 \oplus \mathsf{over}(m))$ that leads to $\hat{ft}'$ such that $\hat{ft} = \hat{ft}'$. Fix a fetching order p. We construct p' based on each rule $r \in Rule$.

   (a) r is added to $\delta_n(sw).ft$ by p. There are two possibilities.

      i. $m_0 \oplus m(\mathsf{add}(r)) > 0$ and $m_0 \oplus m(\mathsf{del}(r)) = 0$. Then we have $m'_0 \oplus \mathsf{over}(m)(\mathsf{add}(r)) = \infty$. If $m'_0 \oplus \mathsf{over}(m)(\mathsf{del}(r)) = 0$, then we do not restrict p' because r must be added. If $m'_0 \oplus \mathsf{over}(m)(\mathsf{del}(r)) = \infty$, then we require p' fetch $\mathsf{del}(r)$ first and then $\mathsf{add}(r)$.

      ii. $m_0 \oplus m(\mathsf{add}(r)) > 0$ and $m_0 \oplus m(\mathsf{del}(r)) > 0$. Then we have $m'_0 \oplus \mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $m'_0 \oplus \mathsf{over}(m)(\mathsf{del}(r)) = \infty$. We require p' fetch $\mathsf{del}(r)$ first and then $\mathsf{add}(r)$.

   (b) r is deleted from $\delta_n(sw).ft$. There are two possibilities.

      i. $m_0 \oplus m(\mathsf{add}(r)) = 0$ and $m_0 \oplus m(\mathsf{del}(r)) > 0$. Then we have $m'_0 \oplus \mathsf{over}(m)(\mathsf{del}(r)) = \infty$. If $m'_0 \oplus \mathsf{over}(m)(\mathsf{add}(r)) = 0$, then we do not restrict p' because r must be deleted. If $m'_0 \oplus \mathsf{over}(m)(\mathsf{add}(r)) = \infty$, then we require p' fetch $\mathsf{add}(r)$ first and then $\mathsf{del}(r)$.

ii. $m_0 \oplus m(\mathsf{add}(r)) > 0$ and $m_0 \oplus m(\mathsf{del}(r)) > 0$. Then we have $m_0' \oplus \mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $m_0' \oplus \mathsf{over}(m)(\mathsf{del}(r)) = \infty$. We require p' fetch $\mathsf{add}(r)$ first and then $\mathsf{del}(r)$.

(c) $m_0 \oplus m(\mathsf{add}(r)) = 0$ and $m_0 \oplus m(\mathsf{del}(r)) = 0$. There are three possibilities.

i. $m_0' \oplus \mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $m_0' \oplus \mathsf{over}(m)(\mathsf{del}(r)) = \infty$. If $r \in \delta_n'(sw).ft$, then we require p' fetch $\mathsf{del}(r)$ first and then $\mathsf{add}(r)$. If $r \notin \delta_n'(sw).ft$, then we require p' fetch $\mathsf{add}(r)$ first and then $\mathsf{del}(r)$.

ii. $m_0' \oplus \mathsf{over}(m)(\mathsf{add}(r)) = \infty$ and $m_0' \oplus \mathsf{over}(m)(\mathsf{del}(r)) = 0$. Then we have $m_0'(\mathsf{add}(r)) = \infty$ and $m_0'(\mathsf{del}(r)) = 0$. By Lemma 16, $r \in \delta_n'(sw).ft$. Hence we do not restrict p' because r must be added to a flow table that already contains r.

iii. $m_0' \oplus \mathsf{over}(m)(\mathsf{add}(r)) = 0$ and $m_0' \oplus \mathsf{over}(m)(\mathsf{del}(r)) = \infty$. Then we have $m_0'(\mathsf{add}(r)) = 0$ and $m_0'(\mathsf{del}(r)) = \infty$. By Lemma 16, $r \notin \delta_n'(sw).ft$. Hence we do not restrict p' because r must be deleted from a flow table that does not contain r.

Intuitively, p' adds or deletes a rule in $\delta_n'(sw).ft$ if p adds or deletes the rule in $\delta_n(sw).ft$. For those control messages of the form $\mathsf{add}(r)$ and $\mathsf{del}(r)$ that are not in $m_0 \oplus m$ but in $m_0' \oplus \mathsf{over}(m)$, either p' adds r to $\delta_n'(sw).ft$ containing r already, or deletes r from $\delta_n'(sw).ft$ not containing r, or neutralizes $\mathsf{add}(r)$ and $\mathsf{del}(r)$ to keep $r \in \hat{ft}'$ iff $r \in \delta_n'(sw).ft$. Hence, $\hat{ft} = \hat{ft}'$. By Corollary 1, for any fetching order p_2 of $update(\hat{ft}, l_2@[b])$ that leads to $\delta_{n+1}(sw).ft$, there is a fetching order p_2' of $update^e(\hat{ft}', \mathsf{over}(l_2@[b]))$ that leads to $\delta_{n+1}'(sw).ft$ such that $\delta_{n+1}(sw).ft = \delta_{n+1}'(sw).ft$.

3. $l = [b]@l_2@[b, \hat{m}]$ or $[b]@l_2@[b]$. The proof is analogous to the case (2) where $m = \emptyset$.

Since we have proved all cases, the lemma holds. $\square$

Since TS_4 simulates TS_3 by Lemma 17, we have the following theorem.

Theorem 14. *For an SDN property $\square\phi$, if TS_4 satisfies $\square\phi$, TS_3 satisfies $\square\phi$.*

3.5.4 Proofs for All Packets In One Shot

We prove that TS_5 simulates TS_4. We define a relation $R \subseteq S_4 \times S_5$ such that $((\pi, \delta, cs, rq), (\pi', \delta', cs', rq')) \in R$ iff for all $pkt \in Packet$, for all $c \in Clients$, $\pi(c)(pkt) = \infty \rightarrow \pi'(c)(pkt) = \infty$ and for all $sw \in Switches$, $\delta(sw).pq(pkt) = \infty \rightarrow \delta'(sw).pq(pkt) = \infty$, $\delta(sw).cq = \delta'(sw).cq$, $\delta(sw).fq = \delta'(sw).fq$, $\delta(sw).ft = \delta'(sw).ft$, and $\delta(sw).wait = \delta'(sw).wait$, and $cs = cs'$, and $rq = rq'$.

Theorem 15. *(1) The relation R is a simulation relation. (2) For a safety property $\Box\phi$, if TS_5 satisfies $\Box\phi$, TS_4 satisfies $\Box\phi$.*

Proof. Proof of claim (1):

It is straightforward to verify that for all $(s,t) \in R$, if $s \xrightarrow{\alpha}_4 s'$, then there are t' and α' such that $t \xrightarrow{\alpha'}_5 t'$ and $(s', t') \in R$. In particular, $\alpha = \alpha'$ if α is not a match action. If α is a match action $match(sw, pkt, r)$ in A_4, then α' is a match action $match(sw, pkt\_lst, r\_lst)$ in A_5 such that pkt is in pkt_lst and r is in r_lst.

By claim (1), TS_5 simulates TS_4 and hence claim (2) holds accordingly. □

3.5.5 Proofs for Controller Optimization

Theorem 16. *Given an SDN $\mathcal{N}$ where the size of the request queue of the controller is one, and a safety property $\Box\phi$. (1) $TS_5 \triangleq TS_6$. (2) TS_5 satisfies $\Box\phi$ iff TS_6 satisfies $\Box\phi$.*

Proof. Proof of claim (1):

We first prove $TS_6 \trianglelefteq TS_5$, i.e., for each initial infinite execution $\rho = t_0 \xrightarrow{\alpha_1}_6 t_1 \xrightarrow{\alpha_2}_6 \ldots$ in TS_6, there is an initial infinite execution $\rho' = s_0 \xrightarrow{\beta_1}_5 s_1 \xrightarrow{\beta_2}_5 \ldots$ such that $\rho \triangleq \rho'$. We construct ρ' by scanning ρ from the beginning. For all $i \geq 0$, if $t_i \xrightarrow{nomatch\_ctrl(sw,pkt,cs)}_6 t_{i+1}$ in ρ, then we split $nomatch\_ctrl(sw, pkt, cs)$ into two actions $nomatch(sw, pkt)$ and $ctrl(pkt, cs)$ and introduce a new intermediate state u_i such that $s_i \xrightarrow{nomatch(sw,pkt)}_5 u_i \xrightarrow{ctrl(pkt,cs)}_5 s_{i+1}$ in ρ'. If $t_i \xrightarrow{\alpha}_6 t_{i+1}$ and α is not $nomatch\_ctrl(sw, pkt, cs)$, then define $s_i \xrightarrow{\alpha}_5 s_{i+1}$. The construction of ρ' ensures that for all $i \geq 0$, $s_i = t_i$. Moreover, if u_i is the successor of s_i then $L_5(s_i) = L_5(u_i)$ because $nomatch(sw, pkt)$ is a stutter action. Therefore, $\rho' \triangleq \rho$.

We then prove $TS_5 \trianglelefteq TS_6$. Let $\rho = s_0 \xrightarrow{\beta_1}_5 s_1 \xrightarrow{\beta_2}_5 \ldots$ be an initial infinite execution in TS_5. The construction of ρ' is the following. We walk through ρ until we find a

$nomatch(sw, pkt)$ action. If we cannot find it, then ρ is in TS_6 by definition. Otherwise, ρ is of the form $\rho = s_0 \xrightarrow{\beta_1}_5 s_1 \xrightarrow{\beta_2}_5 \dots \xrightarrow{\beta_{i-1}}_5 s_{i-1} \xrightarrow{nomatch(sw,pkt)}_5 s_i \xrightarrow{\beta_{i+1}}_5 s_{i+1} \dots$ where for all $1 \leq j \leq i-1$, β_j is not a no-match action. By definition of $\rightarrow_6$, we have an execution $\hat{\rho} = s_0 \xrightarrow{\beta_1}_6 s_1 \xrightarrow{\beta_2}_6 \dots \xrightarrow{\beta_{i-1}}_6 s_{i-1} \xrightarrow{nomatch(sw,pkt)}_5 s_i \xrightarrow{\beta_{i+1}}_5 s_{i+1} \dots$. We now consider two cases.

(1) Control action $ctrl(pkt, cs)$ does not appear in β_j for all $j > i$. Since we assume that the size of the request queue of the controller is 1, we know that for all $j > i$, $nomatch(sw, pkt)$ is independent of β_j. We can keep permuting $nomatch(sw, pkt)$ backward with $\beta_{i+1}, \beta_{i+2}, \dots$ and at the limit, we get an execution ρ'' in which $nomatch(sw, pkt)$ is never executed.

$$\rho'' = s_0 \xrightarrow{\beta_1}_6 s_1 \xrightarrow{\beta_2}_6 \dots \xrightarrow{\beta_{i-1}}_6 s_{i-1} \xrightarrow{\beta_{i+1}}_5 u_i \xrightarrow{\beta_{i+2}}_5 u_{i+1} \dots$$

Since the size of the request queue of the controller is one, for all $j > i$, β_j is not a no-match action and by the definition of $\rightarrow_6$, we have

$$\rho' = s_0 \xrightarrow{\beta_1}_6 s_1 \xrightarrow{\beta_2}_6 \dots \xrightarrow{\beta_{i-1}}_6 s_{i-1} \xrightarrow{\beta_{i+1}}_6 u_i \xrightarrow{\beta_{i+2}}_6 u_{i+1} \dots$$

Moreover, since $nomatch(sw, pkt)$ is a stutter action, we have $\rho' \triangleq \rho$.

(2) Control action $ctrl(pkt, cs)$ appears and the first one is β_k for some $k > i$. Hence $\hat{\rho}$ has the form: $\hat{\rho} = s_0 \xrightarrow{\beta_1}_6 \dots \xrightarrow{\beta_{i-1}}_6 s_{i-1} \xrightarrow{nomatch(sw,pkt)}_5 s_i \xrightarrow{\beta_{i+1}}_5 s_{i+1} \dots s_{k-1} \xrightarrow{\beta_k}_5 s_k \dots$.

Since the size of request queue is one, then $nomatch(sw, pkt)$ is independent of any actions β_j where $i < j < k$. We then permute $nomatch(sw, pkt)$ with β_j successively and end up with an execution as follows: $s_0 \xrightarrow{\beta_1}_6 \dots \xrightarrow{\beta_{i-1}}_6 s_{i-1} \xrightarrow{\beta_{i+1}}_5 u_i \xrightarrow{\beta_{i+2}}_5 \dots u_{k-2} \xrightarrow{nomatch(sw,pkt)}_5 s_{k-1} \xrightarrow{\beta_k}_5 s_k \dots$.

Since for all $i < j < k$, β_j is not a no-match action, by definition of $\rightarrow_6$, we have $\rho_1 = s_0 \xrightarrow{\beta_1}_6 \dots \xrightarrow{\beta_{i-1}}_6 s_{i-1} \xrightarrow{\beta_{i+1}}_6 u_i \xrightarrow{\beta_{i+2}}_6 \dots u_{k-2} \xrightarrow{nomatch\_ctrl(sw,pkt,cs)}_6 s_k \dots$.

The execution $s_0 \rightarrow_5^* s_k$ of ρ and the execution $s_0 \rightarrow_6^* s_k$ of ρ_1 are stutter equivalent because $nomatch(sw, pkt)$ is a stutter action.

Now the next task to make the rest execution of ρ_1 from s_k and the rest execution of ρ from s_k stutter equivalent. By repeating the reasoning in cases (1) and (2) from s_k on in ρ_1, at the limit, we end up with an execution ρ' in TS_6 such that $\rho' \triangleq \rho$.

Since claim (1) holds, claim (2) holds immediately. □

3.6 Related Work

There is a lot of systems and networking interest in SDNs [40, 62] and standards such as Openflow [91]. From the formal methods perspective, research has focused on verified programming language frameworks for writing SDN controllers [43, 55]. Here, verification refers to correct compilation from Frenetic to executable code, or to checking composability of programs, not the correctness of invariants.

Previous model checking attempts for SDNs mostly focused either on proving a static snapshot of the network [69] or on model checking or symbolic simulation techniques for a fixed number of packets [22, 98]. Recent work extended to controller updates and arbitrary number of packets [114], but used a manual process to add non-interference lemmas. In contrast, our technique automatically deals with unboundedly many packets and, thanks to the partial-order techniques, scales to much larger configurations than reported in [114]. Program verification for SDN controllers using loop invariants and SMT solving has been proposed recently [6]. While the invariants can quantify over the network (and therefore not limited to finite topologies), the model of the network ignores asynchronous interleavings of packet and control message processing that we handle here.

Our work builds on top of distributed enumerative model checking and the PReach tool [10]. Our contribution is identifying domain specific state space reduction heuristics that enable us to explore large configurations.

Chapter 4

Expand, Enlarge, and Check for Branching Vector Addition Systems

4.1 Introduction

Branching vector addition systems (BVAS) are an expressive model that generalize vector addition systems (VAS, or Petri nets) with branching structures. Intuitively, one can consider a VAS as producing a linear sequence of vectors using unary rewrite rules, where a rewrite rule takes a vector v and adds a constant δ to it, as long as the sum $v + \delta$ remains non-negative on all co-ordinates. A branching VAS adds a second, binary rewrite rule that takes two vectors v_1 and v_2 and rewrites them to $v_1 + v_2 + \delta$ for a constant δ, again provided the sum is non-negative on all co-ordinates. Thus, a BVAS generates a derivation tree of vectors, starting with a multiset of initial vectors, or axioms, at the leaves and generating a vector at the root of a derivation, where each internal node in the tree applies a unary or a binary rewrite rule. The reachability problem for BVAS is to check if a given vector can be derived, and the coverability problem asks, given a vector v, if a vector $v' \geq v$ can be derived. These generalize the corresponding problems for VAS. Several verification problems, such as the analysis of recursively parallel programs [12] and the analysis of some cryptographic protocols [120], have been shown to reduce to the coverability problem for BVAS.

Coverability for BVAS is known to be decidable, both through a generalized Karp-Miller construction [119] as well as through a bounding argument [35]. Further, the bounding argument characterizes the complexity of the problem: coverability is

2EXPTIME-complete [35] (contrast with the EXPSPACE-completeness for VAS [106]). The Karp-Miller construction is non-primitive recursive, since BVAS subsume VAS [68].

Despite potential applications, the study of BVAS has so far remained in the domain of theoretical results, and to the best of our knowledge, there have not been any attempts to build analysis tools for coverability. In contrast, tools for VAS coverability have made steady progress and can now handle quite large benchmarks derived from the analysis of multi-threaded programs [65, 70]. In our view, one reason is that a direct implementation of the algorithms from [35, 119] are unlikely to perform well: Karp-Miller trees for VAS do not perform well in practice, and Demri et al.'s complexity-theoretically optimal algorithm performs a non-deterministic guess and enumeration by an alternating Turing machine.

In this dissertation, we apply the *expand, enlarge, and check* paradigm (EEC) [47] to the analysis of BVAS. EEC is a successful heuristic for checking coverability of well-structured transition systems such as Petri nets. It constructs a sequence of under- and over-approximations of the state space of a system such that, for a target state t, (1) if t is coverable, then a witness is found by an under-approximation, (2) if t is not coverable, then a witness for un-coverability is found by an over-approximation, and (3) eventually, one of the two outcomes occur and the algorithm terminates.

EEC offers several nice features for implementation. First, each approximation it considers is finite-state, thus opening the possibility of applying model checkers for finite-state systems. Second, EEC is goal-directed: it computes abstractions that are precise enough to prove or disprove coverability of a target, unlike a Karp-Miller procedure that computes the exact coverability set independent of the target. Third, it allows a forward abstract exploration of the state space, which is often more effective in practice.

Our first contribution is to port the EEC paradigm to the coverability analysis of BVAS. We show how to construct a sequence of under- and over-approximations of derivations such that if a target is coverable, an under-approximation derives a witness for coverability, and if a target is not coverable, an over-approximation derives a witness for un-coverability. We generalize the proof of correctness of EEC for well-structured systems. Since there is no BVAS analogue of a backward-reachability algorithm for VAS, our proofs instead use induction on derivations and the Karp-Miller construction of [119].

A natural question is how well EEC performs in the worst case compared to asymptotically optimal algorithms. For example, even for VAS, it is unknown if the EEC algorithm can match the known EXPSPACE upper bound for coverability, or if it matches the non-primitive recursive lower bound for Karp-Miller trees. Our second contribution is to bound the number of iterations of the EEC algorithm in the worst case. We show that we can compute a constant c of size doubly exponential in the size of the BVAS and the target vector such that the EEC algorithm is guaranteed to terminate in c iterations. In each iteration, the algorithm explores approximate state spaces of derivations, that correspond to exploring AND-OR trees of size doubly exponential in the input. In other words, if each exploration is performed optimally, we get an optimal asymptotic upper bound for EEC. Specifically, for VAS, we get an EXPSPACE upper bound, since there are doubly exponential iterations and each iteration checks two reachability problems over doubly-exponential state spaces. (In practice though, model checkers do not implement space-optimal reachability procedures.) While our proof uses Rackoff-style bounds [35, 106], our implementation does not require any knowledge of these bounds. A similar argument was used in [15] to show a doubly exponential bound on the backward reachability algorithm for VAS.

We have implemented the EEC-based procedure for BVAS coverability. Our motivation for analyzing BVAS came from the analysis of recursively parallel programs [12, 45]. It is known that the analysis of asynchronous programs, a co-operatively scheduled concurrency model, can be reduced to coverability of VAS [45], and there have been EEC-based tools for these programs [64]. However, some asynchronous programs use features such as posting a set of tasks in a handler and waiting on the first task to return, that are not reducible to asynchronous programs. Bouajjani and Emmi [12] define a class of recursively parallel programs that can express such constructs, and show that the safety verification problem for this class is equivalent to coverability of BVAS. We applied this reduction in our implementation, and used our tool to model check safety properties of recursively parallel programs. We coded the control flow of tasks in a simple web server [34] and showed that our tool can successfully check for safety properties and find bugs. On our examples, the EEC algorithm terminates in one iteration, that is, with a $\{0, 1, \infty\}$ abstraction. While our evaluations are preliminary, we believe there is a potential for model checking tools for complex concurrent programs based on BVAS coverability.

