Formal Semantics and Scalable Verification for the Border Gateway Protocol using Proof Assistants and SMT Solvers

Konstantin Weitz

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Michael D. Ernst, Chair

Zachary Tatlock, Chair

Arvind Krishnamurthy

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Formal Semantics and Scalable Verification
for the Border Gateway Protocol
using Proof Assistants and SMT Solvers

Konstantin Weitz

Co-Chairs of the Supervisory Committee:
Professor Michael D. Ernst
Computer Science & Engineering

Professor Zachary Tatlock
Computer Science & Engineering

To reliably and securely route traffic across the Internet, Internet Service Providers (ISPs) must configure their Border Gateway Protocol (BGP) routers to implement policies restricting how routing information can be exchanged with other ISPs. Correctly implementing these policies in low-level router configuration languages, with configuration code distributed across all of an ISP's routers, has proven challenging in practice, and misconfiguration has led to extended worldwide outages and traffic hijacks.

We present Bagpipe, the first system that enables ISPs to declaratively specify control-plane policies and that automatically verifies that router configurations implement such policies using an SMT solver.

We evaluated the expressiveness of Bagpipe's policy specification language on 10 configuration scenarios from the Juniper TechLibrary, and evaluated the efficiency of Bagpipe on three ISPs with a total of over 240,000 lines of Cisco and Juniper BGP configuration. Bagpipe revealed 19 policy violations without issuing any false positives.

To ensure Bagpipe correctly checks configurations, we verified its implementation in Coq, which required developing both the first formal semantics for BGP based on RFC 4271; and SpaceSearch, a new framework for verifying solver-aided tools.

We provide evidence for the correctness and usefulness of our BGP semantics by verifying Bagpipe, formalizing Gao and Rexford's pen-and-paper proof on the stability of BGP (this proof required a necessary extension to the original proof), and performing random differential testing of C-BGP (a BGP simulator) revealing 2 bugs in C-BGP, but none in our BGP semantics.

We provide evidence for the general usefulness of SpaceSearch, by building and verifying two solver-aided tools. The first tool is Bagpipe, the second tool, SaltShaker, checks that RockSalt's x86 semantics for a given instruction agrees with STOKE's x86 semantics. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE. After these systems were patched by their developers, SaltShaker verified the semantics' agreement on 15,255 instructions in under 2h.

To my dear wife, Sarah.

ACKNOWLEDGMENTS

I am deeply grateful to my PhD advisers Michael D. Ernst and Zachary Tatlock. When I was a fresh PhD student, with no advanced knowledge of programming languages, Mike took it upon himself to teach and mentor me. Soon thereafter, Zach joined the University of Washington and introduced me to the wonderful world of formal reasoning using proof assistants. Since then, Mike and Zach not only taught me how to identify impactful research, how to effectively communicate complex ideas, and how to connect with like-minded researchers; but also provided the honest feedback, encouragement, and friendship that were essential to the completion of this PhD thesis. Mike and Zach are outstanding advisers, and I am proud to consider them my mentors, collaborators, and friends.

Special thanks also to Arvind Krishnamurthy, for teaching me about the Border Gateway Protocol (BGP) and its exciting challenges, and to Emina Torlak, for teaching me everything I know about SMT solvers and also for developing the excellent Rosette tool. I further wish to thank Stefan Heule, Steven Lyubomirsky, and Doug Woos for their help developing and evaluating Bagpipe and SpaceSearch.

Thanks also to all the wonderful people of the Programming Languages group at the University of Washington. Over the four-and-a-half years of my PhD, they have not only been excellent collaborators, eager to discuss research and push for late night paper deadlines, but also dear friends, who would literally traverse the Brazilian jungle with me.

I also wish to thank those colleagues who improved my research, e.g. by commenting on paper drafts, identifying related work, or providing mentorship. These include Nate Foster, Sorin Lerner, Ratul Mahajan, Todd Millstein, Jennifer Rexford, David Walker, and many anonymous reviewers. I am also grateful for Tim Kleefass and Sebastian Neuner from BelWü, as well as Jann Haber, Hannes Rist, and Christoph Wurm from Selfnet, for sharing their BGP configurations and providing valuable feedback. Thanks also to Magda Balazinska, Alvin Cheung, Shumo Chu, Daniel Halperin, Bill Howe, Gene Kim, Siwakorn Srisakaokul, and Dan Suciu, for their collaboration on various research projects that are not part of this thesis [32, 93, 13].

Finally, thanks to my wife for her love and support; and my parents, who are always there for me, even as I pursued my PhD on the other side of the world.

I have received so much support during my PhD, by so many people, it seems likely that I have forgotten someone. Even if you are not listed explicitly above, rest assured that I am very grateful for your support.

Parts of this thesis have previously been published. Chapter 2 is based in [96], Chapter 3 is based in [95], and Chapter 4 is based in [97].

Chapter 1

INTRODUCTION

The Internet is a collection of interconnected networks run by universities, corporations, regional ISPs, and nation-wide ISPs. These networks, collectively known as Autonomous Systems (ASes), use the Border Gateway Protocol (BGP) to exchange route announcements that describe the paths that packets (e.g. sent via TCP) can take to travel across the Internet. To route packets reliably and securely, ASes must configure their BGP routers to restrict how route announcements can be used and exchanged. For example, to avoid unprofitable routes, an AS codifies its contracts with other ASes in its BGP router configurations. Because BGP gives ASes freedom to configure their routers, BGP provides very few general guarantees — essentially all desirable properties have to be proven for a particular topology and set of router configurations.