4.2 Preliminaries

Well quasi ordering. A *quasi ordering* $(X, \preceq)$ is a reflexive and transitive binary relation on X. A quasi ordering $(X, \preceq)$ is a *well quasi ordering* iff for every infinite sequence $x_0, x_1, \ldots$ of elements from X, there exists $i < j$ with $x_i \preceq x_j$. A subset X' of X is *upward closed* if for each $x \in X$, if there is an $x' \in X'$ with $x' \preceq x$ then $x \in X'$. A subset X' of X is *downward closed* if for each $x \in X$, if there is an $x' \in X'$ with $x \preceq x'$ then $x \in X'$. Given $x \in X$, we write $x\!\downarrow$ and $x\!\uparrow$ for the *downward closure* $\{x' \in X \mid x' \preceq x\}$ and *upward closure* $\{x' \in X \mid x \preceq x'\}$ of x respectively. Downward and upward closures are naturally extended to sets, i.e., $X\!\downarrow = \bigcup_{x \in X} x\!\downarrow$ and $X\!\uparrow = \bigcup_{x \in X} x\!\uparrow$. A subset $S \subseteq X$ is *minimal* iff for every two elements $x, x' \in S$, we have $x \not\preceq x'$.

Numbers and vectors. We write $\mathbb{N}$, $\mathbb{N}^+$ and $\mathbb{Z}$ for the set of non-negative, positive and arbitrary integers, respectively. Given two integers a and b, we write $[a, b]$ for $\{n \in \mathbb{Z} \mid a \leq n \leq b\}$.

For a vector $v \in \mathbb{Z}^k$ and $i \in [1, k]$, we write $v[i]$ for the ith component of v. Given two vectors $v, v' \in \mathbb{Z}^k$, $v \leq v'$ iff for all $i \in [1, k]$, $v[i] \leq v'[i]$. Moreover, $v < v'$ iff $v \leq v'$ and $v' \not\leq v$. It is well-known that $(\mathbb{N}^k, \leq)$ is a well quasi ordering. We write $\mathbf{0}$ for the zero vector.

Given a finite set $S \subseteq \mathbb{Z}$ of integers, we write $\max(S)$ for the greatest integer in the set. We define $\max(\emptyset) = 0$. Given a vector $v \in \mathbb{Z}^k$, let $\max(v) = \max(\{v[1], \ldots, v[k]\})$. When $k = 0$, we have $\max(()) = 0$. We define $\min(S)$ analogously. We write $\min(0, v)$ for the vector $(\min(\{0, v[1]\}), \ldots, \min(\{0, v[k]\}))$. The vector $\max(0, v)$ is defined analogously. For simplicity, we write v^- for the vector $-\min(0, v)$ and v^+ for the vector $\max(0, v)$. Given a finite set of vectors $R \subseteq \mathbb{Z}^k$, let $R^{-/+}$ be the set $\{v^{-/+} \mid v \in R\}$

respectively. We define $\max(R) = \max(\{\max(v^+) \mid v \in R\})$. The size of a vector is the number of bits required to encode it, all numbers being encoded in binary.

Trees. A *finite binary tree* $\mathcal{T}$, which may contain nodes with one child, is a non-empty finite subset of $\{1,2\}^*$ such that, for all $n \in \{1,2\}^*$ and $i \in \{1,2\}$, $n \cdot 2 \in \mathcal{T}$ implies $n \cdot 1 \in \mathcal{T}$, and $n \cdot i \in \mathcal{T}$ implies $n \in \mathcal{T}$. The nodes of $\mathcal{T}$ are its elements. The root of $\mathcal{T}$ is ε, the empty word. All notions such as parent, child, subtree and leaf, have their standard meanings. The height of $\mathcal{T}$ is the number of nodes in the longest path from the root to a leaf.

BVAS, derivations, and coverability. A *branching vector addition system* (BVAS) [35, 119] is a tuple $\mathcal{B} = (k, A, R_1, R_2)$, where $k \in \mathbb{N}$ is the *dimension*, $A \subseteq \mathbb{N}^k$ is a non-empty finite set of *axioms*, and $R_1, R_2 \subseteq \mathbb{Z}^k$ are finite sets of *unary and binary rules*, respectively. The size of a BVAS, $size(\mathcal{B})$ is the number of bits required to encode a BVAS, where numbers are encoded in binary.

The semantics of a BVAS $\mathcal{B}$ is captured using derivations. Intuitively, a derivation starts with a number of axioms from A, proceeds by applying rules from $R_1 \cup R_2$, and ends with a single vector. Applying a unary rule means adding it to a derived vector, and applying a binary rule means adding it to the sum of two derived vectors. While applying rules, all derived vectors are required to be non-negative. Formally, a *derivation* $\mathcal{D}$ of $\mathcal{B}$ is defined inductively as follows.

($D1$) If $v \in A$, then $\overline{v}$ is a derivation.

($D2$) If $\mathcal{D}_1$ is a derivation with a derived vector $v_1 \in \mathbb{N}^k$, then for each unary rule $\delta_1 \in R_1$ with $\mathbf{0} \leq v_1 + \delta_1$,

$$\mathcal{D} : \dfrac{\begin{matrix} \vdots\, \mathcal{D}_1 \\ v_1 \end{matrix}}{v}\; \delta_1$$

is a derivation, where $v = v_1 + \delta_1$.

($D3$) If $\mathcal{D}_1$ and $\mathcal{D}_2$ are derivations with derived vectors $v_1, v_2 \in \mathbb{N}^k$ respectively, then for each binary rule $\delta_2 \in R_2$ with $\mathbf{0} \leq v_1 + v_2 + \delta_2$,

$$\mathcal{D} : \dfrac{\begin{matrix} \vdots\, \mathcal{D}_1 \\ v_1 \end{matrix} \qquad \begin{matrix} \vdots\, \mathcal{D}_2 \\ v_2 \end{matrix}}{v}\; \delta_2$$

is a derivation, where $v = v_1 + v_2 + \delta_2$.

A derivation $\mathcal{D}$ can be represented as a finite binary tree whose nodes are labelled by non-negative vectors. Therefore, all notions of trees can be naturally applied to derivations. For a derivation $\mathcal{D}$ and its node n, we write $\mathcal{D}(n)$ for the non-negative vector labelled at n. We say $\mathcal{D}$ *derives a vector* v iff $\mathcal{D}(\varepsilon) = v$.

A derivation $\mathcal{D}$ is *compact* iff for each node n and for each its ancestor n', we have $\mathcal{D}(n) \neq \mathcal{D}(n')$. Given a derivation $\mathcal{D}$ with a node n and an ancestor n' of n with $\mathcal{D}(n) = \mathcal{D}(n')$, a *contraction* $\mathcal{D}[n' \leftarrow n]$ over $\mathcal{D}$ is obtained by replacing the subtree rooted at n' with the subtree rooted at n in $\mathcal{D}$. We write $\mathsf{compact}(\mathcal{D})$ for the compact derivation computed by a finite sequence of contractions over $\mathcal{D}$.

Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$, we say a vector v is *reachable* in $\mathcal{B}$ iff there is a derivation $\mathcal{D}$ with $\mathcal{D}(\varepsilon) = v$. We write $\mathit{Reach}(\mathcal{B}) = \{v \mid \exists \mathcal{D}.\ \mathcal{D}(\varepsilon) = v\}$ for the set of reachable vectors in $\mathcal{B}$. We say a vector v is *coverable* in $\mathcal{B}$ iff there is a derivation $\mathcal{D}$ with $v \leq \mathcal{D}(\varepsilon)$. We call a derivation $\mathcal{D}$ a *covering witness* of v iff $v \leq \mathcal{D}(\varepsilon)$. The *coverability problem* asks, given a BVAS $\mathcal{B}$ and a vector $t \in \mathbb{N}^k$, whether t is coverable in $\mathcal{B}$. Equivalently, t is coverable iff $t \in \mathit{Reach}(\mathcal{B})\downarrow$.

4.3 Under- and Over-approximation

We give two approximate analyses for BVAS: an under-approximation that fixes a finite set of vectors and only considers those vectors in that finite set, and an over-approximation that introduces limit elements. The under-approximation can show that a vector is coverable and the over-approximation can prove that a vector is not coverable.

4.3.1 Underapproximation

Truncated Derivations Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and an $i \in \mathbb{N}$, define $C_i \subseteq \mathbb{N}^k$ as $A \cup \{0, \ldots, i\}^k$. Given a vector $v \in \mathbb{N}^k$ and an $i \in \mathbb{N}$, We write $\mathsf{under}(v, i)$ for a *truncated vector* such that for all $j \in [1, k]$, $\mathsf{under}(v, i)[j] = v[j]$ if $v[j] \leq i$, $\mathsf{under}(v, i)[j] = i$ otherwise. For all vector $v \in \mathbb{N}^k$ and for all $i \in \mathbb{N}$, $\mathsf{under}(v, i) \leq v$. A *truncated derivation* $\mathcal{F}$ w.r.t. i is defined inductively as follows.

$(T1)$ If $v \in A$, then $\overline{v}$ is a truncated derivation.

$(T2)$ If $\mathcal{F}_1$ is a truncated derivation with a derived truncated vector $v_1 \in \mathbb{N}^k$, then for each unary rule $\delta_1 \in R_1$ with $\mathbf{0} \leq v_1 + \delta_1$,

$$\mathcal{F} : \frac{\overset{\vdots\, \mathcal{F}_1}{v_1}}{v}\, \delta_1$$

is a truncated derivation, where $v = \mathsf{under}(v_1 + \delta_1, i)$.

$(T3)$ If $\mathcal{F}_1$ and $\mathcal{F}_2$ are truncated derivations with derived truncated vectors $v_1, v_2 \in \mathbb{N}^k$ respectively, then for each binary rule $\delta_2 \in R_2$ with $\mathbf{0} \leq v_1 + v_2 + \delta_2$,

$$\mathcal{F} : \frac{\overset{\vdots\, \mathcal{F}_1}{v_1} \qquad \overset{\vdots\, \mathcal{F}_2}{v_2}}{v}\, \delta_2$$

is a truncated derivation, where $v = \mathsf{under}(v_1 + v_2 + \delta_2, i)$.

Analogously to derivations, a truncated derivation $\mathcal{F}$ is a finite binary tree whose nodes are labelled by truncated vectors. We say $\mathcal{F}$ *derives a truncated vector* v iff $\mathcal{F}(\varepsilon) = v$. We naturally extend the notions of *compactness*, *covering witness*, and *coverability* to truncated derivations w.r.t. $\leq$.

Lemma 18. *Let $\mathcal{B} = (k, A, R_1, R_2)$ be a BVAS and $i \in \mathbb{N}$. For any $h \in \mathbb{N}^+$, there are finitely many truncated derivations of a BVAS of height h.*

Given a BVAS $\mathcal{B}$, we define a total ordering $\sqsubseteq$ on truncated derivations according to their heights as follows. Since for each $h \in \mathbb{N}^+$ there are only finitely many, say k_h, truncated derivations of height h, we can enumerate them without repetition, arbitrarily as $\mathcal{F}_{h1}, \ldots, \mathcal{F}_{hk_h}$. We define $\mathcal{F}_{mi} \sqsubseteq \mathcal{F}_{nj}$ iff $m < n$, or $m = n$ and $i \leq j$.

The Forest $\mathsf{Under}(\mathcal{B}, C_i)$ Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and $i \in \mathbb{N}$, we construct a forest $\mathsf{Under}(\mathcal{B}, C_i)$ whose nodes are compact truncated derivations by the following rules:

$(U1)$ For each axiom $v \in A$, the truncated derivation $\overline{v}$ is a root.

($U2$) Let $\mathcal{F}_1$ be a compact truncated derivation in the forest. Let $\mathcal{F}$ be a truncated derivation obtained by applying a unary rule $\delta_1 \in R_1$ to $\mathcal{F}_1$ (as in rule T2). If $\mathsf{compact}(\mathcal{F})$ has not been added to the forest then add $\mathsf{compact}(\mathcal{F})$ as a child of $\mathcal{F}_1$ in the forest.

($U3$) Suppose compact truncated derivations $\mathcal{F}_1, \mathcal{F}_2$ are in the forest. Let $\mathcal{F}$ be a truncated derivation obtained by applying a binary rule $\delta_2 \in R_2$ to $\mathcal{F}_1$ and $\mathcal{F}_2$ (as in rule T3). If $\mathsf{compact}(\mathcal{F})$ has not been added to the forest then we add $\mathsf{compact}(\mathcal{F})$ to the forest as a child of $\mathcal{F}'$ where $\mathcal{F}'$ is the greater one between $\mathcal{F}_1$ and $\mathcal{F}_2$ w.r.t. the total order $\sqsubseteq$.

The following lemma shows that the construction of $\mathsf{Under}(\mathcal{B}, C_i)$ eventually terminates, and that it can be used to prove coverability.

Theorem 17 (Underapproximation)**.** *Let $\mathcal{B}$ be a BVAS.*

1. *For any $i \in \mathbb{N}$, the forest $\mathsf{Under}(\mathcal{B}, C_i)$ is finite.*

2. *Given an $i \in \mathbb{N}$, for any truncated derivation $\mathcal{F}$, there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $\mathcal{F}(\varepsilon) \leq \mathcal{D}(\varepsilon)$.*

3. *For any vector $v \in \mathbb{N}^k$, we have $v \in Reach(\mathcal{B})\downarrow$ iff there exists $i \in \mathbb{N}$ such that there is a truncated derivation $\mathcal{F}$ in $\mathsf{Under}(\mathcal{B}, C_i)$ with $v \leq \mathcal{F}(\varepsilon)$.*

Proof. Part (1). Fix i. It is easy to see that there are finitely many trees in the forest and each tree is finitely branching, since there are at most finitely many trees of a given height. If the forest is not finite, then by König's lemma, there is an infinite simple path of compact truncated derivations $\mathcal{F}_1, \mathcal{F}_2, \ldots$ in the forest such that for every $i \geq 1$, $\mathcal{F}_i$ is a sub-compact truncated derivation of $\mathcal{F}_{i+1}$. This induces an infinite sequence of truncated vectors $\mathcal{F}_1(\varepsilon), \mathcal{F}_2(\varepsilon) \ldots$ such that for every $i \neq j$, $\mathcal{F}_i(\varepsilon) \neq \mathcal{F}_j(\varepsilon)$. However, since for all $\mathcal{F}$ in the forest, $\mathcal{F}(\varepsilon) \in C_i$ and C_i is finite, such infinite sequence of truncated vectors does not exist.

Part (2). By induction on the height of $\mathcal{F}$.

Part (3). $\Rightarrow$: Since $Reach(\mathcal{B}) \cap v\uparrow \neq \emptyset$, there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $v \leq \mathcal{D}(\varepsilon)$. Let S be the union of the set of axioms A and the set of all vectors in $\mathsf{compact}(\mathcal{D})$.

Because both sets are finite, let i be $\max(S)$. Then $\mathsf{compact}(\mathcal{D})$ is in $\mathsf{Under}(\mathcal{B}, C_i)$ and $v \leq \mathcal{D}(\varepsilon) = \mathsf{compact}(\mathcal{D})(\varepsilon)$.

$\Leftarrow$: By Part (2), there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $\mathcal{F}(\varepsilon) \leq \mathcal{D}(\varepsilon)$. Since $\mathcal{D}(\varepsilon) \in \mathit{Reach}(\mathcal{B})$ and $v \leq \mathcal{F}(\varepsilon)$, $v \in \mathit{Reach}(\mathcal{B})\downarrow$. $\square$

4.3.2 Overapproximation

To define over-approximation of derivations, we introduce *extended derivations* which consider vectors over $\mathbb{N} \cup \{\infty\}$. We then present an algorithm that builds a forest over-approximating the downward closure of reachable vectors of a given BVAS and prove termination and correctness.

Let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ be the extension of the natural numbers with infinity. An *extended vector* is an element of $\mathbb{N}_\infty^k$. For extended vectors $u, u' \in \mathbb{N}_\infty^k$, we write $u \leq^\infty u'$ iff for all $i \in [1,k]$, we have $u[i] \leq u'[i]$ or $u'[i] = \infty$. We write $u <^\infty u'$ iff $u \leq^\infty u'$ and $u' \not\leq^\infty u$. We always use words starting with the letter u to denote an extended vector (e.g. u, u', u_1 etc.) and words starting with the letter v to denote a vector in $\mathbb{Z}^k$ (e.g. v, v', v_1 etc.). Extended vectors describe sets of vectors: we define $\gamma : \mathbb{N}_\infty^k \to 2^{\mathbb{N}^k}$ as $\gamma(u) = \{v \in \mathbb{N}^k \mid v \leq^\infty u\}$, and naturally extend γ to sets of extended vectors.

Proposition 2. *[47] (1) Given an extended vector $u \in \mathbb{N}_\infty^k$ and a finite set of extended vectors $S \subseteq \mathbb{N}_\infty^k$, $\gamma(u) \subseteq \gamma(S)$ iff there is $u' \in S$ such that $u \leq^\infty u'$. (2) Given two finite and minimal sets $S_1, S_2 \subseteq \mathbb{N}_\infty^k$, $S_1 = S_2$ if and only if $\gamma(S_1) = \gamma(S_2)$.*

Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$, there exists a finite and minimal subset $\mathsf{CS}(\mathcal{B}) \subseteq \mathbb{N}_\infty^k$ such that $\gamma(\mathsf{CS}(\mathcal{B})) = \mathit{Reach}(\mathcal{B})\downarrow$. We shall call $\mathsf{CS}(\mathcal{B})$ the *finite representation* of $\mathit{Reach}(\mathcal{B})\downarrow$.

Extended Derivations Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and an $i \in \mathbb{N}$, let $C_i = \{0, \ldots, i\}^k \cup A$ and $L_i = \{0, \ldots, i, \infty\}^k \setminus \{0, \ldots, i\}^k$. Given two sets $S_1 \subseteq \mathbb{N}^k$ and $S_2 \subseteq \mathbb{N}_\infty^k$, we say that S_2 is an *overapproximation* of S_1 iff $S_1 \subseteq \gamma(S_2)$. Moreover, we say that S_2 is *the most precise overapproximation* of S_1 in $L_i \cup C_i$ iff there is no finite and minimal subset $S \subseteq L_i \cup C_i$ such that $S_1 \subseteq \gamma(S) \subset \gamma(S_2)$. In the following, in case S_2 is a singleton set $\{u\}$, we write that u is (the most precise) overapproximation of S_1 for simplicity.

Given an extended vector $u \in \mathbb{N}_\infty^k$ and an $i \in \mathbb{N}$, We write $\mathsf{over}(u,i)$ for the extended vector such that for all $j \in [1,k]$, $\mathsf{over}(u,i)[j] = u[j]$ if $u[j] \leq i$, $\mathsf{over}(u,i)[j] = \infty$ otherwise. Note that $\mathsf{over}(u,i)$ is an overapproximation of $\gamma(u)$, and interestingly, is the most precise overapproximation of $\gamma(u)$ in $L_i \cup C_i$ [47].

We can naturally extend the addition of vectors to the addition of extended vectors by assuming that $\infty + \infty = \infty$ and $\infty + c = \infty$ for all $c \in \mathbb{Z}$.

Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and $i \in \mathbb{N}$, an *extended derivation* $\mathcal{E}$ is defined inductively as follows.

$(E1)$ If $v \in A$, then $\overline{v}$ is an extended derivation.

$(E2)$ If $\mathcal{E}_1$ is an extended derivation with a derived extended vector $u_1 \in \mathbb{N}_\infty^k$, then for each unary rule $\delta_1 \in R_1$ with $\mathbf{0} \leq^\infty u_1 + \delta_1$,

$$\mathcal{E} : \dfrac{\stackrel{\vdots\, \mathcal{E}_1}{u_1}}{u}\, \delta_1$$

is an extended derivation, where $u = \mathsf{over}(u_1 + \delta_1, i)$.

$(E3)$ If $\mathcal{E}_1$ and $\mathcal{E}_2$ are extended derivations with derived extended vectors $u_1, u_2 \in \mathbb{N}_\infty^k$ respectively, then for each binary rule $\delta_2 \in R_2$ with $\mathbf{0} \leq^\infty u_1 + u_2 + \delta_2$,

$$\mathcal{E} : \dfrac{\stackrel{\vdots\, \mathcal{E}_1}{u_1} \qquad \stackrel{\vdots\, \mathcal{E}_2}{u_2}}{u}\, \delta_2$$

is an extended derivation, where $u = \mathsf{over}(u_1 + u_2 + \delta_2, i)$.

Analogously to derivations, an extended derivation $\mathcal{E}$ is a finite binary tree whose nodes are labelled by extended vectors. For an extended derivation $\mathcal{E}$ and its node n, we write $\mathcal{E}(n)$ for the extended vector labelled at n. We say $\mathcal{E}$ *derives an extended vector* u iff $\mathcal{E}(\varepsilon) = u$. We naturally extend the notions of *compactness*, *covering witness*, and *coverability* to extended derivations w.r.t. $\leq^\infty$. Similar to derivations, the following lemma shows that there are finitely many extended derivations of a given height.

Lemma 19. *Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and $i \in \mathbb{N}$, for each $h \in \mathbb{N}^+$, there are finitely many extended derivations of height h.*

Given a BVAS $\mathcal{B}$, we define a total ordering $\sqsubseteq_e$ on extended derivations according to their heights. Since for each $h \in \mathbb{N}^+$ there are only finitely many, say k_h, extended derivations of height h, we can enumerate them without repetition, arbitrarily as $\mathcal{E}_{h1}, \ldots, \mathcal{E}_{hk_h}$. We define $\mathcal{E}_{mi} \sqsubseteq_e \mathcal{E}_{nj}$ iff $m < n$, or $m = n$ and $i \leq j$.

The Forest $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and an $i \in \mathbb{N}$, we construct a forest $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ whose nodes are compact extended derivations by following the rules below.

$(O1)$ For each axiom $v \in A$, the extended derivation $\overline{v}$ is a root.

$(O2)$ If a compact extended derivation $\mathcal{E}_1$ is already in the forest and $\mathsf{compact}(\mathcal{E})$ has not been added in the forest where $\mathcal{E}$ is computed by applying a unary rule to $\mathcal{E}_1$ as in Rule E2, then add $\mathsf{compact}(\mathcal{E})$ as a child of $\mathcal{E}_1$ in the forest.

$(O3)$ If compact extended derivations $\mathcal{E}_1, \mathcal{E}_2$ are already in the forest and $\mathsf{compact}(\mathcal{E})$ has not been added in the forest where $\mathcal{E}$ is computed by applying a binary rule to $\mathcal{E}_1$ and $\mathcal{E}_2$ as in Rule E3, then we add $\mathsf{compact}(\mathcal{E})$ to the forest as a child of $\mathcal{E}'$ where $\mathcal{E}'$ is the greater one between $\mathcal{E}_1$ and $\mathcal{E}_2$ w.r.t. the total order $\sqsubseteq_e$.

Theorem 18 (Overapproximation)**.** *Let $\mathcal{B}$ be a BVAS.*

1. *For each $i \in \mathbb{N}$, the forest $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ is finite.*

2. *Given $i \in \mathbb{N}$, for any derivation $\mathcal{D}$, there is a compact extended derivation $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ with $\mathcal{D}(\varepsilon) \leq^\infty \mathcal{E}(\varepsilon)$.*

3. *For $v \in \mathbb{N}^k$, $Reach(\mathcal{B}) \cap v\uparrow = \emptyset$ iff there exists an $i \in \mathbb{N}$ such that for any compact extended derivation $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$, we have $\gamma(\mathcal{E}(\varepsilon)) \cap v\uparrow = \emptyset$.*

Proof. The proof of Part (1) is similar to the proof of Theorem 17(1), because $L_i \cup C_i$ is finite.

The proof of Part (2) is by induction on the height of $\mathcal{D}$.

Part (3). $\Leftarrow$: Suppose $Reach(\mathcal{B}) \cap v\uparrow \neq \emptyset$. Then there is a derivation $\mathcal{D}$ in $\mathcal{B}$ such that $\mathcal{D}(\varepsilon) \in v\uparrow$. Using Part (2), we can find $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ such that $\mathcal{D}(\varepsilon) \leq^\infty \mathcal{E}(\varepsilon)$. For $\mathcal{E}$, we have $\mathcal{D}(\varepsilon) \in \gamma(\mathcal{E}(\varepsilon))$ and thus $\gamma(\mathcal{E}(\varepsilon)) \cap v\uparrow \neq \emptyset$.

Algorithm 1: EEC Algorithm to decide the coverability problem of BVAS.

```
Input: A BVAS B = (k, A, R_1, R_2) and a vector t ∈ N^k.
Output: "Cover" if t is coverable in B, "Uncover" otherwise.
begin
    i ⟵ 0
    while true do
        Compute Under(B, C_i)                      // Expand
        Compute Over(B, L_i, C_i)                  // Enlarge
        // Check
        if ∃F ∈ Under(B, C_i). t ≤ F(ε) then
            return "Cover"
        else if ∀E ∈ Over(B, L_i, C_i). t ≰^∞ E(ε) then
            return "Uncover"
        i ⟵ i + 1
```

$\Rightarrow$: Since $Reach(\mathcal{B}) \cap v\uparrow = \emptyset$ iff $Reach(\mathcal{B})\downarrow \cap v\uparrow = \emptyset$, $\gamma(\mathsf{CS}(\mathcal{B})) \cap v\uparrow = \emptyset$. Take $i \in \mathbb{N}$ such that $\mathsf{CS}(\mathcal{B}) \subseteq L_i \cup C_i$. For every extended derivation $\mathcal{E}$ in $\mathcal{B}$, we have $\gamma(\mathcal{E}(\varepsilon)) \subseteq \gamma(\mathsf{CS}(\mathcal{B}))$. This can be proved by induction on the height of $\mathcal{E}$.

For every compact extended derivation $\mathcal{E}$ in $\mathsf{Over}(\mathcal{B}, L_i, C_i)$, we therefore have that $\gamma(\mathcal{E}(\varepsilon)) \subseteq \gamma(\mathsf{CS}(\mathcal{B}))$. Hence $\gamma(\mathcal{E}(\varepsilon)) \cap v\uparrow = \emptyset$. □

4.3.3 EEC Algorithm

Algorithm 1 shows the schematic of the EEC algorithm. It takes as input a BVAS $\mathcal{B}$ and a target vector t. It uses an abstraction parameter i, initially 0, and defines the family of abstractions C_i and L_i. It iteratively computes the under-approximation Under and over-approximation Over w.r.t. i. If the under-approximation covers t, it returns "Cover"; if the over-approximation shows t cannot be covered, it returns "Uncover." Otherwise, it increments i and loops again. From Theorems 17 and 18, we conclude that this algorithm eventually terminates with the correct result.

We briefly remark on two optimizations. First, instead of explicitly keeping forests of derivations in Over and Under, we can only maintain the vectors that label the roots of the derivations. The structure of the forest was required to prove termination in [119], but can be reconstructed using only the vectors and the timestamps at which the vectors were added. Second, in Under (resp. Over), we can only keep maximal vectors (resp. extended vectors): if two vectors $v_1 \leq v_2$ (resp. extended vectors $u_1 \leq^\infty u_2$), we can omit v_1 (resp. u_1) and only keep v_2 (resp. u_2). Indeed, if $t \leq v_1$ in Under, we also have $t \leq v_2$, and so the cover check succeeds in the EEC algorithm. Further, if $t \not\leq^\infty u_2$ in Over, we have $t \not\leq^\infty u_1$, and so the uncover check succeeds as well. We thank Sylvain Schmitz for these observations.

4.4 Complexity Analysis

We now give an upper bound on the number of iterations of the EEC algorithm. Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and a derivation $\mathcal{D}$, for each internal node n, we write $\delta(n) \in \mathbb{Z}^k$ for the rule $\delta \in R_1 \cup R_2$ that is applied to derive $\mathcal{D}(n)$. We extend this notation to truncated and extended derivations as well. Given a derivation $\mathcal{D}$ and an $i \in \mathbb{N}^k$, we define a truncated derivation $\mathsf{under}(\mathcal{D}, i)$ inductively as follows:

1. If n is a leaf, then $\mathsf{under}(\mathcal{D}, i)(n) = \mathcal{D}(n)$.
2. If n has a child n' and $\mathcal{D}(n) = \mathcal{D}(n') + \delta(n)$, then $\mathsf{under}(\mathcal{D}, i)(n) = \mathsf{under}(\mathsf{under}(\mathcal{D}, i)(n') + \delta(n), i)$.
3. If n has two children n', n'' and $\mathcal{D}(n) = \mathcal{D}(n') + \mathcal{D}(n'') + \delta(n)$, then $\mathsf{under}(\mathcal{D}, i)(n) = \mathsf{under}(\mathsf{under}(\mathcal{D}, i)(n') + \mathsf{under}(\mathcal{D}, i)(n'') + \delta(n), i)$.

We can also define an extended derivation $\mathsf{over}(\mathcal{D}, i)$ inductively by following the above rules except that we replace all $\mathsf{under}(\Box, i)$ by $\mathsf{over}(\Box, i)$.

We start with some intuition in the special case of vector addition systems. A *vector addition system* (VAS) $\mathcal{V}$ is a BVAS $(k, \{a\}, R, \emptyset)$. For simplicity, we write a VAS as just (k, a, R). Note that a derivation $\mathcal{D}$ of a VAS $\mathcal{V}$ is degenerated to a sequence of non-negative vectors. In the following, we say the *length* of $\mathcal{D}$ instead of the height of $\mathcal{D}$ for convenience in the VAS context. For VAS, Rackoff [106] proved the coverability problem is EXPSPACE-complete by showing that if a covering witness (derivation) exists, then there must exist one whose length h is at most doubly exponential in the size of the VAS $\mathcal{V}$ and the target vector t. Further, there is a derivation of length at most h in which the maximum constant is bounded by $i := h \cdot \mathit{size}(\mathcal{V}) + \max(t)$. This is because in h steps, a vector can decrease at most $h \cdot \mathit{size}(\mathcal{V})$, so if any co-ordinate goes over i, it remains higher than $\max(t)$ after executing the path. By the same argument, if there is an extended derivation of length at most h and constant i covering t, then we can find a derivation for t.

If t is coverable, using the above argument and Theorem 17, we see that $\mathsf{Under}(\mathcal{V}, C_i)$ will contain a covering witness of t. If t is not coverable, then the above argument shows that all extended derivations of $\mathsf{Over}(\mathcal{V}, L_i, C_i)$ of length at most h will not cover t. However, there may be longer extended derivations in $\mathsf{Over}(\mathcal{V}, L_i, C_i)$. For these, we can show that $\mathsf{Over}(\mathcal{V}, L_i, C_i)$ also contains a contraction of that extended derivation of length at most h. In both cases, EEC terminates in i iterations, which is doubly exponential in the size of the input.

We now show the bound for BVAS. The following lemma is the key observation in the optimal algorithm of [35].

Lemma 20. *[35] Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and a vector $t \in \mathbb{N}^k$, if t is coverable in $\mathcal{B}$, then there is a covering witness (derivation) $\mathcal{D}$ whose height is at most* $(\max((R_1 \cup R_2)^-) + \max(t) + 2)^{(3k)!}$.

Moreover, the following lemma shows that the maximum constant appearing in a height-bounded derivation can remain polynomial in the height.

Lemma 21. *Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$, a vector $t \in \mathbb{N}^k$ and a derivation $\mathcal{D}$ whose height is at most h, for any bound $i \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, $\mathcal{D}$ is a covering witness of t iff $\mathsf{under}(\mathcal{D}, i)$ is a covering witness of t.*

Proof. Fix an i such that $i \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$.

$\Leftarrow$: It holds by Theorem 17.

$\Rightarrow$: Given a derivation $\mathcal{D}$, we say that an index j is *marked* iff during the construction of $\mathsf{under}(\mathcal{D}, i)$, there is a vector v, which is computed after applying a rule and before comparing to i, such that $v[j] > i$.

Given a derivation $\mathcal{D}$, during the construction of $\mathsf{under}(\mathcal{D}, i)$, for each index $j \in [1, k]$, we check the following: If j is marked, then there is a node n such that $\mathsf{under}(\mathcal{D}, i)(n)[j] = i$. Since $\mathit{height}(\mathsf{under}(\mathcal{D}, i)) = \mathit{height}(\mathcal{D}) \leq h$, we know that the length of the path from n to the root ε is at most h. Hence $\mathsf{under}(\mathcal{D}, i)(\varepsilon)[j] \geq \mathsf{under}(\mathcal{D}, i)(n)[j] - h \cdot \max((R_1 \cup R_2)^-) = i - h \cdot \max((R_1 \cup R_2)^-) \geq \max(t) \geq t[j]$. On the other hand, if j is not marked, we have that for all node n, $\mathsf{under}(\mathcal{D}, i)(n)[j] = \mathcal{D}(n)[j]$.

Hence $\mathsf{under}(\mathcal{D}, i)(\varepsilon)[j] = \mathcal{D}(\varepsilon)[j] \geq t[j]$. Hence $\mathsf{under}(\mathcal{D}, i)$ is a covering witness of t. □

We now prove the case where the target vector t is coverable. We show that $\mathsf{Under}(\mathcal{B}, C_i)$ contains a truncated derivation covering t, where i is bounded by a doubly exponential function of the input.

Lemma 22. *Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and a vector $t \in \mathbb{N}^k$, if t is coverable in $\mathcal{B}$, then there exists $\mathcal{F} \in \mathsf{Under}(\mathcal{B}, C_i)$ such that $t \leq \mathcal{F}(\varepsilon)$ for some $i = 2^{2^{O(n \log n)}}$, where $n = size(\mathcal{B}) + size(t)$.*

Proof. Let h be the bound from Lemma 20. Clearly, $h = 2^{2^{O(nlogn)}}$. Pick $i = h^2$. By Lemma 20, there is a derivation $\mathcal{D}$ that covers t and whose height is at most h. Since $i = h^2 \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, by Lemma 21, there is a truncated derivation $\mathsf{under}(\mathcal{D}, i)$ that covers t. Moreover, $\mathsf{compact}(\mathsf{under}(\mathcal{D}, i))$ is in $\mathsf{Under}(\mathcal{B}, C_i)$. □

Assume now that the target vector $t \in \mathbb{N}^k$ is not coverable. Lemma 23, from [35], connects derivations of "small" height to extended derivations for high enough constants. Lemma 24 shows that extended derivations of "large" height can be contracted. The proof of this lemma mimicks the proof for (ordinary) derivations.

Lemma 23. *[35] Given a BVAS (k, A, R_1, R_2), a vector $t \in \mathbb{N}^k$, and a derivation $\mathcal{D}$ whose height is at most h, for any bound $i \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, $\mathcal{D}$ is a covering witness of t iff $\mathsf{over}(\mathcal{D}, i)$ is a covering witness of t.*

Lemma 24. *Let $\mathcal{B} = (k, A, R_1, R_2)$ be a BVAS and $i \in \mathbb{N}$. If there is an extended derivation $\mathcal{E}$ that covers $t \in \mathbb{N}^k$, then there is a contraction of $\mathcal{E}$ whose height is at most $(\max((R_1 \cup R_2)^-) + \max(t) + 2)^{(3k)!}$.*

Finally, we prove that if t is not coverable, then $\mathsf{Over}(\mathcal{B}, L_i, C_i)$ does not find an extended derivation covering t, for i as above.

Lemma 25. *Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and $t \in \mathbb{N}^k$, there is an $i = 2^{2^{O(n \log n)}}$, where $n = size(\mathcal{B}) + size(t)$, such that if t is not coverable in $\mathcal{B}$, then for all extended derivations $\mathcal{E} \in \mathsf{Over}(\mathcal{B}, L_i, C_i)$, we have $\mathcal{E}$ does not cover t.*

Proof. Suppose not. Then there is an $\mathcal{E} \in \mathsf{Over}(\mathcal{B}, L_i, C_i)$ so that $\mathcal{E}$ covers t. Let h be the bound from Lemma 24, and let $i = h^2$. We consider two cases: (1) The height of $\mathcal{E}$ is at most h. Then since $i = h^2 \geq h \cdot \max((R_1 \cup R_2)^-) + \max(t)$, by Lemma 23, t is coverable in $\mathcal{B}$. Contradiction. (2) The height of $\mathcal{E}$ is greater than h. By Lemma 24, there is a contraction of $\mathcal{E}$ that covers t and whose height is at most h. Following the arguments in case (1), we again get a contradiction. □

Our main theorem follows from Lemmas 22 and 25.

Theorem 19. *Given a BVAS $\mathcal{B} = (k, A, R_1, R_2)$ and a vector $t \in \mathbb{N}^k$, the EEC algorithm terminates in $2^{2^{O(n \log n)}}$ iterations, where $n = size(\mathcal{B}) + size(t)$.*

The bound on the number of iterations also provides a bound on the overall asymptotic complexity of the algorithm. For BVAS, each iteration of the EEC algorithm performs two instances of AND-OR reachability to perform the cover and the uncover checks. Moreover, the size of the graph is at most doubly exponential in the size of the BVAS, since the finite component of each vector is bounded by a doubly exponential function of the input. Since AND-OR reachability can be performed in time linear in the size of the graph, this gives a 2EXPTIME algorithm. For VAS, each iteration of the EEC algorithm performs two instances of reachability to perform the checks. Thus, if reachability is implemented in a space optimal (NLOGSPACE) way, we get an EXPSPACE upper bound. (In practice, reachability is implemented using a linear time algorithm, which leads to a 2EXPTIME upper bound.)

Chapter 5

BBS: A Phase-Bounded Model Checker for Asynchronous Programs

5.1 Introduction

In many asynchronous applications, a single-threaded worker process interacts with a task queue. In each scheduling step of these programs, the worker takes a task from the queue and executes its code atomically to completion. Executing a task can call "normal" functions as well as post additional asynchronous tasks to the queue. Additionally, tasks can be posted to the queue by the environment. This basic concurrency model has been used in many different settings: in low-level server and networking code, in embedded code and sensor networks [46], in smartphone programming environments such as Android or iOS, and in Javascript. While the concurrency model enables the development of responsive applications, interactions between tasks and the environment can give rise to subtle bugs.

Bouajjani and Emmi introduced *phase-bounding* [13]: a bounded systematic search for asynchronous programs that explores all program behaviors up to a certain phase of asynchronous tasks. Intuitively, the *phase* of a task is defined as its depth in the task tree: the main task has phase 1, and each task posted asynchronously by a task at phase i has phase $i + 1$. Their main result is a sequentialization procedure for asynchronous programs for a given fixed bound L on the task phase.

Though *phase-bounding* was well understood in theory, as far as we are aware, there were no tools that evaluated the practical value of phase-bounding by showing that many bugs in realistic applications can be effectively found within small phase bounds.

We describe our tool BBS[1] that implements phase-bounding to analyze C programs generated from TinyOS applications, which are widely used in wireless sensor networks. Our empirical results indicate that a variety of subtle memory-violation bugs are manifested within a small phase bound (3 in most of the cases). From our evaluation, we conclude that phase-bounding is an effective approach in bug finding for asynchronous programs.

While our evaluation focuses on TinyOS, our tool is generic, and can be ported to other platforms that employ a similar programming model. We leave certain extensions, such as handling multiple worker threads, and the experimental evaluation of this technique to other domains, such as smartphone applications or Javascript programs, for future work.

5.2 Sequentialization Overview

We now give an informal overview of Bouajjani and Emmi's sequentialization procedure. Given an asynchronous program, we first perform the following simple transformation to reduce assertion checking to checking if a global bit is set: (1) we add a global Boolean variable `gError` whose initial value is *false*; (2) we replace each assertion $\mathbf{assert}(e)$ by $\texttt{gError} = !e;\ \mathbf{if}(\texttt{gError})\ \mathbf{return};$; and (3) we add $\mathbf{if}(\texttt{gError})\ \mathbf{return};$ at the beginning of each task's body and after each procedure call. The translation ensures that an assertion fails iff `gError` is *true* at the end of `main`.

Intuitively, the sequentialization replaces asynchronous posts with "normal" function calls. These function calls carry an additional parameter that specifies the phase of the call: the phase of a call corresponding to an asynchronous post is one more than the phase of the caller. The sequentialization maintains several versions of the global state, one for each phase, and calls the task on the copy of the global state at its phase. The task can immediately execute on that global state, without messing up the global state at the posting task's phase. Since tasks are executed in FIFO order, notice that when two tasks t_1 and t_2 are posted sequentially (at phase i, say), the global state after running t_1 is exactly the global state at which t_2 starts executing. Thus, the copy of the

[1] BBS stands for *Buffer phase-Bounded Sequentializer* and can be downloaded at https://github.com/zilongwang/bbs.

global state at phase i correctly threads the global state for all tasks executing at phase i.