Router configuration is a challenging and error-prone task for ASes. Large ASes maintain millions of lines of frequently-changing configurations that run distributed across hundreds of routers [36, 84]. Router misconfigurations are common and have led to highly visible failures affecting ASes and their billions of users. For example, in 2009, YouTube was inaccessible worldwide for several hours due to a misconfiguration in Pakistan [10]. In 2010 and 2014, China Telecom hijacked significant but unknown fractions of international traffic for extended periods [17, 74, 54, 51]. Goldberg surveys several additional major outages and their causes [26]. Less visible is the high cost ASes pay every day to develop and maintain configurations.

Given BGP's vital role, there exist router configuration guidelines [25, 29, 12, 89], checkers that statically check for router misconfigurations [21, 22, 7], and simulators that dynamically check for router misconfigurations [64, 66, 59, 55]. These aim to help ASes correctly configure their routers, but they fall short of providing guarantees about the absence of certain router misconfigurations, because they are based on simplified semantics of BGP or no semantics at all.

To improve upon this situation, this thesis presents Bagpipe[1]. Bagpipe defines a declarative domain-specific language that enables BGP administrators to express control-plane specifications, such as "*an AS's routers will never accept routes for invalid IP addresses*", "*an AS's routers will always forward certain routes to other ASes*", and "*an AS's routers will always prefer routes from customers over routes from providers*". Given a specification expressed in this language, Bagpipe automatically verifies that an AS's router configurations satisfy the given specification, using an SMT solver and the novel *initial network reduction*,

[1]Bagpipe is open-source, see bagpipe.uwplse.org.

which shows that a specification can be verified using a search for counterexamples of a specification over a finite, instead of infinite, space of network traces.

Bagpipe's specifications are rich enough to express policies inferred from real AS configurations, express BGP policies found in the literature (such as the Gao-Rexford model [25] and prefix-based filtering [56]), and express policies for 10 configuration scenarios from the Juniper TechLibrary [40, 9]. The Bagpipe verifier works out-of-the-box for these policies and existing Juniper and Cisco router configurations. To evaluate Bagpipe's efficiency, we applied it to three ASes with over 240,000 lines of Cisco and Juniper BGP configuration. Bagpipe found 19 apparent errors without issuing any false positives.

To ensure that Bagpipe correctly checks configurations, we verified its implementation in Coq, which required developing both the first formal semantics of the BGP specification RFC 4271, and also a new framework called SpaceSearch to verify solver-aided tools (such as Bagpipe).

In contrast to previous BGP semantics [25, 29, 89], our semantics is fully formal (it is implemented in the Coq proof assistant) and models all required features of the BGP specification modulo low-level details such as bit representation of update messages and TCP. We performed three case studies to provide evidence for the correctness of our semantics, and to show how to use our semantics as a basis for reliable tools that help BGP administrators avoid router misconfiguration. 1) We verified the soundness of Bagpipe. 2) We formalized and extended the seminal pen-and-paper proof by Gao & Rexford on the convergence of BGP, revealing necessary extensions to Gao & Rexford's original assumptions. 3) We tested the popular BGP simulator C-BGP against our semantics, revealing 2 bugs in C-BGP.

SpaceSearch is a framework to verify solver-aided tools by means of a proof assistant. Two case studies provide evidence for the general usefulness of SpaceSearch. 1) We constructed and verified Bagpipe, 2) We construct and verify SaltShaker, which checks that RockSalt's [58] x86 semantics for a given instruction agrees with STOKE's [69] x86 semantics. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE. After these systems were patched by their developers, SaltShaker verified the semantics' agreement on 15,255 instructions in under 2h.

In summary, this thesis is structured as follows:

- Chapter 2 describes Bagpipe, a formally verified solver aided tool to check BGP router configurations. The major contributions of Bagpipe are:
  - A domain specific language to express AS-wide policies as declarative specifications.
  - The initial network reduction, which enables Bagpipe to verify specifications by searching a finite set of network traces.
  - An efficient verifier based on this reduction and SpaceSearch. The verifier employs a novel combination of search space partitioning, unused variable omission, symbolic execution, and predicate abstraction.
  - An evaluation of Bagpipe on 10 configuration scenarios and three real ASes with over 240,000 lines of Cisco and Juniper BGP configuration.

- Chapter 3 describes the first mechanized formal semantics of the BGP specification RFC 4271 [65] which is implemented in Coq. We evaluate the semantics using the following case studies:
  - A soundness proof of the Bagpipe tool that checks that BGP configurations adhere to given specifications.
  - A formalization and extension of the pen-and-paper proof by Gao & Rexford on the convergence of BGP, revealing necessary extensions to Gao & Rexford's original configuration guidelines.
  - A random differential tester for the BGP simulator C-BGP, revealing 2 bug in C-BGP.
- Chapter 4 describes SpaceSearch, a framework for verifying scalable solver-aided tools in Coq. We evaluate the framework using the following case studies:
  - A solver aided tool to check BGP router configurations called Bagpipe, which is formally verified against the BGP specification RFC 4271 [65].
  - A solver aided tool called SaltShaker to verify that RockSalt's x86 semantics for a given instruction agrees with STOKE's x86 semantics. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE.