The remaining complication is connecting the various copies of the global state. For example, the global state when phase i starts is the same as the global state at the *end* of executing phase $i-1$, but we do not know what that state is (without executing phase $i-1$ first). Here, we use non-determinism. We guess the initial values of the global state for each phase at the beginning of the execution. At the end of the execution, we check that our guess was correct, using the then available values of the global states at each phase. If the guess was correct, we check if some copy of `gError` is set to true: this would imply a semantically consistent run that had an assertion failure.

We now make the translation a bit more precise. Given a phase bound $L \in \mathbb{N}$, i.e., the maximal number of phases to explore, the sequentialization consists of four steps:

1. Track the phase of tasks at which they run in an execution. Intuitively, the phase of `main`, the initial task, is 1, and if a task at phase i executes **post** $p(e)$, then the new task p is at phase $i+1$. As an example, consider an error trace in Figure 5.1, task t_0 is at phase 1, and tasks t_1, t_2 are at phase 2. This tracking can be done by augmenting each procedure's parameter list with an integer k that tracks the phase of the procedure. Consequently, we also replace each normal synchronous procedure call $p(e)$ by $p(e, k)$, and each asynchronous call **post** $p(e)$ by **post** $p(e, k+1)$.

2. Replace each **post** $p(e, k+1)$ by **if**$(k < L)$ $p(e, k+1)$;, meaning that if some task at phase k posts the task p and $k+1$ does not exceed the phase bound L, the task p is immediately called and runs at phase $k+1$ instead of putting it into the task queue.

3. For each global variable g, create L copies of it, denoted by $g[1], \ldots, g[L]$. Set the initial value of the first copy $g[1]$ to the initial value of g, and nondeterministically guess the initial values of the other copies. For each statement of a program, if g appears, then replace it by $g[k]$. Intuitively, the i-th copy of global variables is used to record the evolution of global valuations along an execution at phase i.

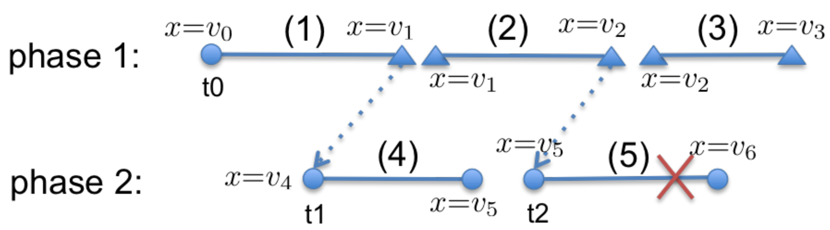


FIGURE 5.1: An error trace before sequentialization. Circles denote the starting or ending points of tasks. Solid lines denote the execution of tasks. Triangles with dashed arrows indicate a post statement that posts a task to the queue; triangles without dashed arrows are statements right after post statements. The cross represents where the assertion fails. This error trace is read as follows: task t_0 runs, posts tasks t_1 and t_2 to the task queue, and completes. Then t_1 and t_2 runs one after another. We divided the error trace into execution segments (1)–(5), ordered by their execution order. Values of the global state x at each segment are shown. E.g., when segment (1) starts and ends, $x = v_0$ and $x = v_1$, respectively. When segment (4) starts and ends, $x = v_4$ and $x = v_5$, respectively. Note that due to the FIFO order, $v_3 = v_4$.

4. Run the initial task t_0 at phase 1. When t_0 returns, for each phase $i \in [2, L]$, enforce that the guessed initial values of the i-th copy are indeed equal to the final values of the $(i-1)$-th copy. Finally, a bug is found if some copy of `gError` equals $true$.

Step 4 is better explained through an example. We present how a sequentialized execution in Figure 5.2 is related to an error trace of Figure 5.1. Suppose that the phase bound $L = 2$ and the above first three steps have been done correctly.

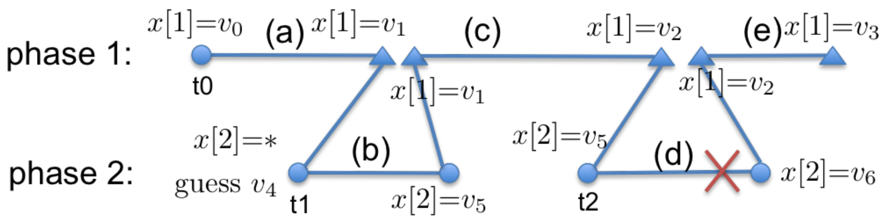


FIGURE 5.2: The sequentialized error trace after sequentialization. Values of each copy of the global state x at each segment are shown. E.g., when segment (a) starts and ends, the first copy $x[1] = v_0$ and $x[1] = v_1$, respectively. When segment (b) starts, the second copy $x[2]$ is guessed to v_4. When segment (b) ends, $x[2] = v_5$.

Consider segment (a) in Figure 5.2 and segment (1) in Figure 5.1. When task t_0 starts, notice that the global state x in segment (1) and its first copy $x[1]$ in segment (a) are always the same because both are initialized to v_0, and in each step of their executions, the way that segment (1) modifies x is the same as the way that segment

(a) modifies $x[1]$. In this case, we say that segment (a) uses the first copy of the global state to "mimic" the evolution of the global state in segment (1).

Since the last statement of segment (a) is **if**$(k < L)\ p(e, k+1)$; and the current phase $k = 1$, segment (b) starts. Notice that segment (b) runs at phase 2 and only modifies the second copy of the global state $x[2]$. Additionally, if we assume that the initial value of $x[2]$ are guessed correctly, i.e., v_4, shown in Figure 5.2, then segment (b) uses the second copy of the global state to "mimic" the evolution of the global state in segment (4).

After segment (b) completes, the control goes back to phase 1 and segment (c) starts. Note that segment (b) does not modify the first copy $x[1]$, and hence when segment (c) starts, the value of $x[1]$ is still v_1. As a result, segment (c) uses the first copy of the global state to "mimic" the evolution of the global state in segment (2).

After segment (c) completes, segment (d) starts. Note that since segment (c) does not modify the second copy $x[2]$, the value of $x[2]$ is still v_5 at the beginning of segment (d), which is the same as the value of x at the beginning of segment (5). Hence segment (d) uses $x[2]$ to mimic x in segment (5). When segment (d) completes, segment (e) starts to use the first copy $x[1]$ to mimic segment (3).

Finally, When segment (e) completes, by using `assume` statements, we enforce that the initial value for the second copy $x[2]$ is indeed guessed to v_4 in order to satisfy the FIFO order imposed by the task queue. After the enforcement, the sequential execution in Figure 5.2 and the error trace in Figure 5.1 reach exactly the same set of global states. Hence we conclude that a bug is found.

5.3 Experimental Evaluation

We first provide a brief introduction to TinyOS applications. We then present the design of BBS and elaborate on our experimental results.

5.3.1 TinyOS Execution Model

TinyOS [58] is a popular operating system designed for wireless sensor networks. It uses nesC [46] as the programming language and provides a toolchain that translates nesC programs into embedded C code and then compiles the C code into executables which are deployed on sensor motes to perform operations such as data collection.

TinyOS provides a programming language (nesC) and an execution model tailored towards asynchronous programming. A nesC program consists of tasks and interrupt handlers. When the program runs, TinyOS associates a scheduler, a stack, and a task queue with it, and starts to run the "main" task on the stack. Tasks run to completion and can post additional tasks into the task queue. When a task completes, the scheduler dequeues the first task from the task queue, and runs it on the stack.

Hardware interrupts may arrive at any time (when the corresponding interrupt is enabled). For instance, a timer interrupt may occur periodically so that sensors can read meters, or a receive interrupt may occur to notice sensors that packets arrived from outside. When an (enabled) interrupt occurs, TinyOS pre-empts the running task and executes the corresponding interrupt handler defined in the nesC program. An interrupt handler can also post tasks to the task queue, which is used as a mechanism to achieve deferred computation and hide the latency of time-consuming operations such as I/O. Once the interrupt handler completes, the interrupted task resumes.

5.3.2 BBS Overview

We implemented BBS to perform phase-bounded analysis for TinyOS applications. BBS checks user-defined assertions as well as two common memory violations in C programs: out-of-bound array accesses (OOB) and null-pointer dereference.

The workflow of BBS is shown in Figure 5.3. First, given a TinyOS application consisting of nesC files, the nesC compiler `nescc` combines them together and generates a self-contained embedded C file. `nescc` supports many mote platforms and generates different embedded C code based on platforms. In our work, we let `nescc` generate embedded C code for MSP430 platforms.

BBS takes as inputs the MSP430 embedded C file containing assertions and a phase bound, and executes three modules.

The first module performs preprocessing and static analysis on the C program to instrument interrupts and assertions. Interrupt handlers are obtained from `nescc`-generated attributes in the code. A naive way to instrument interrupts is to insert

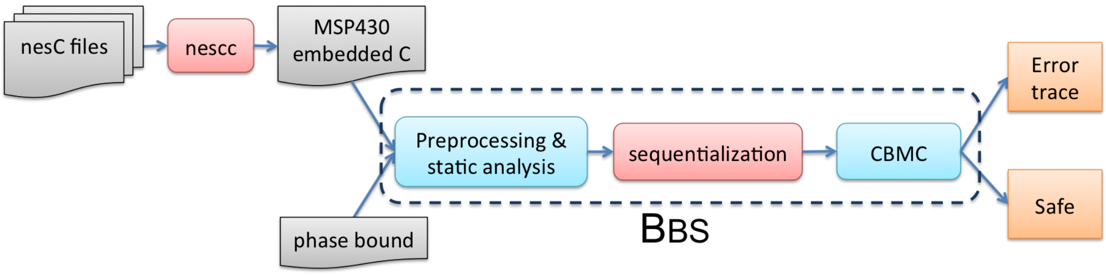


FIGURE 5.3: The workflow of BBS

them before each statement of the C program. However, if a statement does not have potentially raced variables[2], we do not need to instrument interrupts before it, because the execution of such statements commutes with the interrupt handler: either order of execution leads to the same final state. Thus BBS performs static analysis to compute potentially raced variables and instruments interrupts accordingly.

The second module implements the sequentialization algorithm. The resulting sequential C program is fed into the bounded model checker CBMC [28, 73], which outputs either an error trace or "program safe" up to the phase bound and the bound imposed by CBMC.

5.3.3 Experimental Experience with BBS

We used BBS to analyze eight TinyOS applications in the `apps` directory from TinyOS's source tree. These benchmarks cover most of the basic functionalities provided by a sensor mote such as timers, radio communication, and serial transmission.

In Table 5.1, we summarize the size and complexity of these benchmarks in terms of (1) lines of code in the cleanly reformatted ANSI C program after the preprocessing stage, (2) the number of types of tasks that can be posted, (3) the number of types of hardware interrupts that are expected, (4) the number of global variables as well as the number of potentially raced variables (found by the static analysis).

In each of the first three benchmarks, we manually injected a realistic memory violation bug that programmers often make. The rest five benchmarks were previously known to be buggy [17, 30, 80, 108]. The TestSerial benchmark contains two bugs and

[2] A potentially raced variable is accessed by both tasks and interrupt handlers, and at least one access from both is a write.

Benchmark	LOC	Tasks	Interrupt Types	Global variables	Potentially raced global variables
TestAdc	6738	9	2	100	19
TestEui	7467	13	3	138	17
TestAM	11259	13	5	154	27
BlinkFail	3153	3	1	64	5
TestSerial	6590	10	3	127	17
TestPrintf	6882	13	3	136	18
TestDissemination	13004	17	5	166	37
TestDip	17091	25	7	243	49

TABLE 5.1: TinyOS benchmarks

Benchmark	Bug type	Min phase	Time		Error Trace
			Seq. (s)	CBMC (s)	(in steps)
TestAdc	NullPtr	2	3.92	15.92	2014
TestEui	OOB	2	3.97	12.78	9425
TestAM	NullPtr	3	5.88	342.99	12925
BlinkFail	OOB	3	2.55	2.69	3773
TestSerial	OOB	4	3.75	23.92	13531
	User-defined	4		39.01	14161
TestPrintf	OOB	3	3.78	30.32	14154
TestDissemination	NullPtr	3	5.95	843.68	17307
TestDip	NullPtr	3	7.69	681.81	20274

TABLE 5.2: Experimental results

each of the rest has one bug. We ran BBS on these benchmarks to see whether it could find these bugs efficiently within small phase bounds.

Experimental results All experiments were performed on a 2 core Intel Xeon X5650 CPU machine with 64GB memory and 64bit Linux (Debian/Lenny). Table 5.2 lists the analysis results, showing that BBS successfully uncovered all bugs that are injected in the first three benchmarks, as well as all previously known bugs in the rest five benchmarks. We report the type of bugs, the minimal phases that are required to uncover the bugs, the time used in both sequentialization and CBMC, and the lengths of error traces. Notice that all bugs were found within small phase bounds, that is, at most 4 phases. This result indicates that the phase-bounded approach effectively uncovers interesting bugs within small phase bounds for realistic C programs.

Chapter 6

LLSPLAT: A Concolic Testing Tool with Bounded Model Checking

6.1 Introduction

With the increasing power of computers and advances in constraint solving technologies, an automated *dynamic* testing technique called concolic testing [52, 111] has received much attention due to its low false positives and high code coverage [21, 25]. Concolic testing runs a program under test with a random input vector. It then generates additional input vectors by analyzing previous execution paths. Specifically, concolic testing selects one of the branches in a previous execution path and generates a new input vector to steer the next execution toward the opposite branch of the selected branch. By carefully selecting branches for the new inputs, concolic testing avoids generating redundant input vectors that execute the same program path, and thus *enumerates* all non-redundant program paths. In practice, concolic testing suffers from *path explosion*: it has to enumerate a *huge* number of non-redundant execution paths [3, 21, 25, 50].

On the other hand, *bounded model checking* (BMC) [29, 31, 73, 92] is a fully symbolic testing technique. Given a program under test and a bound k, BMC unrolls loops and inlines function calls k times to construct an *acyclic* program which is an under-approximation of the original program. It then performs *verification condition* (VC) generation over the acyclic program to obtain a formula which encodes the acyclic program and a property to check. The formula is then fed into a SAT solver. If the formula is proved to be valid by the solver, the property holds. Otherwise, the solver provides a model from which we can extract an execution of the program that violates the property. BMC provides a way to encode and reason about multiple execution paths using a single formula, but its scalability is often limited by deterministic dependencies between program paths and data values.

A natural question is *whether there is a way to combine concolic testing with BMC to boost the exploration among the huge number of program paths*? In this dissertation, we provide a positive answer and propose a concolic+BMC algorithm. Intuitively, given a program under test, the algorithm starts with the per-path search mode in concolic testing while referring to the control flow graph (CFG) of the program to identify easy-to-analyze portions of code that do not contain loops, recursive function calls, or other instructions that are difficult to generate formulas using BMC. Whenever a concolic execution encounters such a portion, the algorithm switches to the BMC mode and generates a BMC formula for the portion, and identifies a frontier of hard-to-analyze instructions. The BMC formula summarizes the effects of all execution paths through the easy-to-analyze portion up to the hard frontier. When the concolic execution reaches the frontier, the algorithm switches back to the per-path search mode to handle the cases that are difficult to summarize by BMC.

We have developed LLSPLAT, a tool that implements the concolic+BMC algorithm for C programs. We evaluate LLSPLAT with two state-of-the-art concolic testing tools CREST [19] and KLEE [20], using 36 programs from SVCOMP15 [117]. The evaluation shows that (1) for the same time budget (an hour per program), LLSPLAT provides on average 31%, 19%, 20%, 21% higher branch coverage than CREST's four search strategies, and on average 21% higher branch coverage than KLEE, and (2) LLSPLAT achieves higher branch coverage quickly: in our experiments, it starts to outperform CREST and KLEE after at most 3 minutes. In addition, we also evaluate LLSPLAT with the state-of-the-art bounded model checker CBMC [73] using 13 sequentialized SystemC benchmarks. The experiments show that LLSPLAT finds bugs more quickly than CBMC.

6.2 A Motivating Example

We illustrate the inadequacy of concolic testing and BMC acting alone, and the benefits of their combination, using the function `foo` below. The function runs in an infinite loop, and receives two inputs in each iteration. One input `c` is a character and the other input `s` is a character array. The function `foo` hits an error if the variable `state` is 9 and the input array `s` holds the string "reset". Similar functions like `foo` are often generated by lexers.

```
void foo() {
 char c, s[6];
 int state = 0;

 while(1) {
   c = input(); s = input();

   if (c == '[' && state == 0) state = 1;
   if (c == '(' && state == 1) state = 2;
   if (c == '{' && state == 2) state = 3;
   if (c == '~' && state == 3) state = 4;
   if (c == 'a' && state == 4) state = 5;
   if (c == 'x' && state == 5) state = 6;
   if (c == '}' && state == 6) state = 7;
   if (c == ')' && state == 7) state = 8;
   if (c == ']' && state == 8) state = 9;
   if (s[0] == 'r' && s[1] == 'e' && s[2] == 's' &&
       s[3] == 'e' && s[4] == 't' && s[5] == 0 && state == 9)
     goto ERROR;
 }
ERROR: assert(0);
}
```

LISTING 6.1: A motivating example

To reveal the error in the function `foo`, concolic testing systematically explores all execution paths of the function. Since the function `foo` runs in an infinite loop, the number of distinct feasible executions is infinite. To perform concolic testing we need to bound the number of iterations of the loop if we perform a depth-first search of the execution paths. There are 17 possible choices of values of `c` and `s` that concolic testing would consider, and at least 9 iterations are required to hit the error. Hence, concolic testing will explore about $17^9 \approx 10^{11}$ execution paths. It is unlikely that concolic testing can hit the error in a reasonable time budget. We confirm this fact by testing the function `foo` using CREST [19] and KLEE [20]. Both tools could not hit the error in an hour. It is worth mentioning that, if there were code consisting of many conditionals after the ERROR label instead of the assertion, things would get even worse because concolic testing cannot reach them, which is a primary reason for poor branch coverage.

On the other hand, BMC by itself does not reveal the error in `foo` either. BMC relies on the user to figure out an appropriate unrolling bound k to reach a deep error. To prevent inadequate unrolling bound which leads to unsoundness, BMC adds an *unrolling assertion* $assert(\neg cond)$ as the last statement of the k-th unrolling, where $cond$ is the looping condition of the unrolled loop. Thus, no matter what unrolling bound k is set for the infinite loop in `foo`, an unrolling assertion $assert(0)$ is always added, which prevents BMC from hitting the actual error. We validated this fact by running the example with CBMC [73].

In our concolic+BMC approach, whenever a concolic execution encounters a conditional, it has a choice either to save a predicate representing that a particular branch is taken along the execution as concolic testing does, or to save a BMC formula, for example, that encodes the entire conditional. Which choice is taken depends on whether the conditional is "simple" enough to generate a BMC formula easily. For example, a conditional is simple if there are no loops and recursive function calls[1] inside it.

[1]The program size after function inlining can be exponentially larger than the size of the original program.

Program	P	::=	$(\text{var } g)^* \cdot Fn^+$
Functions	Fn	::=	$f((\text{var } p)^*) \cdot (\text{var } l)^* \cdot BB^+$
Basic blocks	BB	::=	$Inst^* \cdot TermInst$
Instructions	$Inst$	::=	$x \longleftarrow e \mid f(e^*) \mid x \longleftarrow input()$
Basic block terminators	$TermInst$	::=	$ret \mid br\ e\ BB1\ BB2 \mid br\ BB \mid ERROR$
Variables	x	::=	$g \mid p \mid l$

FIGURE 6.1: Program Model

Since all conditionals are simple in function `foo`, the concolic+BMC approach can easily hit the error. We validated this fact by using LLSPLAT to test function `foo`. LLSPLAT found the bug in 3s.

6.3 Concolic Testing

LLSPLAT implements the concolic testing algorithm used in DART/CUTE [52, 111]. We first review the algorithm, and then describe how LLSPLAT modifies it.