Chapter 2

BAGPIPE

2.1 Introduction

Over 3 billion people are connected to the Internet through university and corporate networks, regional ISPs, and nation-wide ISPs [35]. These networks, collectively known as Autonomous Systems (ASes), use the Border Gateway Protocol (BGP) to exchange route announcements, which describe paths that traffic can take across the Internet. To route traffic reliably and securely, ASes must configure their BGP-speaking border routers to implement policies restricting how route announcements can be used and exchanged.

Router misconfigurations are common and have led to many visible failures [17, 74, 54, 51]. For example, in 2008, in response to a government order, the Pakistan Telecom AS intended to block YouTube by announcing a non-existent YouTube route to ASes within Pakistan. Due to a misconfiguration, this route was also advertised to an AS outside Pakistan, PPCC. PPCC forwarded the route to its neighbors. The non-existent route to YouTube then quickly spread throughout the Internet and was selected for packet forwarding by most ASes. YouTube was then unavailable to most Internet users, as their packets to YouTube were incorrectly forwarded to Pakistan Telecom. About two hours later, PPCC fixed the problem by disconnecting Pakistan Telecom from the Internet [10]. If Pakistan Telecom had correctly implemented its policy to only block YouTube to ASes within Pakistan, or if PPCC had correctly implemented its policy to only import routes that an AS actually owns, this outage could have been avoided.

Other failures could also have been prevented by correctly implementing appropriate BGP policies [26]. However, doing so with little to no tool support is difficult and expensive, particularly since large ASes maintain millions of lines of frequently changing configurations distributed across hundreds of routers [36, 84].

This chapter presents Bagpipe[1], which uses automatic verification to prevent router misconfiguration. An AS operator expresses control-plane policies as declarative specifications. Then, Bagpipe verifies that the router configurations satisfy the specifications.

A straightforward implementation of Bagpipe would need to consider infinitely many possible network traces to determine whether a BGP configuration violates a router policy; if no trace is a counterexample to the specification, then the policy holds. The novel *initial network reduction* enables Bagpipe to verify a specification by searching for counterexamples over a finite space of network traces.

---

[1]Bagpipe is open-source, see http://bagpipe.uwplse.org

While the initial network reduction makes Bagpipe's search space finite, the space is still far too large to permit brute-force enumeration. Bagpipe partitions the search space by case splitting on the ranges of some symbolic variables, which enables parallel symbolic exploration using an SMT-based symbolic execution engine [82] to efficiently exploit range information within each partition. Bagpipe also uses predicate abstraction to further reduce the search space, coalescing announcements with the same control flow in the BGP configuration.

Bagpipe is an expressive, efficient, and effective verification tool. Bagpipe's policy specification language is expressive: specifications are rich enough to express policies inferred from real AS configurations, express BGP policies found in the literature (such as the Gao-Rexford model [25] and prefix-based filtering [56]), and express policies for 10 configuration scenarios from the Juniper TechLibrary [40, 9]. Bagpipe is efficient: we applied it to three ASes with over 240,000 lines of BGP configuration written in the Cisco and Juniper configuration language (which Bagpipe supports out-of-the-box). Bagpipe is effective: it revealed 19 policy violations without issuing any false positives.

This chapter's contributions include:

- A means of expressing AS-wide policies as declarative specifications (Section 2.3).
- The initial network reduction, which enables Bagpipe to verify specifications by searching a finite set of network traces (Section 2.4).
- An efficient verifier based on this reduction, which employs a novel combination of search space partitioning, unused variable omission, symbolic execution, and predicate abstraction (Section 2.5).
- An evaluation of Bagpipe on 10 configuration scenarios and on 3 real ASes with over 240,000 lines of Cisco and Juniper BGP configuration (Section 2.6).

2.2 Overview

This section provides background on BGP, shows example policies that AS operators may need to guarantee, and illustrates how Bagpipe automatically verifies such policies.

2.2.1 Background

The core of the Internet is a network of routers that forward data packets toward their destinations. Routers learn of a route — a path through other routers to a destination — via the Border Gateway Protocol (BGP). Using BGP, a router selects at most one route ϱ per destination p and, once selected, adds itself to ϱ and forwards the announcement to its neighbors. The router then forwards packets for p to the router from which ϱ was received. The forwarding and selection of routes takes place on the *control plane*. It runs separately and asynchronously from the *data plane*, on which routers forward data packets using the routes selected by the control plane. Bagpipe verifies control plane policies that characterize both (1) which routes may be selected and used by the data plane and (2) which routes a router may forward.

```
# Control plane for router r.

# The following RIBs contain the announcements
# received, selected, and forwarded by the router r.
# Initially, no such announcements are available.
∀ n pfx, adjRIBsIn[n][pfx]  = notAvailable
∀ pfx,   locRIB[pfx]        = notAvailable
∀ n pfx, adjRIBsOut[n][pfx] = notAvailable

# Process incoming announcement ann
# from the source src for the prefix pfx.
while (src, pfx, ann) = recvUpdateMessage():
  adjRIBsIn[src][pfx] = ann

  # Import and select announcements.
  locRIB[pfx] = notAvailable
  for n in r.neighbors:
    annImp = IMPORT(r, n, pfx, adjRIBsIn[n][pfx])
    if better(annImp, locRIB[pfx]):
      locRIB[pfx] = annImp

  # Export and forward announcements.
  for n in r.neighbors:
    annExp = EXPORT(r, n, pfx, locRIB[pfx])
    if annExp != adjRIBsOut[n][pfx]:
      adjRIBsOut[n][pfx] = annExp
      sendAnnBGP(n, pfx, annExp)
```

Figure 2.1: Simplified BGP Router Implementation. This pseudocode sketches the high-level behavior specified by the BGP standard, RFC 4271 [65]. AS operators can only configure the `IMPORT` and `EXPORT` rules. Section 2.3.1 describes restrictions on the `IMPORT` and `EXPORT` programs to ensure conformity with RFC 4271; for example, `IMPORT` must return `notAvailable` on announcements with AS routing loops.

The Internet's routers are owned by *Autonomous Systems* (ASes) such as universities, corporate networks, regional ISPs, and nationwide ISPs; each router is operated by exactly one AS. ASes are identified by globally unique AS numbers. Bagpipe verifies control plane policies for a single AS because AS operators do not control their neighbors' configurations. We call the routers owned and operated by the single AS under consideration *internal*; other routers are *external* and may behave arbitrarily.

Router control planes forward routes via *update messages*, which consist of a *prefix* and an *announcement*.

- A prefix is a set of destination IP addresses. Sets of destinations are represented in Classless Interdomain Routing (CIDR) notation. A set of destinations in CIDR notation ip/n (e.g. `192.168.1.0/24`) is referred to as a *prefix*, because it contains all IP addresses starting with the same n bits as ip.
- An announcement contains metadata about the route including: *AS-path*, a list of the ASes the announcement has previously traversed which is used for avoiding routing loops as well as serving as a measure of "distance" on the Internet; *communities*, a set of flags which are uninterpreted by the BGP protocol but can be used by ASes to exchange additional information about a route; and *local preference*, a number that influences route selection.

Figure 2.1 sketches how each Internet router r manages the control plane, as described by the BGP specification, RFC 4271 [65]. r maintains three tables, or *Routing Information Bases*, to track announcements it has received (`adjRIBsIn`), selected for routing packets on the data plane (`locRIB`), and forwarded to its neighbors (`adjRIBsOut`). Initially (lines 6 to 8), all these tables contain only `notAvailable` to indicate that r has not received any announcements. r then enters an infinite loop to process incoming BGP update messages. Each incoming update message (line 12) includes the address *src* of the router that forwarded the update message, the prefix *pfx* that the update message is offering to route to, and the announcement *ann*. Receipt of an update message triggers a three-phase process.

1. r stores the received announcement in its `adjRIBsIn` routing table for announcements for prefix *pfx* from neighbor *src* (line 13).

2. The router applies the configurable `IMPORT` program to each announcement in `adjRIBsIn` for prefix *pfx* and each neighbor n (lines 17 to 18). `IMPORT` may transform or replace the announcement, including returning `notAvailable`.

   The router chooses the best (possibly-transformed) announcement (lines 19 to 20). An announcement with higher local preference is considered `better` than an announcement with lower local preference. If two announcements have the same local preference, other factors like the length of their AS-path are considered. The router stores the best announcement in `locRIB[`*pfx*`]`, which is used by the data plane to forward packets.

Since AS operators cannot directly control the `better` test that selects announcements, they influence selection by configuring their `IMPORT` rules to either drop announcements or to modify the local preference so that the `better` test (line 19) will select their desired announcements.

3. For every neighbor, the router applies the configurable `EXPORT` program to the announcement that was selected as best. If the result of `EXPORT` differs from the announcement that was most recently sent to that neighbor, then `EXPORT`'s result is stored in `adjRIBsOut` and forwarded to the neighbor. A router can retract a previously advertised announcement by announcing `notAvailable` for the route's destination prefix.

To verify an AS configuration, it is not sufficient to check the `IMPORT` and `EXPORT` programs for each router individually. This is because internal routers establish invariants on the announcements they forward to other internal neighbors (e.g. the ISP Internet2 establishes the invariant that its internal routers never send announcements for invalid prefixes). The configurations of those neighbors often rely on such invariants. Thus, the verification task is to establish that the AS configuration correctly implements the policy for all routers in the AS at all times, for all possible sequences of incoming update messages.

Figure 2.2 illustrates an example network of several ASes. One AS is under consideration with three internal routers r_0, r_1, and r_2. The external router e_0's data-plane is capable of directly delivering packets with destinations in the prefix `128.208.7.0/24`. Note that the data plane delivers packets along a route in the opposite direction from which update messages for that route were forwarded through the control plane. Consider the following example *trace*, i.e., sequence of events happening in the AS:

1. The internal router r_0 receives an update message m_0 from the external router e_0, which indicates that e_0 has a route to the prefix `128.208.7.0/24`.
2. r_0 extends m_0's announcement with its own AS number, and selects it to route packets.
3. r_0 forwards the selected announcement in the update message m_1 to its neighbors, including the internal router r_1.
4. r_1 receives the update message m_1 from r_0.[2]
5. r_1 selects the received announcement to route packets.
6. r_1 forwards the selected announcement in the update message m_2 to its neighbors, including the external router e_0.

r_0, r_1, and e_0 can use the selected announcements to forward packets with destinations in the prefix `128.208.7.0/24`: for example, `128.208.7.1` and `128.208.7.42` but not `128.208.8.0`.