6.3.1 Program Model

We describe how concolic testing works on a simple language shown in Figure 6.1. A *program* consists of a set of *global* variables and a set of *functions*. Each function consists of a name, a sequence of formal parameters, a set of *local* variables, and a set of *basic blocks* representing the *control flow graph* (CFG) of the function. Each basic block consists of a list of *instructions* followed by a *terminating instruction*. There are three types of instructions: $x \longleftarrow e$ is an assignment, $f(e^*)$ is a function call, and $x \longleftarrow input()$ indicates that the variable x is a program input. There are four types of terminating instructions: ret is a return instruction, $br\ e\ BB1\ BB2$ is a conditional branch, $br\ BB$ is an unconditional branch, and $ERROR$ indicates program abortion. We omit an explicit syntax of expressions. We assume there is an entry function `main` that is not called anywhere. We assume each function has an *entry* basic block, and every basic block of the function is reachable from it.

6.3.2 The Concolic Testing Algorithm

To test a program P, concolic testing tries to explore all execution paths of P. It first instruments the program P by Algo 2, and outputs an instrumented program P'. Ignore the red-highlighted lines in the algorithms for now because they are used in the concolic+BMC approach we describe later. Algo 3 repeatedly runs the instrumented program P'. Due to limited space, we omit the instrumentation for function calls, and the code that bounds the search depth in the search strategy — these are identical to previous work [52, 111].

Algo 2 first makes a copy P' of the program P, and inserts various global variables and function calls which are used for the symbolic execution. It then returns the instrumented program P'. Algo 4 presents the definitions of the instrumented functions. The expressions enclosed in double quotes (“e”) represent syntactic objects. We denote $\&x$ to be the address of a variable x.

The function $initInput(\text{“}x\text{”})$ initializes the input variable x using the *input map* I in all runs except the first. The variable x is assigned randomly in the first run. The function also saves a fresh symbolic variable for x in the symbolic store.

The function $updateSymStore(\text{“}x\text{”}, \text{“}e\text{”})$ updates x's symbolic expression in the symbolic store $symStore$ based on the expression e. We write $symexpr(\text{“}e\text{”})$ to represent the symbolic expression by substituting each variable v in “e” with its symbolic expression $symStore[\&v]$. For example, if “e” $=$ “$a + b$”, $symStore[\&a] = e_a$, and $symStore[\&b] = e_b$, then $symexpr(\text{“}e\text{”}) = e_a + e_b$.

The function $addPathConstraint(\text{“}e\text{”}, e)$ updates the *path constraint* $pathC$ and the *coverage history* $branch\_hist$. Symbolic predicate expressions from the branching points are collected in the list $pathC$. At the end of the execution, $pathC$ contains all predicates whose *conjunction* holds for the execution path. To explore paths of the program under test, each run (except the first) is executed based on the coverage history computed in the previous run. The coverage history is a list of $BranchNode$s. A $BranchNode$ has two boolean fields: $isCovered$ records which branch is taken, and $done$ records whether both branches have executed in prior runs (with the same history up to this branch node).

Algorithm 2: Instrumentation

```
Program instrument(P):
    P' ⟵ P
    Add to P' global vars i ⟵ 0, inputNo ⟵ 0, symStore ⟵ [], pathC ⟵ []
    Govs ⟵ {BB | BB is a governor in P}
    Add to P' global vars bmcNo ⟵ 0, currGov ⟵ None, init ⟵ None
    foreach BB ∈ P' do
        if BB ∈ GR(gov) for some gov ∈ Govs then continue
        foreach Inst ∈ BB do
            switch Inst do
                case x ⟵ input()
                    Replace Inst by InitInput("x")
                case x ⟵ e
                    Add updateSymStore("x", "e") before Inst
                case br e BB1 BB2
                    if BB ∈ Govs then
                        Add startBMC(BB) before Inst
                        foreach d ∈ Dests(BB) do
                            Add endBMC(BB, d) as the 1st instruction of d
                    else
                        Add addPathConstraint("e", e) before Inst
                case Return
                    if Inst is in the main function then
                        Add SolveConstraint() before Inst
                case ERROR
                    Add print("ERROR found") before Inst
    return P'
```

The function $solveConstraint()$ determines new inputs that forces the next run to execute the last unexplored branch of the j-th conditional in *branch_hist*.

6.4 Combining Concolic Testing with BMC

We now present the concolic+BMC algorithm. The key observation is that given a program P under test, the instrumented program for P can additionally refer to the (static) CFG of P and perform static analysis at run time. Section 6.4.1 describes how to identify program portions for BMC formula generation. Section 6.4.2 describes the BMC formula generation algorithm. Section 6.4.3 integrates this with concolic testing.

Algorithm 3: run_llsplat

```
void run_llsplat(P):
    I ⟵ []; branch_hist ⟵ []; completed ⟵ false
    CFG_P ⟵ CFGofProgram(P)
    while ¬completed do execute instrument(P)
```

6.4.1 Identifying Program Portions for BMC

Preliminaries Given a CFG, a basic block m *dominates* a basic block n if every path from the entry basic block of the CFG to n goes through m. We denote $Dom(m)$ to be the set of basic blocks which m dominates. A depth-first search of the CFG forms a *depth-first spanning tree* (DFST). There are edges in CFG that go from a basic block m to an ancestor of n in DFST (possibly to m itself). We call these edges *back edges*, and recall the following result [32].

Lemma 26. *A directed graph is acyclic iff a depth-first search yields no back edge.*

Governors, Governed Regions, and Destinations Given a basic block m, a basic block $n \in Dom(m)$ is *polluted* in $Dom(m)$ in the following four cases: (1) n contains function call instructions, (2) n has no successors, (3) n is the source or the target of a back edge, or (4) n is reachable from a polluted basic block $k \in Dom(m)$. A basic block m *effectively dominates* a basic block n if $n \in Dom(m)$ and n is not polluted in $Dom(m)$. We denote $Edom(m)$ to be the set of basic blocks that m effectively dominates.

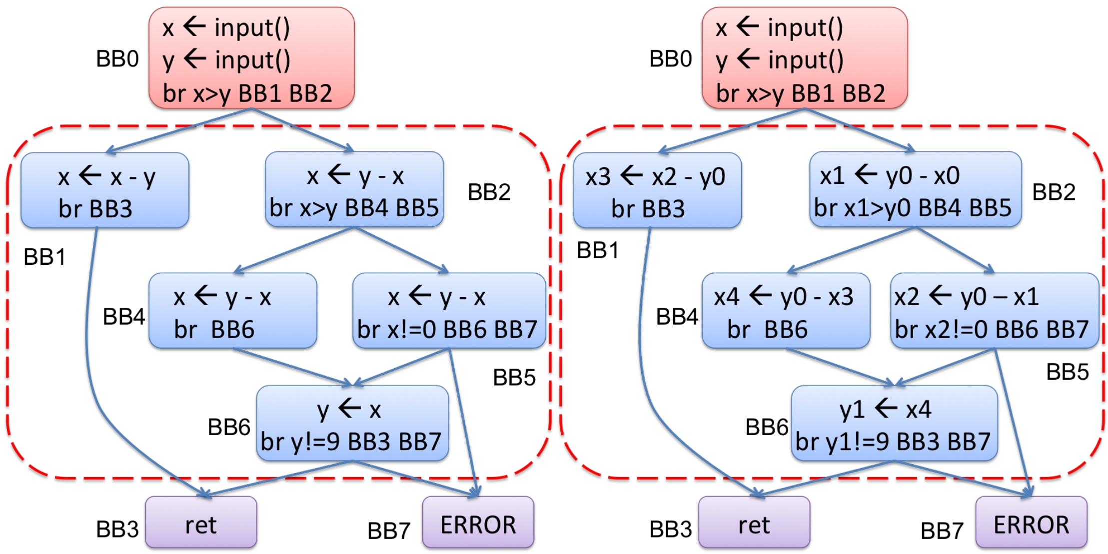
A basic block m is called a *governor candidate* if (1) the terminating instruction of m is of the form $br\ \ e\ \ BB1\ \ BB2$, (2) m dominates both $BB1$ and $BB2$, and (3) $Edom(BB1)$ and $Edom(BB2)$ are not empty. Given a governor candidate m with its two successors $BB1$ and $BB2$, the *governed region* of m, denoted by $GR(m)$, is $Edom(BB1) \cup Edom(BB2)$. A basic block n is a *destination* of $GR(m)$ if $n \notin GR(m)$ and n is a successor of some basic block $k \in GR(m)$. Let the set $Dests(m)$ be all destinations of $GR(m)$. A basic block gov is a *governor* if gov is a governor candidate, and there is no governor candidate m with $gov \in GR(m)$. We prove the following lemma about governors. The proof is in Section 6.7.

Lemma 27. *For any governor gov, (1) $GR(gov)$ is acyclic and does not have function calls, (2) gov dominates every basic block $BB \in GR(gov)$.*

Algorithm 4: Concolic Testing

```
void InitInput("x"):
    inputNo ⟵ inputNo + 1
    j ⟵ inputNo
    if I[j] is undefined then
        x ⟵ random()
        I[j] ⟵ x
    else
        x ⟵ I[j]
    // sym_j is a fresh variable for x
    symStore[&x] ⟵ sym_j

void updateSymStore("x", "e"):
    symStore[&x] ⟵ symexpr("e")

struct BranchNode:
    isCovered : bool
    done : bool

void addPathConstraint("e", b):
    if b then
        pathC[i] ⟵ symexpr("e")
    else
        pathC[i] ⟵ ¬symexpr("e")
    if i < |branch_hist| then
        if i = |branch_hist| − 1 then
        branch_hist[i].done ⟵ true
    else
        branch_hist[i] ⟵
        BranchNode(isCovered : b, done :
        false)
    i ⟵ i + 1

void SolveConstraint():
    j = i - 1
    while j ≥ 0 do
        if ¬branch_hist[j].done then
            if branch_hist[j] is BmcNode then
                foreach d such that ¬branch_hist[j].isCovered[d] do
                    if ⋀_{0≤k≤j−1} pathC[k] ∧ rmLastDest(path_c[j]) ∧ ⋁_{c∈Edges_d[d]} c has a
                    solution I′ then
                        branch_hist ⟵ branch_hist[0..j]
                        I ⟵ I′
                        return
                j ⟵ j − 1
            else
                branch_hist[j].isCovered ⟵ ¬branch_hist[j].isCovered
                pathC[j] ⟵ ¬pathC[j]
                if pathC[0..j] has a solution I′ then
                    branch_hist ⟵ branch_hist[0..j]
                    I ⟵ I′
                    return
                j ⟵ j − 1
        else
            j = j − 1
    if j < 0 then completed ⟵ true
```

Example Consider the program in Fig 6.2a. $BB0$ is a governor. Its governed region $GR(BB0)$ includes $BB1$, $BB2$, $BB4$, $BB5$, and $BB6$, which are inside the red dash circle. $BB3$ and $BB7$ are the destinations in $Dests(BB0)$. Though $BB2$ is a governor candidate, it is not a governor because it is in $GR(BB0)$.



(A) A program under test (B) Variable renaming

FIGURE 6.2: An Example

6.4.2 Translating Governed Regions to BMC Formulas

A governed region is ideal for generating a BMC formula because it is acyclic, does not have function calls, and is "sufficiently" large in the sense that it includes as many (unpolluted) basic blocks as its governor governs. We present our algorithm that translates a governed region to a BMC formula, and provide an example.

The BMC Formula Generation Algorithm

Given a governor gov, we construct a BMC formula ϕ for $GR(gov)$ in five steps:

1. Renaming variables in $GR(gov)$ into an SSA-form. Let $AccVars$ be the set of variables accessed by the instructions in $GR(gov)$. Let a *version map* $\mathcal{V}$ be a map from each variable $x \in AccVars$ to a variable x_α with a version $\alpha \in \mathbb{N}$. We naturally extend the notation $\mathcal{V}$ to expressions: we denote $\mathcal{V}(e)$ to be an expression that replaces each variable x in e by $\mathcal{V}(x)$. Since $GR(gov)$ is acyclic, there exists a topological ordering over the basic blocks in $GR(gov)$. Without loss of generality, let $BB_1, BB_2, \ldots, BB_n$ be the list of all basic blocks in $GR(gov)$ after a topological sort, where n is the number of basic blocks in $GR(gov)$. For each $1 \leq i \leq n$ and

each instruction I in BB_i, we rename each variable in I according to the version map $\mathcal{V}$, and update the version map $\mathcal{V}$. Initially, for each $x \in AccVars$, $\mathcal{V}(x) = x_0$. If I is an assignment $x \longleftarrow e$ and $\mathcal{V}(x) = x_\alpha$, then we rewrite I to $x_{\alpha+1} \longleftarrow \mathcal{V}(e)$ and set $\mathcal{V}(x) = x_{\alpha+1}$. If I is a conditional branch $br\ e\ BB1\ BB2$, then we rewrite I to $br\ \mathcal{V}(e)\ BB1\ BB2$.

2. Create a boolean variable g_{BB} for each basic block $BB \in GR(gov)$.

3. Compute an *edge map* $Edges$ that maps each basic block $BB \in GR(gov) \cup Dests(gov)$ to a list of *edge formulas* as follows. For each $BB \in GR(gov)$, if its terminating instruction is $br\ e\ BB1\ BB2$, then we add $g_{BB} \wedge e$ to $Edges[BB1]$, and add $g_{BB} \wedge \neg e$ to $Edges[BB2]$; if it is $br\ BB1$, then we add g_{BB} to $Edges[BB1]$. Let the governor's terminating instruction be $br\ e\ BB1\ BB2$. Let e_0 be an expression obtained by replacing each variable x in e with x_0. We set $Edges[BB1] = e_0$ and $Edges[BB2] = \neg e_0$.

4. Compute a *block map* $Blks$ that maps each basic block $BB \in GR(gov)$ to a *block formula*. For each $BB \in GR(gov)$, let $I_1, I_2, \ldots, I_k$ be the non-terminating instructions in BB. For each $1 \leq i \leq k$, if I_i is $x_\alpha \longleftarrow e$, we define an *instruction formula* c_i to be $x_\alpha = ite(g_{BB}, e, x_{\alpha-1})$. We set $Blks[BB] = \bigwedge_{1 \leq i \leq k} c_i$.

5. Create the final BMC formula ϕ, defined as follows:

$$\phi : \bigwedge_{BB \in GR(gov)} \left(\left(g_{BB} = \bigvee_{c \in Edges[BB]} c \right) \wedge Blks[BB] \right)$$

Intuitively, ϕ claims that for each basic block $BB \in GR(gov)$, (1) BB is taken (i.e., g_{BB} is true) if one of its predecessor is taken, and (2) the block formula of BB must hold.

Our BMC formula generation algorithm has the following important property. The proof is in Section 6.7.

Theorem 20. *Let gov be a governor and T be an arbitrary topological ordering over $GR(gov)$. After the BMC algorithm is done w.r.t. T, for any destination $d \in Dests(gov)$, (1) the formula*

BB	$Edges[BB]$	$Blks[BB]$
$BB1$	$\{x0 > y0\}$	$x3 = ite(g_{BB1}, x2 - y0, x2)$
$BB2$	$\{\neg(x0 > y0)\}$	$x1 = ite(g_{BB2}, y0 - x0, x0)$
$BB3$	$\{g_{BB1}, g_{BB6} \wedge y1 \neq 9\}$	
$BB4$	$\{g_{BB2} \wedge x1 > y0\}$	$x4 = ite(g_{BB4}, y0 - x3, x3)$
$BB5$	$\{g_{BB2} \wedge \neg(x1 > y0)\}$	$x2 = ite(g_{BB5}, y0 - x1, x1)$
$BB6$	$\{g_{BB4}, g_{BB5} \wedge x2 \neq 0\}$	$y1 = ite(g_{BB6}, x4, y0)$
$BB7$	$\{g_{BB5} \wedge \neg(x2 \neq 0), g_{BB6} \wedge \neg(y1 \neq 9)\}$	

TABLE 6.1: Edge formulas and block formulas

$\phi \wedge \bigvee_{c \in Edges[d]} c$ encodes all executions from gov to d, and (2) for every execution ρ from gov to d, the final version of each variable x in ϕ represents the value of x when ρ enters d.

Example We illustrate our BMC algorithm by reusing the example in Fig 6.2a. The topological order we use for the variable renaming is $BB2, BB5, BB1, BB4, BB6$. After variable renaming, the resulting program is in Fig 6.2b. After Step 4 of the algorithm, the edge map $Edges$ and the block map $Blks$ are shown in Table 6.1.

To give a flavor of the correctness of Theorem 20(2), we examine an execution ρ : $BB0, BB1, BB3$ as an example. When ρ enters the destination $BB3$, the largest version of x and y along ρ is $x3$ and $y0$, but their final versions in ϕ are $x4$ and $y1$. However, since $BB2, BB4, BB5$ and $BB6$ are not taken along ρ, we have $x4 = x3$, $x2 = x1 = x0$, and $y1 = y0$. Since $BB1$ is taken, we have $x3 = x2 - y0$. Thus $x4 = x0 - y0$ and $y1 = y0$. We conclude that $x4$ and $y1$ represent the values of x and y when ρ enters the destination $BB3$.

6.4.3 Integrating BMC Formulas with Concolic Testing

To integrate BMC with concolic testing, we add the red lines in Algo 2, 3, and 4. During the instrumentation in Algo 2, we first compute a set $Govs$ of all governors of the program P. Since basic blocks in governed regions are used to generate BMC formulas, we skip instrumenting them. When a basic block BB has two successors, if BB is a governor, we instrument a function call $startBMC(BB)$ before BB's terminating instruction, and for each destination $d \in Dests(BB)$, we instrument a function call $endBMC(BB, d)$ as the first instruction of d. If BB is not a governor, we perform the old instrumentation in concolic testing.

In Algo 3, we read the CFG of the uninstrumented program P because it is used to generate BMC formulas along concolic executions.

Algorithm 5: startBMC and endBMC

```
void startBMC(gov):
    currGov ⟵ gov; bmcNo ⟵ bmcNo + 1
    init ⟵ ⋀_{x∈AccVars(gov)} (x_0^{bmcNo} = symStore[&x]) // x_0^{bmcNo} is a fresh variable
struct BmcNode:
    isCovered : BasicBlock ⟶ bool
    Edges_d : BasicBlock ⟶ formula
    done : bool
void endBMC(gov, d):
    if currGov ≠ gov then return
    (φ, V_final, Edges) ⟵ doBMC(CFG_P, gov)
    if i <| branch_hist | then
        if i =| branch_hist | −1∧∀d′ ∈ Dests(gov)\{d}. branch_hist[i].isCovered[d′]
        then branch_hist[i].done ⟵ true
    else
        branch_hist[i] ⟵ BmcNode(isCovered : λdest ∈ Dests(gov). false,
        Edges_d : λdest ∈ Dests(gov). addSup(Edges[dest], bmcNo), done : false)
    branch_hist[i].isCovered[d] ⟵ true
    pathC[i] ⟵ init ∧ addSup(φ ∧ ⋁_{c∈Edges[d]} c, bmcNo)
    i ⟵ i + 1
    foreach x ∈ AccVars(gov) do SymStore[&x] ⟵ addSup(V_final(x), bmcNo)
```

The definition of $startBMC(gov)$ is given in Algo 5. It saves the governor gov that will be used to generate a BMC formula using $currGov$. Then it increments $bmcNo$, which records the number of BMC formulas that have been generated so far along the concolic execution. It then uses $init$ to "glue" the execution before entering $GR(gov)$ with the BMC formula for $GR(gov)$. More concretely, for each variable $x \in AccVars(gov)$, an equation $x_0^{BmcNo} = symStore[\&x]$ is created, and $init$ is the conjunction of all such equations. Intuitively, the initial version x_0^{BmcNo} represents the value of x when the concolic execution enters $GR(gov)$, which is also represented by $symStore[\&x]$.