[2] r_1 does not extend m_1 with its own AS number because it received the announcement from an internal neighbor.

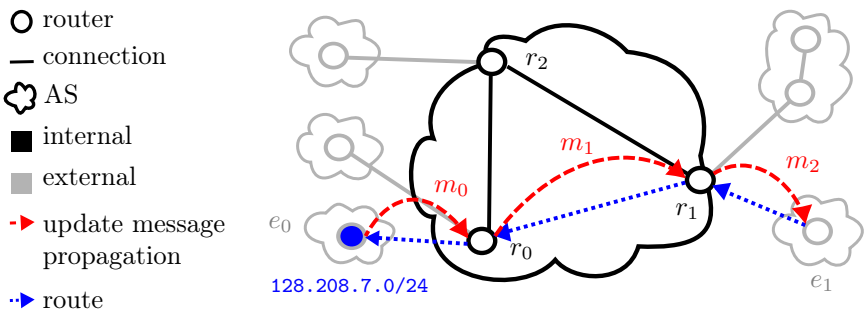

Figure 2.2: BGP Routing Example. The AS under consideration (black) consists of three internal routers connected to five external routers. These routers use the Border Gateway Protocol (BGP) to establish a route between e_1 and e_0. This route can be used to forward packets.

2.2.2 Policies

An AS operator decides on policies that restrict the announcements that the AS's BGP routers' control planes select and exchange. Some policies are used to ensure security properties: for instance, a country's ASes might want to ensure that national traffic is routed within the country [68]. Other policies are used to uphold business contracts, for example, to honor agreements that certain announcements should not be publicly shared [74].

This section uses the *BlockToExternal* policy as a running example. *BlockToExternal* prohibits internal routers from forwarding "classified" announcements to external routers. An announcement is considered classified if it had the `BTE` community set *when it was received by the AS*.

Bagpipe's specifications are assertions written in the Racket language that restrict the announcements selected and exchanged by an AS's internal routers. In Bagpipe, the *BlockToExternal* policy is expressed as:

```
(implies
  (external? receiver)
  (not (has-community? 'BTE (original sent))))
```

The policy asserts that if an announcement (*sent*) is forwarded by an internal router to one of its neighboring external routers (*receiver*), then the announcement that gave rise to *sent* (i.e. the original announcement received by the AS before it was modified by import and export filters), must not have contained the `BTE` community.

AS operators implement policies by configuring their AS's routers. A BGP router is configured with `IMPORT` and `EXPORT` programs that modify announcements in order to influence which ones are selected and forwarded (and therefore used to route packets on the data plane). These programs are typically written in either the Juniper or Cisco configuration

language, which are loop-free imperative programming languages with domain-specific syntax and semantics.

Consider the following snippet of Juniper configuration code to implement the *BlockToExternal* specification for the neighbor with IP `62.40.125.17`.

```
neighbor 62.40.125.17 {                                  — start neighbor configuration
  export [RULE1 RULE2... BLOCK-BTE...];                  — call export rules
  ...}
policy-statement BLOCK-BTE {                             — define export rule
  term block-to-external {
       from community BTE;                               — match announcement
       then reject; }}                                   — reject matched announcement
```

The `export` statement runs each passed rule (`policy-statement`) from left to right, stopping once a rule either accepts or rejects the announcement. Each executed rule can also modify the announcement. Note that if the AS operator does not correctly order statements, they may not fire on the announcements they are intended to check or modify. The `BLOCK-BTE` rule rejects an announcement if it has the `BTE` community set, and otherwise does nothing.

To manually verify that an AS's router configurations implement the *BlockToExternal* specification, an AS operator must check that:

- For every external neighbor, every router has an export rule that drops announcements whose `BTE` community is set. These rules are similar to the `BLOCK-BTE` rule in the example above.
- Each of these rules is executed: no preceding rule accepts the announcement. For example, the rules `RULE1` and `RULE2` in the above example configuration should not accept announcements whose `BTE` community is set.
- No rule in any router clears the `BTE` community.

More complex policies, such as the Gao-Rexford policy discussed later in this chapter, are even harder to verify manually. Large ASes have millions of lines of configuration, making manual verification of even simple policies expensive and error-prone.

2.2.3 Bagpipe

As illustrated in Fig. 2.3, Bagpipe enables AS operators to express their AS-wide policies as declarative specifications, and Bagpipe then *automatically* verifies that their router configurations correctly implement such policies. Since AS operators generally do not have access to their neighboring ASes' configurations, Bagpipe soundly assumes that external routers may exhibit any behavior. Bagpipe verifies control plane policies to ensure that inter-AS guarantees are upheld (e.g. confidential announcements are not leaked) and to prevent control-plane performance issues. For example, ASes often reject update messages

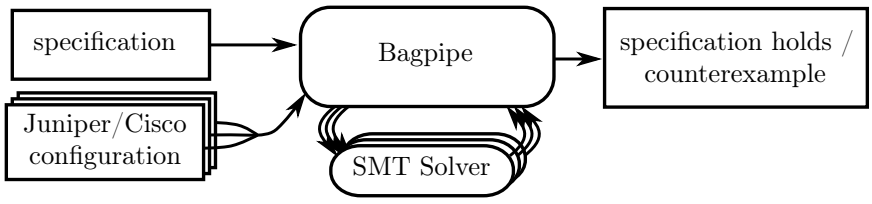

Figure 2.3: Bagpipe Workflow. Bagpipe takes a specification and an AS's router configurations as input. Bagpipe verifies that the configuration correctly implements the specification using multiple concurrent SMT solver calls, and then either indicates success or returns a counterexample that AS operators can use to debug their configurations.

whose prefix is too long (corresponding to a too-small set of destinations) in order to avoid filling their routing tables with too many routes and thus degrading performance.[3]

To verify that an AS's routers are safely configured, Bagpipe must ensure that every possible behavior of the AS satisfies the policy. Bagpipe models the behavior of an AS as a *trace* of announcement receipts, route selections, and forwarding events in the network. Bagpipe models the policy in Racket as a boolean predicate over traces. For example, to verify the *BlockToExternal* specification from Section 2.2.2, Bagpipe must guarantee that in every trace, no internal router forwards a classified announcement. Note that Bagpipe verifies policies even for oscillating and non-deterministic BGP networks; thus, *the guarantees provided by Bagpipe hold even outside a BGP network's steady state.*

To verify a policy, it is not feasible to brute-force search the space of all possible traces for a counterexample, because the set of all traces is infinite. Not only can external routers send arbitrarily long sequences of announcements, but the BGP protocol itself can oscillate (in which case routers keep announcing and withdrawing routes indefinitely).

To address this challenge, Section 2.4 introduces the *initial network reduction* (INR). In the *initial network*, all RIBs contain only `notAvailable` (as in Fig. 2.1 after line 8). Intuitively, the INR exploits the observation that if a router will ever select or forward a particular announcement, it will also do so the initial network. An AS's behavior is thus "maximal" in the initial network. Bagpipe exploits this fact to reduce its search for a counterexample from the infinite set of traces to a finite set of traces that process two arbitrary external announcements in the initial network.

The initial network reduction makes Bagpipe's search space finite, but it is still too large for naive brute force search. Section 2.5 introduces four optimizations that Bagpipe implements

[3]From the data plane perspective, this behavior may seem unreasonable: if an AS would be willing to forward a packet with an IP in the prefix `64.57.29.0/24` to a particular AS, it seems it should also be willing to forward a packet with an IP in the range `64.57.29.0/25` (since the latter represents a strict subset of the former). However, filling RIBs with many announcements for small prefixes can severely degrade performance.

to improve search performance. Instead of using brute-force search, Bagpipe searches the space symbolically using an SMT solver (§2.5.1). Bagpipe uses predicate abstraction to soundly reduce the number of announcements that must be considered (§2.5.2). Bagpipe hoists symbolic variables so that search can be parallelized across many nodes in a cluster (§2.5.3). Bagpipe avoids enumerating hoisted variables whose values are not used (§2.5.4).

2.3 Specifications

This section first formalizes the behavior of the BGP control plane in Section 2.3.1; and then uses the formalization to describe a general framework for expressing BGP policy specifications, and what it means for these specifications to hold in Section 2.3.2.

2.3.1 Control Plane Formalization

This section and Figure 2.4 formalize the BGP control plane discussed in Section 2.2. A more complete formalization appears in a technical report [94].

The set of routers is represented as a set of IP addresses, connected via bidirectional links over which routers exchange announcements.

A *trace* is a sequence of *event*s within the AS under consideration. There are three kinds of event. The event $recv(r, i, p, a)$ means that a router r received an update message from its neighbor i for prefix p with announcement a. The event $slct(r, p, a)$ means that a router r selected an announcement a for prefix p. The event $frwd(r, o, p, a)$ means that a router r forwarded an update message to its neighbor o for prefix p with announcement a.

A valid trace must have the following 4 properties: 1) A router can receive an update message from an internal neighbor i only if there has previously been a corresponding forwarding event by i. A router can always receive an update message from an external neighbor, because Bagpipe treats external neighbors as "havoc". 2) A router selects an announcement in its `locRIB` if and only if it is chosen as the result of the import and selection process (shown in Fig. 2.1 on lines 16 to 20). 3) A router can forward an update message only if its announcement is the result of applying the export process (shown in Fig. 2.1 on lines 23 to 27) to a selected announcement. 4) The receipt of an update message by a router must be immediately followed by all the resulting select and forward events at that router. The end of Section 2.2.1 provides an example trace. A network state is *reachable* via some trace t if it is the result of executing every event in the trace t.

$\mathcal{A}$ represents an announcement as specified by the BGP protocol. It is a record consisting of an AS path *aspath* (the AS numbers of every AS traversed by the announcement), local preference *pref* (the `better` routine of Fig. 2.1 prefers announcements with higher local preference), and a set of communities *communities* (used by ASes to exchange additional information about a route; a community is a number). Bagpipe can be easily extended to track additional information in announcements, e.g. to model BGP extensions.