The definition of $endBMC(gov, d)$ is given in Algo 5. If the passed-in governor gov is the one saved in $currGov$, it performs the BMC generation algorithm described in Section 6.4.2 to obtain a BMC formula ϕ for the governed region $GR(gov)$, the final version map $\mathcal{V}_{final}$, and the edge map $Edges$. Moreover, the coverage history $branch\_hist$ is updated. We extend $branch\_hist$ to be a list of $BranchNode \cup BmcNode$. A $BmcNode$

has three fields: $isCovered$ records which destinations have been covered in prior runs, $Edges\_d$ maps each destination to its edge formulas, and $done$ records whether all destinations have been covered in prior runs. Given a formula ψ and a number j, we denote $addSup(\psi, j)$ to be the formula obtained by replacing each variable x in ψ with a new variable x^j. We first create a formula $\phi \wedge \bigvee_{c \in Edges[d]} c$ which represents all executions from the governor gov to the destination d by Theorem 20. Since the governed region may be reached multiple times along an execution, we compute a formula $\psi \equiv addSup(\phi \wedge \bigvee_{c \in Edges[d]} c, bmcNo)$ which specifies that ψ is the $bmcNo$-th BMC formula along the execution. We then add $init \wedge \psi$ to the path constraint. Finally, to let the concolic execution proceed, for each variable $x \in AccVars(gov)$, we update the symbolic store so that $symStore[\&x]$ represents the value of x when the execution enters the destination d. By Theorem 20, no matter which execution from gov to d is taken, the final version $\mathcal{V}_{final}(x)$ always represents the value of x at that moment. Thus, we set $symStore[\&x]$ accordingly.

The function $SolveConstraint$ is extended as shown in Algo 4. If the node $branch\_hist[j]$ is a $BmcNode$, we find an uncovered destination d, and asks if there is an execution that goes to d. The formula $rmLastDest(pathC[j])$ is defined by removing the disjunction of edge formulas of d' from $pathC[j]$ where d' is the destination covered by the just terminating execution. If there are new inputs I' for such an execution to d, a new run is started with inputs I'.

Example

We again reuse the example in Fig 6.2a. Suppose LLSPLAT randomly generates $x = 10$ and $y = 5$ in the first run. When the run terminates, the path constraint is of size 1, and $pathC[0] = init \wedge \phi \wedge \psi_d$, defined as follows. Note that the superscript 1 of the variables in $pathC[0]$ represents that it is the first BMC formula generated along the run. The symbolic variables $sym1$ and $sym2$ are created for x and y when $InitInput(\text{“}x\text{”})$ and $InitInput(\text{“}y\text{”})$ are called.

$$init \equiv x0^1 = sym1 \wedge y0^1 = sym2$$
$$\phi \equiv \begin{bmatrix} g^1_{BB1} = x0^1 > y0^1 \wedge \\ g^1_{BB2} = \neg(x0^1 > y0^1) \wedge \\ g^1_{BB4} = \left(g^1_{BB2} \wedge x1^1 > y0^1\right) \wedge \\ g^1_{BB5} = \left(g^1_{BB2} \wedge \neg(x1^1 > y0^1)\right) \wedge \\ g^1_{BB6} = \left(g^1_{BB4} \vee (g^1_{BB5} \wedge x2^1 \neq 0)\right) \end{bmatrix} \bigwedge \begin{bmatrix} x3^1 = ite(g^1_{BB1}, x2^1 - y0^1, x2^1) \wedge \\ x1^1 = ite(g^1_{BB2}, y0^1 - x0^1, x0^1) \wedge \\ x4^1 = ite(g^1_{BB4}, y0^1 - x3^1, x3^1) \wedge \\ x2^1 = ite(g^1_{BB5}, y0^1 - x1^1, x1^1) \wedge \\ y1^1 = ite(g^1_{BB6}, x4^1, y0^1) \end{bmatrix}$$
$$\psi_d \equiv g^1_{BB1} \vee \left(g^1_{BB6} \wedge y1^1 \neq 9\right)$$

The coverage history $branch\_hist$ is of size one. $branch\_hist[0]$ is a $BmcNode$ defined below:

$$branch\_hist[0].isCovered = [BB3 \mapsto true, BB7 \mapsto false] \qquad branch\_hist[0].done = false$$
$$\begin{aligned} branch\_hist[0].Edges\_d = [&BB3 \mapsto \{g^1_{BB1}, g^1_{BB6} \wedge y1^1 \neq 9\}, \\ &BB7 \mapsto \{g^1_{BB5} \wedge \neg(x2^1 \neq 0), g^1_{BB6} \wedge \neg(y1^1 \neq 9)\}] \end{aligned}$$

Now LLSPLAT searches for new inputs for the next run. Since $BB7$ is the only uncovered destination based on $branch\_hist[0].isCovered$, LLSPLAT solves the formula $init \wedge \phi \wedge \bigvee_{c \in branch\_hist[0].Edges\_d[BB7]} c$, that is, LLSPLAT tries to find a feasible execution path that leads to $BB7$ containing $ERROR$. Note that there are three execution paths to $BB7$, and the formula encodes all. LLSPLAT has a choice to produce inputs that follow any of them to $BB7$. Suppose that LLSPLAT generates a model m in which $m(sym1) = 0$ and $m(sym2) = 0$. LLSPLAT starts the second run by setting $x = 0$ and $y = 0$. The run follows the path $BB0, BB2, BB5, BB7$, and terminates. Since there is no uncovered destination, LLSPLAT terminates after the second run.

6.5 Experiments and Evaluation

We have developed a tool LLSPLAT[2] that implements the concolic+BMC algorithm. The evaluation of LLSPLAT is divided into two parts. In the first part, we compare LLSPLAT with two publicly available concolic testing tools, CREST [19] and KLEE [20]. CREST provides four search strategies: (1) an incremental depth-first search(`IDFS`), (2) a control-flow-guided search(`CFG`), (3) a uniform random search(`UR`), and (4) a random search selecting unexplored branches randomly(`RB`). Thus we need to compare against

[2] LLSPLAT can be downloaded at https://github.com/zilongwang/llsplat.

four versions of CREST. In the second part, we compare LLSPLAT with the state-of-the-art bounded model checker CBMC [73]. All experiments were performed on a 2 core Intel Xeon E5-2667 v2 CPU machine with 256GB memory and 64bit Linux.

6.5.1 Comparing LLSPLAT with CREST and KLEE

The goal of the experiments is to answer the following two research questions:

($Q1$) Given an *adequate* time budget, which tool has higher branch coverage?

($Q2$) Given a *limited* time budget, which tool has higher branch coverage?

Note that the fact that a tool A outperforms a tool B under an adequate time budget does not imply that A also outperforms B under a limited time budget. For example, it may happen that A starts outperforming B after a day of testing, but the testing budget is restricted to 10 minutes for each program under test. In this case, B is preferred to A.

We used the `ntdriver-simplified` and `ssh-simplified` benchmarks (36 programs in total) in SVCOMP15 [117] as the evaluation subjects. The benchmark sizes range over 218–2948 lines of code: 3 of the benchmarks have 200–700 lines, 29 of them have 701–2000 lines, and 4 of them have 2001–3000 lines. Since the coverage depends on the initial random input vector, we conducted the experiments 10 times and calculated the average coverage.

To answer the first question, we assume that one hour is an adequate time budget, and ran LLSPLAT, KLEE, and CREST with its four search strategies. Fig 6.4 presents the branch coverage results for each benchmark. We observe that LLSPLAT achieves the highest branch coverage for all benchmarks: it is on average 31%, 19%, 20%, 21% higher than `IDFS`, `CFG`, `UR`, and `RB`, respectively, and is on average 21% higher than KLEE. We also provide a histogram in Fig 6.3 which clusters benchmarks according to their branch coverage improvement. We observe LLSPLAT achieves 15%–35% higher branch coverage for most of the benchmarks than KLEE and CREST, which is close to the average improvement.

To answer the second question, we compute *the crossing time* from which on LLSPLAT *always* outperforms CREST and KLEE on the benchmarks. More concretely, for each benchmark, we analyzed a graph that described how branch coverage evolved

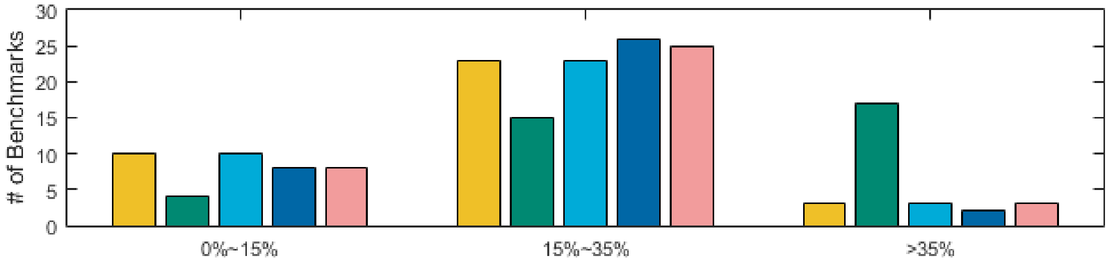


FIGURE 6.3: A histogram for branch coverage improvement of LLSPLAT. X-axis shows three ranges of branch coverage improvement. Each range has five bars, from left to right, corresponding to KLEE, CREST with four strategies `IDFS`, `CFG`, `UR`, and `RB`, respectively. Y-axis is the number of benchmarks.

in an hour using LLSPLAT, CREST, and KLEE, and recorded the time from which on the branch coverage reported by LLSPLAT is always higher than the one reported by CREST and KLEE. Fig 6.5 shows the results. The crossing time of 34 benchmarks is below 50s. The 23th benchmark is the worst (172s). Thus we conclude that when the testing time budget is limited, if it is not too limited (i.e., below 172s), LLSPLAT is still preferable.

6.5.2 Comparing LLSPLAT with CBMC

The goal of the experiments is to see if LLSPLAT can find bugs more quickly than CBMC. To achieve the goal, We used 13 sequentialized SystemC benchmarks [27]. They are known to be buggy, and we test them with LLSPLAT and CBMC to record the time to find the bugs. Table 6.2 shows the results. We observe that LLSPLAT finds bugs more quickly than CBMC in 11 benchmarks (except kundu1 and transmitter1), which indicates that our concolic+BMC approach is better than pure bounded model checking.

6.6 Related Work

Concolic Testing Several approaches analyze *states* (i.e., path constraint and symbolic store) maintained by concolic testing so as to explore the search space efficiently. Godefroid [50] introduced compositional concolic testing. The work was later expended to do compositional concolic testing on demand [2]. The main idea is to generate function summaries for an analyzed function based on the path constraint, and to reuse them if

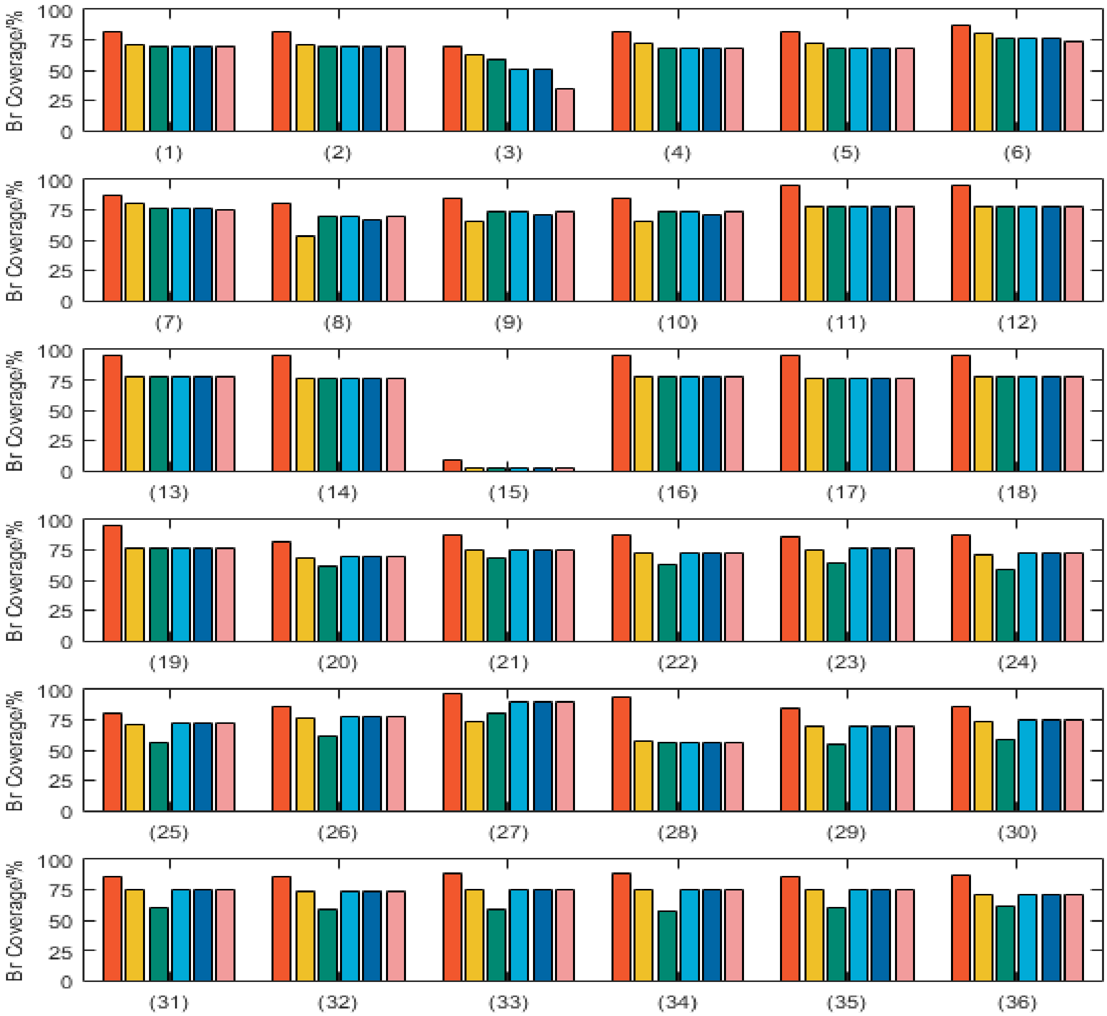


FIGURE 6.4: Branch coverage. X-axis denotes the benchmarks sorted by alphabetical order over their names. Y-axis is branch coverage. Each benchmark has six bars representing different tools. Bars from left to right correspond to LLSPLAT, KLEE, and CREST with four strategies `IDFS`, `CFG`, `UR`, and `RB`, respectively.

the function is called again with similar arguments. Instead of computing dynamic underapproximations of summaries, we compute exact summaries of governed regions using the static representation of the CFG. Kuznetsov et al. [74] introduced the dynamic state-merging (DSM) technique. DSM maintains a history queue of states. Two states may merge (depending on a separate and independent heuristic for SMT query difficulty) if they coincide in the history queue. Our concolic+BMC approach is different because we do not analyze the states to merge execution paths.

Moreover, several approaches combine other testing techniques with concolic testing together. Majumdar and Sen introduced hybrid concolic testing [87] that combines

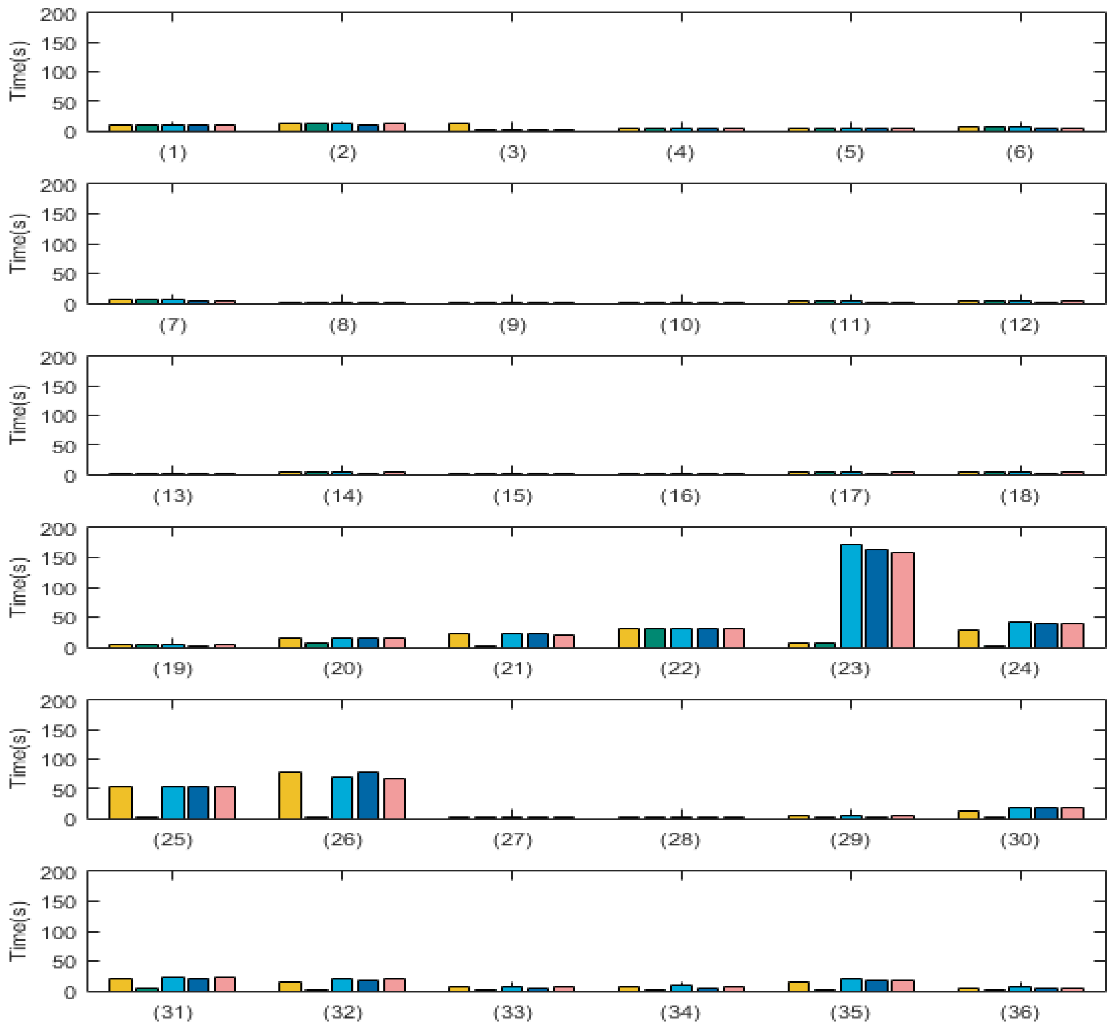


FIGURE 6.5: The crossing time from which on LLSPLAT outperforms KLEE and CREST. X-axis denotes the benchmarks sorted by alphabetical order over their names. Y-axis is the cross time. Each benchmark has five bars, from left to right, corresponding to KLEE and CREST with four strategies `IDFS`, `CFG`, `UR`, and `RB`, respectively.

random testing and concolic testing. Boonstoppel et al. proposed RWSet [11], a path pruning technique identifying redundant execution paths based on similarity of their live variables. Jaffar et al. [61] used interpolation to subsume execution paths that are guaranteed not to hit a buggy location. Avgerino et al. [5] combined static data-flow program analysis techniques with concolic testing. Santelices et al. [109] introduced a technique that merges multiple execution paths based on the control dependency graph of a program. To our best knowledge, we are the first who propose to combine bounded model checking techniques with concolic testing to alleviate path explosion.

A lot of work has focused on search heuristics to quickly guide execution to a

Benchmark	CBMC (s)	LLSPLAT (s)
kundu1	85	timeout
kundu2	419	0.013
pc_sfifo1	38	0.112
pc_sfifo2	27	0.420
pipeline	timeout	0.111
token_ring1	56	2
token_ring2	141	12
token_ring3	269	105
toy1	158	0.024
toy2	151	0.031
transmitter1	16	22
transmitter2	75	21
transmitter3	159	52

TABLE 6.2: Testing time for sequentialized SystemC benchmarks. Timeout is set to 30min.

specific branch [19, 20, 53, 81, 113, 121]. Search heuristics are orthogonal to our approach. Although we have implemented concolic+BMC with a naive bounded depth-first search, the algorithm can be used with other search strategies as well.

Bounded Model Checking VC generation approaches in modern BMC tools can be classified into two categories. The first one is based on *weakest preconditions* [36] by performing a demand-driven backward analysis from the points of interest [7, 24, 42, 77]. The other one encodes a program in a forward manner, such as CBMC [73], ESBMC [31], and LLBMC [92]. We are inspired by the VC generation algorithm of CBMC, and thus conceptually it is the closest work to our BMC algorithm. The VC generation of CBMC differs from ours in four ways. First, though CBMC also does variable renaming, it does it using a fixed order of basic blocks. We relax this requirement and prove that any topological order works for variable renaming. This is important to us, because we do not have to follow the fixed order CBMC uses. In fact, we use the reverse post order of a governed region as our topological order for variable renaming because it has been computed during the construction of depth first spanning tree which identifies back edges. We save the computation time in this way. Secondly, though the VC generation of CBMC also computes edge formulas for each basic block in a given acyclic program, *all predecessors* of the basic block contribute to deriving edge formulas. However, this is not the case in ours. For example, suppose that gov is governor, d is a destination of $GR(gov)$, and there is a predecessor $BB \notin GR(gov)$ of d. This case may happen because BB is polluted. Then our BMC algorithm does not derive an edge formula from BB for d. Thirdly, CBMC does not have the notion of destinations. Since a governed region may have multiple destinations, it is not clear that no matter which destination is chosen, whether the final version of variables in the formula ϕ that encodes the governed region always represents the value of the variables when the destination is reached. We prove this fact. Lastly, since CBMC encodes the entire program, it does not identify *acyclic portions of a program* using the notions such as governors. It also does function inlining and loop unrolling, which we do not.

ESBMC follows the VC generation algorithm of CBMC. It extends BMC to check concurrent programs. LLBMC explicitly models the memory as a variable representing an array of bytes, which requires LLBMC to distinguish if a little-endian or big-endian architecture is analyzed. They are orthogonal to LLSPLAT.

Software Model Checking Large-block encoding [9] is widely used in software model checkers. It encodes control flow edges into one formula, for computing the abstract successor during predicate abstraction. Selective enumeration using SAT solvers [56] and symbolic encodings for program regions, e.g., to summarize loops [72], have been successfully exploited in software model checking.

6.7 Proof Details

6.7.1 Preliminaries

Given a control flow graph (CFG) of a function, $BB_0, BB_1, \ldots, BB_n$ is a *path* of CFG if for each $0 \leq i \leq n-1$, (BB_i, BB_{i+1}) is an edge of CFG. Given an edge (a, b) of the CFG, we call a the *source* of the edge and b the *target* of the edge. A state s of a program is a function that maps each variable x in the program to a value in the domain of x. Given two states s and s', we denote $s \xrightarrow{BB} s'$ to be an execution such that by executing the instructions of BB with the initial state s, the execution ends up with the state s'. Occasionally, if we are not interested in s or s', we omit them and write $\xrightarrow{BB} s'$ or $s \xrightarrow{BB}$.

Given a formula ψ, an *assignment* m of ψ is a function that maps each variable x in ψ to a value in the domain of x. An assignment m is a *model* of ψ, denoted by $m \models \psi$, if ψ evaluates to *true* by m.

Given a set S of variables, a version map $\mathcal{V}$ is a *renaming* function that maps each variable $x \in S$ to a variable x_α for some $\alpha \in \mathbb{N}$. We write $\mathcal{V}(S)$ to be the set of variables $\{y \mid \exists x \in S.\ y = \mathcal{V}(x)\}$. Given a version map $\mathcal{V}$ and an assignment m to the variables in $\mathcal{V}(S)$, we denote $m_{|\mathcal{V}}$ to be an assignment to the variables in S such that for each variable $x \in S$, $m_{|\mathcal{V}}(x) = m(\mathcal{V}(x))$.

We first prove properties of effective dominance sets and governors. We then prove properties of our BMC algorithm.

6.7.2 Properties of Effective Dominance Sets and Governors

Lemma 28. *Given two basic blocks m and n, if $n \in Edom(m)$, then for each path from m to n, any basic block k along the path is not polluted.*

Proof. Suppose not. Then there is a path from m to n along which there is some k that is polluted. We consider two cases. Case 1: $m = n$. Then k must be n. Thus n is polluted and is not in $Edom(m)$. Contradiction. Case 2: $m \neq n$. Then $p : m \rightarrow^* k \rightarrow^+ n$. Since k is polluted and n is reachable by k, n is polluted and is not in $Edom(m)$. Contradiction. □

Lemma 29. *For any basic block m, $Edom(m)$ is acyclic.*

Proof. Suppose not. Then by Lemma 26, there is a basic block $n \in Edom(m)$ such that n is the source of a back edge. Hence n is polluted and is not in $Edom(m)$. Contradiction. □

Lemma 30. *Let m be a governor. The governed region $GR(m)$ is acyclic.*

Proof. Let $BB1$ and $BB2$ be the successors of m. By Lemma 29, $Edom(BB1)$ and $Edom(BB2)$ are acyclic. Moreover, there is not any edge $a \rightarrow b$ where $a \in Edom(BB1)$ and $b \in Edom(BB2)$. Otherwise, we can construct a path $m \rightarrow BB1 \rightarrow^* a \rightarrow b$ which bypasses $BB2$, which indicates that $BB2$ does not dominate b. Similarly, we can prove that there is not any edge $a \to b$ where $a \in Edom(BB2)$ and $b \in Edom(BB1)$. Thus, $GR(m)$ is acyclic. □

Lemma 31. *Let m be a governor. The governed region $GR(m)$ does not have any function calls.*

Proof. Let $BB1$ and $BB2$ be the successors of m. By Lemma 28, $Edom(BB1)$ and $Edom(BB2)$ do not have function calls. Since $GR(m) = Edom(BB1) \cup Edom(BB2)$, so does $GR(m)$. □

Lemma 32. *A governor m dominates every basic block n in its governed region $GR(m)$.*

Proof. By definition of $GR(m)$, we know that n is either dominated by $BB1$ or $BB2$ where $BB1$ and $BB2$ are the successors of m. Without loss of generality, let us assume that $BB1$ dominates n. Since m is a governor, m dominates $BB1$. Since dominance relation is transitive, m dominates n. □

6.7.3 Properties of the BMC Generation Algorithm

Given a governor gov and a basic block $BB \in GR(gov)$, note that if g_{BB} is true, then the block formula $Blks[BB]$ encodes the program logic of BB in SSA form, which leads to Lemma 33.

Lemma 33. *Given a program P and a governor gov, let $\mathcal{V}, \mathcal{V}'$ be the version map before and after the SSA variable renaming for BB. The following two statements hold: (1) If an assignment m with $m(g_{BB}) = true$ is a model of $Blks[BB]$, then $m_{|\mathcal{V}} \xrightarrow{BB} m_{|\mathcal{V}'}$ is an execution of P. (2) If $s \xrightarrow{BB} s'$ is an execution of P, then there is a model m of $Blks[BB]$ such that $m(g_{BB}) = true$, $m_{|\mathcal{V}} = s$, and $m_{|\mathcal{V}'} = s'$.*

Proof. Proof by induction on the number of instructions in BB. □

Given a governor gov, we prove that, for any destination $d \in Dests(gov)$, (1) the formula $\phi \wedge g_d$ where $g_d = \bigvee_{e \in Edges[d]} e$ represents all executions from gov to d, and (2) the final version of each variable $x \in AccVars(gov)$ in ϕ always represents the value of x when an execution from gov to d reaches d.

Lemma 34. *Given a governor gov, for any topological ordering T over $GR(gov)$ and any destination $d \in Dests(gov)$, if m is a model of $\phi \wedge g_d$, then (1) we can construct an execution ρ from the governor gov to the destination d, and (2) for each variable $x \in AccVars(gov)$, if x_α is the final version of x in ϕ, then $m(x_\alpha)$ is the value of x when the execution ρ enters the destination d.*

Proof. Since m is a model of $\phi \wedge g_d$, let the set $Taken$ be $\{BB \mid m(g_{BB}) = true\}$, i.e., the set of all basic blocks whose guard g_{BB} is set to true by m. Since g_d is $true$, we know that the guard of a predecessor of d holds, the guard of a predecessor of the predecessor of d holds, and so on. This indicates that there is a path from gov to d along which the guards g_{BB} of all basic blocks $BB \in GR(gov)$ are set to $true$ by m. Moreover, the guards g_{BB} of all basic blocks $BB \in GR(gov)$ that are not shown along the path are set to *false* by m since the guards of two successors cannot hold at the same time. Hence we know that the set $Taken$ are the intermediate basic blocks of the path from gov to d.

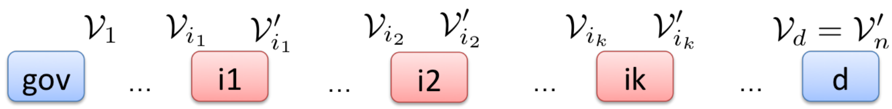


FIGURE 6.6: Topological ordering of the governed region

As shown in Fig 6.6, let $BB_1, \ldots, BB_{i_1}, \ldots, BB_{i_2}, \ldots, BB_{i_k}, \ldots BB_n$ be the sequence of basic blocks in $GR(gov)$ sorted by the topological ordering T such that each $BB_{i_j} \in Taken$ where $1 \leq j \leq k$. Note that $gov, BB_{i_1}, BB_{i_2}, \ldots, BB_{i_k}, d$ is a path. Otherwise, T is not a topological ordering. We now construct an execution along this path. For each $BB_i \in GR(gov)$ where $1 \leq i \leq n$, let $\mathcal{V}_i$ be the version map before BB_i and $\mathcal{V}'_i$ be the one after BB_i. By Lemma 33, we know that for each $1 \leq j \leq k$, $m_{|\mathcal{V}_{i_j}} \xrightarrow{BB_{i_j}} m_{|\mathcal{V}'_{i_j}}$ is an execution. Also, since the guards g_{BB} of all basic block $BB \notin Taken$ are set to *false* by the model m, we have

$$m_{|\mathcal{V}_1} = m_{|\mathcal{V}_{i_1}}, \quad m_{|\mathcal{V}'_{i_1}} = m_{|\mathcal{V}_{i_2}}, \quad \ldots, \quad m_{|\mathcal{V}'_{i_{k-1}}} = m_{|\mathcal{V}_{i_k}}, \quad m_{|\mathcal{V}'_{i_k}} = m_{|\mathcal{V}'_n}$$

Hence, $\xrightarrow{gov} m_{|\mathcal{V}_{i_1}} \xrightarrow{BB_{i_1}} m_{|\mathcal{V}_{i_2}} \xrightarrow{BB_{i_2}} \ldots \xrightarrow{BB_{i_{k-1}}} m_{|\mathcal{V}_{i_k}} \xrightarrow{BB_{i_k}} m_{|\mathcal{V}'_n} \xrightarrow{d}$ is an execution of the program P. Moreover, since $m_{|\mathcal{V}'_n} \xrightarrow{d}$, the final version of each variable x in the model m represents the value of x when the execution enters the destination d. $\square$

Lemma 35. *Given a governor gov, for any topological ordering T over $GR(gov)$ and any destination $d \in Dests(gov)$, if there is an execution from gov to d, then we can construct a model m for the formula $\phi \wedge g_d$.*

Proof. Let $\xrightarrow{gov} s_0 \xrightarrow{BB_{i1}} s_1 \xrightarrow{BB_{i2}} s_2 \ldots \xrightarrow{BB_{i_k}} s_k \xrightarrow{d}$ be an execution from the governor gov to a destination d, as shown in Fig 6.7.

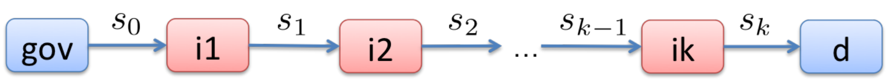


FIGURE 6.7: An execution from the governor gov to a destination d

Let $BB_1, BB_2, \ldots, BB_n$ be the sequence of basic blocks in $GR(gov)$ sorted by the topological ordering T. Note that for $2 \leq j \leq k$, $BB_{i_{j-1}}$ must occur before BB_{i_j} along the sequence. Otherwise, T is not a topological ordering. We present this fact in Fig 6.8.

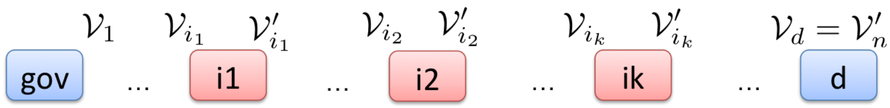


FIGURE 6.8: Topological ordering of the governed region

For each $BB_i \in GR(gov)$ where $1 \leq i \leq n$, let $\mathcal{V}_i$ be the version map before BB_i and $\mathcal{V}'_i$ be the one after BB_i. We now construct an assignment m and prove that m is a model of $\phi \wedge g_d$.

Let $Taken$ be the set $\{BB_{i_j} \mid 1 \leq j \leq k\}$. For each basic block $BB \in GR(gov)$, if $BB \in Taken$, then we set $m(g_{BB}) = \mathit{true}$, $m(g_{BB}) = \mathit{false}$ otherwise. For each variable $x \in AccVars(gov)$, we construct the assignment m in four steps:

(1) If $\mathcal{V}_1(x) = x_l$ and $\mathcal{V}_{i_1}(x) = x_h$, then for each x_α with $l \leq \alpha \leq h$, $m(x_\alpha) = s_0(x)$;

(2) For each $j \in [1, k-1]$, if $\mathcal{V}'_{i_j}(x) = x_l$ and $\mathcal{V}_{i_{j+1}}(x) = x_h$, then for each x_α with $l < \alpha \leq h$, $m(x_\alpha) = s_j(x)$,

(3) If $\mathcal{V}'_{i_k}(x) = x_l$ and $\mathcal{V}'_n(x) = x_h$, then for each x_α with $l < \alpha \leq h$, $m(x_\alpha) = s_k(x)$;

(4) For each $j \in [1, k]$, by Lemma 33, we know that there is a model m_j of $Blks[BB_{i_j}]$ such that $m_j(g_{BB_{i_j}}) = \mathit{true}$, $m_{j|V_{i_j}} = s_{j-1}$ and $m_{j|V'_{i_j}} = s_j$. If $\mathcal{V}_{i_j}(x) = x_l$ and $\mathcal{V}'_{i_j}(x) = x_h$, then for each x_α with $l < \alpha \leq h$, $m(x_\alpha) = m_j(x)$.

Note that each variable x_α is assigned exactly once in the above construction of m, which means m does not make different values to x_α. Now we show m is indeed a model of $\phi \wedge g_d$. We consider two cases depending on whether a basic block $BB \in GR(gov)$ is in the set $Taken$.

1. Suppose $BB \in Taken$. Then BB is BB_{i_j} for some $j \in [1,k]$. First, $m \models Blks[BB_{i_j}]$ according to Step 4 of the above construction. Secondly, m evaluates $g_{BB_{i_j}}$ to true. Lastly, m evaluates $\bigvee_{c \in Edges[BB_{i_j}]} c$ to true by proving the following cases.
   (a) BB_{i_j} is the left successor of the governor gov. Let $br\ e\ BB_{i_j}\ BB2$ be the terminating instruction of gov. Since $s_0 \models e$ and $m_{|\mathcal{V}_1} = s_0$, we have $m \models \mathcal{V}_1(e)$, and thus $m \models \bigvee_{c \in Edges[BB_{i_j}]} c$.
   (b) BB_{i_j} is the right successor of the governor gov. This case is proved similarly as case (a).
   (c) BB_{i_j} is the unique successor of the basic block $BB_{i_{j-1}} \in Taken$. Since $m(g_{BB_{i_{j-1}}}) = true$ and $g_{BB_{i_{j-1}}} \in Edges[BB_{i_j}]$, we have that $m \models \bigvee_{c \in Edges[BB_{i_j}]} c$.
   (d) BB_{i_j} is the left successor of $BB_{i_{j-1}} \in Taken$. Let $br\ e\ BB_{i_j}\ BB2$ be the terminating instruction of $BB_{i_{j-1}}$. Since $s_{j-1} \models e$ and $m_{|\mathcal{V}'_{i_{j-1}}} = s_{j-1}$, we have $m \models \mathcal{V}'_{i_{j-1}}(e)$. Moreover, since $m(g_{BB_{i_j-1}}) = true$, then $m \models g_{BB_{i_j-1}} \wedge \mathcal{V}'_{i_{j-1}}(e)$. Hence $m \models \bigvee_{c \in Edges[BB_{i_j}]} c$.
   (e) BB_{i_j} is the right successor of $BB_{i_{j-1}} \in Taken$. This case is proved similarly as case (d).

2. Suppose $BB \notin Taken$. First, since $m(g_{BB}) = false$, $Blks[BB]$ are conjunctions of equations of the form $x_\alpha = x_{\alpha-1}$. By Steps (1),(2), and (3), we have $m \models Blks[BB]$. Secondly, we prove that m evaluates $\bigvee_{c \in Edges[BB]} c$ to false by contradiction. Suppose there is $c \in \bigvee_{e \in Edges[BB]} e$ such that $m \models c$. We consider the following cases depending on the form of c.

(a) $c \equiv \mathcal{V}_1(e)$. Then BB is the left successor of the governor gov. Let $br\ e\ BB\ BB_{i_1}$ be the terminating instruction of gov. Since $s_0 \models \neg e$ and $m_{|\mathcal{V}_1} = s_0$, we have $m \models \neg\mathcal{V}_1(e)$, and thus $m \not\models c$. Contradiction.

(b) $c \equiv \neg\mathcal{V}_1(e)$. Then BB is the right successor of the governor gov. This case is proved similarly as case (a).

(c) $c \equiv g_{BB'}$. Then we know that $BB' \in Taken$ and BB is the unique successor of BB'. Since $m(g_{BB'}) = true$, then $m(g_{BB}) = true$. Contradiction.

(d) $c \equiv g_{BB_u} \wedge \mathcal{V}'_u(e)$ for some $u \in [1, n]$. Since $m \models g_{BB_u}$, we know that $BB_u \in Taken$ and BB is the left successor of BB_u. Without loss of generality, let BB_u be BB_{i_j} for some $j \in [1, k]$. Then $\mathcal{V}'_u = \mathcal{V}'_{i_j}$. Let $br\ e\ BB\ BB'$ be the terminating instruction of BB_{i_j}. Note that $s_j \models \neg e$. Since $m_{|\mathcal{V}'_{i_j}} = s_j$, we have $m \models \neg\mathcal{V}'_{i_j}(e)$. Contradiction.

(e) $c \equiv g_{BB_u} \wedge \neg\mathcal{V}'_u(e)$ for some $u \in [1, n]$. Since $m \models g_{BB_u}$, we know that $BB_u \in Taken$ and BB is the right successor of BB_u. This case is proved similarly as case (d).

Now that we have proved for each $BB \in GR(gov)$, $m \models g_{BB} = \bigvee_{c \in Edges[BB]} c$ and $m \models Blks[BB]$. Thus $m \models \phi$. We now prove that $m \models g_d$ where $g_d = \bigvee_{c \in Edges[d]} c$. Suppose that the destination d is the unique successor of BB_{i_k}, then $g_{BB_{i_k}} \in Edges[d]$. Since $m \models g_{BB_{i_k}}$, $m \models g_d$. Suppose that d is the left successor of BB_{i_k}, that is, the terminating instruction of BB_{i_k} is $br\ e\ d\ BB2$. Since $s_k \models e$ and $m_{|\mathcal{V}'_{i_k}} = s_k$, we have $m \models \mathcal{V}'_{i_k}(e)$. Since $g_{BB_{i_k}} \wedge \mathcal{V}'_{i_k}(e) \in Edges[d]$, we have $m \models g_d$. By the similar reasoning, if d is the right successor of BB_{i_k}, we also have $m \models g_d$. Hence $m \models \phi \wedge g_d$. □

Theorem 21. *Given a governor gov and a destination $d \in Dests(gov)$, the formula $\phi \wedge g_d$ encodes all executions from gov to d. Moreover, the final version of each variable x in ϕ represents the value of x when an execution from gov to d enters d.*

Proof. Proved by Lemma 34 and 35. □

Bibliography

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. “General decidability theorems for infinite-state systems”. In: *LICS ’96*. IEEE, 1996, pp. 313–321.

[2] S. Anand, P. Godefroid, and N. Tillmann. “Demand-driven Compositional Symbolic Execution”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 367–381.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation”. In: *J. Syst. Softw.* 86.8 (Aug. 2013).

[4] M. F. Atig, A. Bouajjani, and S. Qadeer. “Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads”. In: *TACAS ’09: Proc. 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 5505. LNCS. Springer, 2009, pp. 107–123.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. “Enhancing Symbolic Execution with Veritesting”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 1083–1094.

[6] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. “VeriCon: Towards Verifying Controller Programs in Software-defined Networks”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 282–293.

[7] M. Barnett and K. R. M. Leino. “Weakest-precondition of Unstructured Programs”. In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’05. New York, NY, USA: ACM, 2005.