For reasoning, Bagpipe uses *tagged announcements* which consist of the *current* value of the announcement, plus the *original* value of the announcement at the time when it entered

$IP := [0,255] \times [0,255] \times [0,255] \times [0,255]$ — ip addresses
$P := IP \times [0,32]$ — prefixes

$R \subseteq IP$ — routers
$R_i \subseteq R$ — internal routers
$R_e \subseteq R$ — external routers
$in(r) = out(r) = neighbor(r) \subseteq R$ — router r's neighbors

$$\begin{aligned} event := \; & recv : (r : R) \to in(r) \to P \to A \to event \\ \mid \; & slct : (r : R) \to P \to A \to event \\ \mid \; & frwd : (r : R) \to out(r) \to P \to A \to event \end{aligned}$$

$trace \subseteq list(event)$ — a trace is a legal sequence of events

na — placeholder used when announcement is not available
$\mathcal{A} := \{pref : \mathbb{N}, communities : \mathcal{P}(\mathbb{N}), aspath : list(asn)\}$
— BGP announcement; $\mathcal{P}$ stands for powerset.
$A := \{current : \mathcal{A}, original : \mathcal{A}\} \cup \{na\}$ — tagged ann

$imp : (r : R) \to in(r) \to P \to A \to A$ — `IMPORT` program of Fig. 2.1
$exp : (r : R) \to out(r) \to P \to A \to A$ — `EXPORT` program of Fig. 2.1

$\texttt{adjRIBsIn} : trace \to (r : R) \to in(r) \to P \to A$ — active received
$\texttt{locRIB} : trace \to (r : R) \to P \to A$ — active selected
$\texttt{adjRIBsOut} : trace \to (r : R) \to out(r) \to P \to A$ — active fwded

Figure 2.4: Control Plane Formalization. A router r is connected via incoming $in(r)$ and outgoing $out(r)$ links. A *trace* is a valid sequence of *event*s within the AS under consideration. *imp* and *exp* refer to a router's configurable import and export programs. In any state reachable via some trace t, $\texttt{adjRIBsIn}(t, r, i, p)$ contains the announcement most recently received for a prefix p by a router r from r's neighbor i. $\texttt{locRIB}(t, r, p)$ contains the announcement most recently selected for a prefix p by a router r. $\texttt{adjRIBsOut}(t, r, o, p)$ contains the announcement most recently forwarded for a prefix p by a router r to r's neighbor o.

the AS under consideration. Tracking this additional provenance information enables AS operators to write policy specifications such as *BlockToExternal*.

A router r imports an announcement a_i for prefix p received from neighbor i using the import program $imp(r, i, p, a_i)$. *imp* is a loop-free imperative program that either modifies or drops (by returning *na*) the passed announcement. The export program *exp* is modeled similarly. These programs are called `IMPORT` and `EXPORT` in the BGP specification and in Section 2.2.

For compliance with RFC 4271, there are some restrictions on how the `IMPORT` and `EXPORT` programs can operate. For example, both programs must map `notAvailable` inputs to `notAvailable` outputs, `IMPORT` must mark announcements with AS routing loops as `notAvailable`, and `EXPORT` must prevent announcements received from internal neighbors from being exported to other internal neighbors.

Each router r stores the most recently received, selected, and forwarded announcements. These are the *active* announcements that can be used to forward packets. Specifically:

- `adjRIBsIn` contains the active received announcements: the most recently received announcement for each prefix and neighbor of r. A new announcement for some prefix from a neighbor always implicitly withdraws (and thus deactivates) any previously-received announcements for that prefix and neighbor.
- `locRIB` contains the active selected announcements: the most recently selected announcement for each prefix.
- `adjRIBsOut` contains the active forwarded announcements: the most recently forwarded announcement for each prefix and neighbor of r. Older routes are implicitly withdrawn by BGP.

Given a trace t, a router r, a neighbor i of r, and a prefix p, the function `adjRIBsIn` computes the network state resulting from the execution of every event in the trace t, i.e. the state reachable via the trace t, and returns the most recently received announcement by r from i with prefix p in that state, or *na* if no announcement has been received. Note that the `adjRIBsIn` formalism takes more arguments than `adjRIBsIn` of Fig. 2.1. When applied to a trace t and a router r, it corresponds to `adjRIBsIn`. The `locRIB` and `adjRIBsOut` are modeled similarly.

2.3.2 Policy Specifications

This section describes a general framework for expressing BGP policies specifications, and what it means for these specifications to hold. Later sections show how Bagpipe automatically verifies that specifications in this general framework hold.

A specification is an invariant over an AS's active announcements, i.e. an invariant over an AS's routing information bases. Formally, a specification τ is a predicate over a record of variables V. V represents certain values in *router*'s routing information bases, namely an announcement *received* from a *sender* for *prefix*, an announcement *selected* for *prefix* which

$V := \{router \in R; prefix \in P; sender \in R; received \in A;$
$\quad selected \in A; bestSender \in R; bestReceived \in A;$
$\quad receiver \in R; sent \in A\}$