[8] A. Barth, J. Mitchell, A. Datta, and S. Sundaram. “Privacy and Utility in Business Processes”. In: *CSF*. IEEE, 2007, pp. 279–294.

[9] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. “Software Model Checking via Large-Block Encoding”. In: *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009, Austin, TX, November 15-18)*. 2009, pp. 25–32.

[10] B. Bingham, J. Bingham, F. de Paula, J. Erickson, G. Singh, and M. Reitblatt. “Industrial Strength Distributed Explicit State Model Checking”. In: *PDMC-HIBI*. 2010, pp. 28–36.

[11] P. Boonstoppel, C. Cadar, and D. Engler. “RWset: Attacking Path Explosion in Constraint-based Test Generation”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary, 2008.

[12] A. Bouajjani and M. Emmi. “Analysis of recursively parallel programs”. In: *POPL*. 2012, pp. 203–214.

[13] A. Bouajjani and M. Emmi. “Bounded phase analysis of message-passing programs”. In: *STTT* 16.2 (2014), pp. 127–146.

[14] A. Bouajjani, S. Qadeer, and S. Fratani. “Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures”. In: *CAV’07: Proc. 19th Int. Conf. on Computer Aided Verification*. Vol. 4590. LNCS. Springer, 2007, pp. 207–220.

[15] L. Bozzelli and P. Ganty. “Complexity Analysis of the Backward Coverability Algorithm for VASS”. In: *RP 11*. LNCS 6945. Springer, 2011, pp. 96–109.

[16] D. Brand and P. Zafiropulo. “On Communicating Finite-State Machines”. In: *J. ACM* 30.2 (Apr. 1983), pp. 323–342.

[17] D. Bucur and M. Z. Kwiatkowska. “Software Verification for TinyOS”. In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN ’10. Stockholm, Sweden: ACM, 2010, pp. 400–401.

[18] P. Buneman, S. Khanna, and W.-C. Tan. “Why and where: A characterization of data provenance”. In: *ICDT*. LNCS 1973. Springer, 2001, pp. 316–330.

[19] J. Burnim and K. Sen. “Heuristics for Scalable Dynamic Test Generation”. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’08. Washington, DC, USA, 2008, pp. 443–446.

[20] C. Cadar, D. Dunbar, and D. Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[21] C. Cadar and K. Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Commun. ACM* 56.2 (Feb. 2013), pp. 82–90.

[22] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. “A NICE Way to Test Openflow Applications”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA, 2012, pp. 127–140.

[23] S. Chaki, S. Rajamani, and J. Rehof. “Types as models: model checking message-passing programs”. In: *POPL*. ACM, 2002, pp. 45–57.

[24] S. Chandra, S. J. Fink, and M. Sridharan. “Snugglebug: A Powerful Approach to Weakest Preconditions”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: ACM, 2009.

[25] T. Chen, X.-S. Zhang, S.-Z. Guo, H.-Y. Li, and Y. Wu. “State of the Art: Dynamic Symbolic Execution for Automated Test Generation”. In: *Future Gener. Comput. Syst.* 29.7 (Sept. 2013), pp. 1758–1773.

[26] J. Cheney, A. Ahmed, and U. Acar. “Provenance as dependency analysis”. In: *Math. Struct. in Computer Science* 21 (2011), pp. 1301–1337.

[27] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. “Verifying SystemC: A Software Model Checking Approach”. In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD ’10. Lugano, Switzerland: FMCAD Inc, 2010, pp. 51–60.

[28] E. Clarke, D. Kroening, and F. Lerda. “A Tool for Checking ANSI-C Programs”. English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by K. Jensen and A. Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 168–176.

[29] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. “Bounded Model Checking Using Satisfiability Solving”. In: *Form. Methods Syst. Des.* 19.1 (July 2001), pp. 7–34.

[30] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. “Efficient Memory Safety for TinyOS”. In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*. SenSys ’07. Sydney, Australia: ACM, 2007, pp. 205–218.

[31] L. Cordeiro, J. Morse, D. Nicole, and B. Fischer. “Context-Bounded Model Checking with ESBMC 1.17”. English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2012, pp. 534–537.

[32] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001.

[33] Y. Cui, J. Widom, and J. Wiener. “Tracing the lineage of view data in a warehousing environment”. In: *ACM TODS* 25 (2000), pp. 179–227.

[34] R Cunningham. “Eel: Tools for debugging, visualization, and verification of event-driven software”. MA thesis. UCLA, 2005.

[35] S. Demri, M. Jurdzinski, O. Lachish, and R. Lazic. “The covering and boundedness problems for branching vector addition systems”. In: *J. Comput. Syst. Sci.* 79.1 (2013), pp. 23–38.

[36] E. W. Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[37] D. L. Dill. "The Murphi Verification System". In: *Proceedings of the 8th International Conference on Computer Aided Verification*. CAV '96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393.

[38] B. Durak. "JSure". Available at https://github.com/berke/jsure.

[39] M. Emmi, A. Lal, and S. Qadeer. "Asynchronous Programs with Prioritized Task-buffers". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. New York, NY, USA: ACM, 2012, 48:1–48:11.

[40] N. Feamster, J. Rexford, and E. Zegura. "The Road to SDN". In: *Queue* 11.12 (Dec. 2013), 20:20–20:40.

[41] A. Finkel and P. Schnoebelen. "Well-structured transition systems everywhere!" In: *Theoretical Computer Science* 256.1-2 (2001), pp. 63–92.

[42] C. Flanagan and J. B. Saxe. "Avoiding Exponential Explosion: Generating Compact Verification Conditions". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: ACM, 2001, pp. 193–205.

[43] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. "Frenetic: A Network Programming Language". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan: ACM, 2011, pp. 279–291.

[44] P. Ganty, J.-F. Raskin, and L. V. Begin. "From Many Places to Few: Automatic Abstraction Refinement for Petri Nets". In: *Fund. Informaticae* 88(3) (2008), pp. 275–305.

[45] P. Ganty and R. Majumdar. "Algorithmic Verification of Asynchronous Programs". Submitted for publication (TOPLAS). 2011.

[46] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC Language: A Holistic Approach to Networked Embedded Systems". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, California, USA: ACM, 2003, pp. 1–11.

[47] G. Geeraerts, J.-F. Raskin, and L. Van Begin. “Expand, Enlarge and Check: new algorithms for the coverability problem of WSTS”. In: *FSTTCS ’04*. LNCS 3328. Springer, 2004, pp. 287–298.

[48] GENI Assignment. *GENI Assignment*. http://groups.geni.net/geni/wiki/GENIEducation/SampleAssignments/OpenFlowFirewallAssignment/ExerciseLayout/Execute.

[49] N. Ghafari, A. Hu, and Z. Rakamarić. “Context-Bounded Translations for Concurrent Software: An Empirical Evaluation”. English. In: *Model Checking Software*. Ed. by J. van de Pol and M. Weber. Vol. 6349. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 227–244.

[50] P. Godefroid. “Compositional Dynamic Test Generation”. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’07. Nice, France: ACM, 2007, pp. 47–54.

[51] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer, 1996.

[52] P. Godefroid, N. Klarlund, and K. Sen. “DART: Directed Automated Random Testing”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. New York, NY, USA, 2005, pp. 213–223.

[53] P. Godefroid, M. Y. Levin, and D. Molnar. “SAGE: Whitebox Fuzzing for Security Testing”. In: *Queue* 10.1 (Jan. 2012), 20:20–20:27.

[54] T. Green, G. Karvounarakis, and V. Tannen. “Provenance semirings”. In: *PODS*. ACM, 2007, pp. 31–40.

[55] A. Guha, M. Reitblatt, and N. Foster. “Machine-verified Network Controllers”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: ACM, 2013, pp. 483–494.

[56] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. “Program analysis via satisfiability modulo path programs”. In: *POPL 2010*. ACM, 2010, pp. 71–82.

[57] G. Higman. “Ordering by divisibility in abstract algebras”. In: *Proc. London Math. Soc.* (3) 2 (1952), pp. 326–336.

[58] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. “System Architecture Directions for Networked Sensors”. In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA: ACM, 2000, pp. 93–104.

[59] G. Holzmann. “The Spin Model Checker”. In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295.

[60] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. “Securing Web application code by static analysis and runtime protection”. In: *WWW*. 2004, pp. 40–52.

[61] J. Jaffar, V. Murali, and J. A. Navas. “Boosting Concolic Testing via Interpolation”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 48–58.

[62] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. “B4: Experience with a Globally-deployed Software Defined Wan”. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM13. Hong Kong, China, 2013, pp. 3–14.

[63] J. Janák. “Issue Tracking Systems”. Diplomová práce. Masarykova univerzita, Fakulta informatiky, 2009.

[64] R. Jhala and R. Majumdar. “Interprocedural Analysis of Asynchronous Programs”. In: *POPL '07*. ACM, 2007, pp. 339–350.

[65] A. Kaiser, D. Kroening, and T. Wahl. “Efficient coverability analysis by proof minimization”. In: *CONCUR 2012*. LNCS 7454. Springer, 2012, pp. 500–515.

[66] A. Kaiser, D. Kroening, and T. Wahl. “Dynamic Cutoff Detection in Parameterized Concurrent Programs”. In: *Proceedings of the 22Nd International Conference on*

Computer Aided Verification. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 645–659.

[67] A. Kaiser, D. Kroening, and T. Wahl. "Efficient Coverability Analysis by Proof Minimization". In: *Proceedings of the 23rd International Conference on Concurrency Theory*. CONCUR'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 500–515.

[68] R. Karp and R. Miller. "Parallel Program Schemata." In: *Journal of Comput. Syst. Sci.* 3.2 (1969), pp. 147–195.

[69] P. Kazemian, G. Varghese, and N. McKeown. "Header Space Analysis: Static Checking for Networks". In: *NSDI*. 2012, pp. 113–126.

[70] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. "Incremental, inductive coverability". In: *CAV 2013*. LNCS. Springer, 2013.

[71] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. "The Click Modular Router". In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297.

[72] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger. "Loop summarization using state and transition invariants". In: *Formal Methods in System Design* 42.3 (2013), pp. 221–261.

[73] D. Kroening, E. Clarke, and K. Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking". In: *Proceedings of DAC 2003*. ACM Press, 2003, pp. 368–371.

[74] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. "Efficient State Merging in Symbolic Execution". In: *SIGPLAN Not.* 47.6 (June 2012), pp. 193–204.

[75] S. La Torre, P. Madhusudan, and G. Parlato. "Model-Checking Parameterized Concurrent Programs Using Linear Interfaces". English. In: *Computer Aided Verification*. Ed. by T. Touili, B. Cook, and P. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 629–644.

[76] S. K. Lahiri, S. Qadeer, and Z. Rakamarić. "Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers". In: *Proceedings of the 21st International Conference on Computer Aided Verification*. CAV '09. Grenoble, France: Springer-Verlag, 2009, pp. 509–524.

[77] A. Lal, S. Qadeer, and S. K. Lahiri. "A Solver for Reachability Modulo Theories". In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 427–443.

[78] A. Lal and T. Reps. "Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis". In: *CAV '08: Proc. 20th Int. Conf. on Computer Aided Verification*. Vol. 5128. LNCS. Springer, 2008, pp. 37–51.

[79] A. Lal, T. Touili, N. Kidd, and T. W. Reps. "Interprocedural Analysis of Concurrent Programs Under a Context Bound". In: *TACAS '08: Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. LNCS. Springer, 2008, pp. 282–298.

[80] P. Li and J. Regehr. "T-check: Bug Finding for Sensor Networks". In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '10. Stockholm, Sweden: ACM, 2010, pp. 174–185.

[81] Y. Li, Z. Su, L. Wang, and X. Li. "Steering Symbolic Execution to Less Traveled Paths". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 19–32.

[82] R. Lipton. *The reachability problem is exponential-space hard*. Tech. rep. 62. Department of Computer Science, Yale University, 1976.

[83] P. Liu and T. Wahl. "Infinite-State Backward Exploration of Boolean Broadcast Programs". In: *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*. FMCAD '14. Austin, TX: FMCAD Inc, 2014, 26:155–26:162.

[84] B. Livshits and M. Lam. "Finding security errors in Java programs with static analysis". In: *Usenix Security Symposium*. 2005, pp. 271–286.

[85] I. Lomazova and P. Schnoebelen. "Some Decidability Results for Nested Petri Nets". In: *Ershov Memorial Conference*. LNCS 1755. Springer, 2000, pp. 208–220.

[86] R. Majumdar, R. Meyer, and Z. Wang. "Static Provenance Verification for Message Passing Programs". In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 2013, pp. 366–387.

[87] R. Majumdar and K. Sen. "Hybrid Concolic Testing". In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426.

[88] R. Majumdar, S. D. Tetali, and Z. Wang. "Kuai: A Model Checker for Software-defined Networks". In: *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*. FMCAD '14. Lausanne, Switzerland: FMCAD Inc, 2014, 27:163–27:170.

[89] R. Majumdar and Z. Wang. "BBS: A Phase-Bounded Model Checker for Asynchronous Programs". English. In: *Computer Aided Verification*. Ed. by D. Kroening and P. C. S. Vol. 9206. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 496–503.

[90] R. Majumdar and Z. Wang. "Expand, Enlarge, and Check for Branching Vector Addition Systems". In: *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. 2013, pp. 152–166.

[91] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM* 38.2 (Mar. 2008), pp. 69–74.

[92] F. Merz, S. Falke, and C. Sinz. "LLBMC: Bounded Model Checking of C and C++; Programs Using a Compiler IR". In: *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE'12. Philadelphia, PA: Springer-Verlag, 2012, pp. 146–161.

[93] R. Meyer and T. Strazny. "Petruchio: From Dynamic Networks to Nets". In: *CAV*. LNCS 6174. Springer, 2010, pp. 175–179.

[94] M. Miller, E. Tribble, and J. Shapiro. "Concurrency Among Strangers". English. In: *Trustworthy Global Computing*. Ed. by R. De Nicola and D. Sangiorgi. Vol. 3705. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 195–229.

[95] M. Minsky. *Finite and Infinite Machines*. Prentice-Hall, 1967.

[96] M. Musuvathi and S. Qadeer. "Iterative Context Bounding for Systematic Testing of Multithreaded Programs". In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 446–455.

[97] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. "Resonance: Dynamic Access Control for Enterprise Networks". In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. WREN '09. Barcelona, Spain: ACM, 2009, pp. 11–18.

[98] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. "A Balance of Power: Expressive, Analyzable Controller Programming". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: ACM, 2013, pp. 79–84.

[99] V. S. Pai, P. Druschel, and W. Zwaenepoel. "Flash: An Efficient and Portable Web Server". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '99. Monterey, California: USENIX Association, 1999, pp. 15–15.

[100] C. A. Petri. "Kommunikation mit Automaten". PhD thesis. Technical University Darmstadt, 1962.

[101] A. Pnueli, J. Xu, and L. Zuck. "Liveness with $(0, 1, \infty)$-Counter Abstraction". In: *CAV*. LNCS 2404. Springer, 2002, pp. 107–122.

[102] POX. http://www.noxrepo.org/pox/about-pox/.

[103] S. Qadeer. "The Case for Context-Bounded Verification of Concurrent Programs". In: *SPIN '08: Proc. of 15th Int. Model Checking Software Workshop*. Vol. 5156. LNCS. Springer, 2008, pp. 3–6.

[104] S. Qadeer and J. Rehof. "Context-Bounded Model Checking of Concurrent Software". In: *TACAS '05: Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3440. LNCS. Springer, 2005, pp. 93–107.

[105] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. "SIMPLE-fying Middlebox Policy Enforcement Using SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM13. Hong Kong, China: ACM, 2013, pp. 27–38.

[106] C. Rackoff. "The covering and boundedness problems for vector addition systems". In: *Theoretical Computer Science* 6.2 (1978), pp. 223–231.

[107] A. Sabelfeld and A. Myers. "Language-based information-flow security". In: *IEEE J. Selected Areas in Communications* 21 (2003), pp. 5–19.

[108] Safe TinyOS. http://docs.tinyos.net/index.php/Safe_TinyOS.

[109] R. Santelices and M. J. Harrold. "Exploiting Program Dependencies for Scalable Multiple-path Symbolic Execution". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy, 2010.

[110] P. Schnoebelen. "Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets". In: *MFCS*. LNCS 6281. Springer, 2010, pp. 616–628.

[111] K. Sen, D. Marinov, and G. Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[112] K. Sen and M. Viswanathan. "Model Checking Multithreaded Programs with Asynchronous Atomic Methods". In: *CAV '06: Proc. 18th Int. Conf. on Computer Aided Verification*. Vol. 4144. LNCS. Springer, 2006, pp. 300–314.

[113] H. Seo and S. Kim. "How We Get There: A Context-guided Search Strategy in Concolic Testing". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014.

[114] D. Sethi, S. Narayana, and S. Malik. "Abstractions for model checking SDN controllers". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. 2013, pp. 145–148.

[115] I. Souilah, A. Francalanza, and V. Sassone. “A Formal Model of Provenance in Distributed Systems”. In: *Workshop on the Theory and Practice of Provenance*. 2009.

[116] D. Suwimonteerabuth, J. Esparza, and S. Schwoon. “Symbolic Context-Bounded Analysis of Multithreaded Java Programs”. In: *SPIN '08: Proc. of 15th Int. Model Checking Software Workshop*. Vol. 5156. LNCS. Springer, 2008, pp. 270–287.

[117] SVCOMP15. *Competition on Software Verification*. https://github.com/dbeyer/sv-benchmarks/tree/master/c/.

[118] S. La Torre, G. Parlato, and P. Madhusudan. “Reducing Context-Bounded Concurrent Reachability to Sequential Reachability”. In: *CAV '09: Proc. 21st Int. Conf. on Computer Aided Verification*. Vol. 5643. LNCS. Springer, 2009, pp. 477–492.

[119] K. Verma and J. Goubault-Larrecq. “Karp-Miller Trees for a Branching Extension of VASS”. In: *Discrete Mathematics & Theoretical Computer Science* 7.1 (2005), pp. 217–230.

[120] K. N. Verma and J. Goubault-Larrecq. “Alternating two-way AC-tree automata”. In: *Inf. Comput.* 205.6 (2007), pp. 817–869.

[121] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. “Fitness-guided path exploration in dynamic symbolic execution”. In: *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. 2009, pp. 359–368.

Zilong Wang's CV

Education

2010–Present	PhD Candidate, MPI-SWS, Kaiserslautern, Germany
2006–2010	B. Tech, Computer Science, Nankai University, Tianjin, China

Publications

R. Majumdar, I. Saha, and Z. Wang. *Systematic Testing for Control Applications*, MEMOCODE 2010.

R. Majumdar, I. Saha, K. Shashidhar, and Z. Wang. *CLSE: Closed-Loop Symbolic Executions*, NFM 2012.

R. Majumdar, R. Meyer, and Z. Wang. *Static Provenance Verification for Message Passing Programs*, SAS 2013.

R. Majumdar, R. Meyer, and Z. Wang. *Provenance Verification*, RP 2013.

R. Majumdar and Z. Wang. *Expand, Enlarge, and Check for Branching Vector Addition Systems*, CONCUR 2013.

R. Majumdar, S. Tetali, and Z. Wang. *Kuai: A Model Checker for Software-defined Networks*, FMCAD 2014.

R. Majumdar and Z. Wang. *BBS: A Phase-Bounded Model Checker for Asynchronous Programs*, CAV 2015.

Teaching Assistant Experience

Fall 2012	**Verification of Reactive Systems**, TU Kaiserslautern, Germany. Responsible for 2 hours per week of tutorial lecture, supervision of projects, preparation of sample answers of homework and final tests, proctoring final tests and grading the tests.

Internship

Spring 2014	**Microsoft Research India**, Bangalore, India. Work with Akash Lal on the project: Houdini Candidate Inference for Static Driver Verifier.
Spring 2010	**University of California, Los Angeles**, Los Angeles, USA. Work with Rupak Majumdar on the project: Static Range Analysis for Floating-point C Programs.
Summer 2009	**University of California, Los Angeles**, Los Angeles, USA. Work with Rupak Majumdar on the project: A Constraint-based Automatic Test Case Generator for C Programs.

Awards and Honors

2009	Cross-disciplinary Scholarship in Science and Technology at UCLA.
2009	TEDA & Vestas Scholarship at Nankai University.
2008	National Scholarship at Nankai University.
2007	Outstanding Student Scholarship (1st class) at Nankai University.