$spec : Type := V \to bool$

$specHolds(\tau : spec) :=$
$\quad \forall\, (t : trace)\ (r \in R_i)\ (p \in P)\ (i \in in(r))\ (o \in out(r)),$
$\quad \texttt{let}\ i^* := \texttt{inLocRIB}(t, r, p)$
$\quad\quad a_i := \texttt{adjRIBsIn}(t, r, p, i)$
$\quad\quad a_i^* := \texttt{adjRIBsIn}(t, r, p, i^*)$
$\quad\quad a_l^* := \texttt{locRIB}(t, r, p)$
$\quad\quad a_o := \texttt{adjRIBsOut}(t, r, p, o)$
$\quad \texttt{in}\ \ \tau(\{router := r; prefix := p; sender := i; received := a_i;$
$\quad\quad\quad selected := a_l^*; bestSender := i^*; bestReceived := a_i^*;$
$\quad\quad\quad receiver := o; sent := a_o\}) = true$

Figure 2.5: Policy Specification Definition. A policy specification $\tau : spec$ is a predicate over a record of variables V, representing certain active announcements. $specHolds(\tau)$ defines what it means for a specification to hold.

was received as *bestReceived* from *bestSender*, and an announcement *sent* to a *receiver* for *prefix*.

The definition of a specification *spec*, and what it means for a specification τ to hold $specHolds(\tau)$, is given in Figure 2.5. A specification τ holds (i.e. an AS's routers correctly implement a specification), if and only if the invariant expressed by τ is true for every router state reachable via any trace. Formally, τ holds if and only if for any network trace t, router r, prefix p, neighbor i, and neighbor o, τ returns *true* when invoked with the most recently received announcement a_i for prefix p and neighbor i, the most recently selected announcement a_l^* for prefix p which was received as a_i^* from i^* (starred variables are associated with the selected announcement), and the most recently forwarded announcement a_o for prefix p and neighbor o, along with r, p, i, and o.

The function $\texttt{inLocRIB}(t, r, p)$ computes the neighbor i^*. $\texttt{inLocRIB}(t, r, p)$ first passes all announcements in the $\texttt{adjRIBsIn}(t, r, i, p)$ to the *imp* program, and then determines the neighbor with the "best" imported announcement.

Expressiveness Specifications are not arbitrary invariants over all active announcements — specifications can only quantify over variables in V. In particular, a specification cannot depend on routers other than r, announcements for prefixes other than p, announcements received from neighbors other than i and i^*, and announcements forwarded to neighbors other than o. For example:

- A specification *cannot* require routers to forward a received announcement for prefix p, if and only if an announcement for some other prefix q has been selected.
- A specification *cannot* require routers to select a received announcement from neighbor i, if and only if exactly k (where $k \geq 3$) announcements from other neighbors have been received.
- A specification *can* require routers to select a received announcement with either prefix p or q, if and only if that announcement has the `IMPORTANT` community set.
- A specification *can* require routers to forward an announcement a_o, if and only if a_o is equal to the selected announcement a_l^*.

As shown in the evaluation, even with these restrictions, Bagpipe still provides sufficient expressiveness for many interesting policies. This is due to two properties of BGP:

1) A BGP router r selects and forwards announcements for a certain prefix p, completely independent of any announcements for any other prefix p' or any other router r' (see Fig. 2.1). For example, the first disallowed policy above cannot be implemented, because the decision process for announcements of prefix p cannot inspect announcements for prefix q.

2) The *imp* and *exp* programs can only consider one announcement at a time. This restriction is defined by the BGP specification:

> [A router's *imp*/*exp* programs] SHALL NOT use any of the following as its inputs: the existence of other routes, the non-existence of other routes, or the path attributes of other routes. [65]

For example, the second disallowed policy above cannot be implemented, because an import rule can only consider a single received announcement at a time.

Instead of interpreting a specification as a network state invariant, some (but not all) specifications can also be interpreted as a the composition of an *import specification*, an *export specification*, and a *selection ranking*.

An import specification π_i restricts how routers can import received announcements. Formally, $\pi_i(r, p, i, a_i, a_l) : bool$ holds if and only if for any network trace, π_i returns *true* for any announcement a_i which was received by router r from neighbor i for prefix p, and which was imported as announcement a_l (a_l does not have to be selected). Note that an import specification quantifies only over a single announcement, i.e. an import specification restricts all announcements independently. An export specification π_e restricts how routers can export selected announcements, and is formalized similarly to an import specification. Import and export specifications resemble the domain-specific languages employed by Frenetic [23], NetCore [57], and NetKat [1].

A selection ranking $\leq$ restricts which announcements routers can select. Formally, $\leq (r, p, i, a_i, i^*, a_i^*) : bool$ holds if and only if for any network trace, router r, prefix p, and neighbor i, the announcement a_i received from neighbor i is ranked lower than the announcement a_i^* received from neighbor i^* (i^* is the neighbor from which r has selected the

announcement). In contrast to much related work, the selection ranking is unique because it considers multiple announcements simultaneously.

Many specifications τ can be decomposed into an import specification π_i, an export specification π_e, and a selection ranking $\leq$ as follows:

$$
\begin{aligned}
&\tau(v) : \mathit{spec} := \\
&\quad \pi_i(\mathit{router}(v), \mathit{prefix}(v), \mathit{bestSender}(v), \\
&\qquad \mathit{bestReceived}(v), \mathit{selected}(v)) \wedge \\
&\quad \pi_e(\mathit{router}(v), \mathit{prefix}(v), \mathit{receiver}(v), \\
&\qquad \mathit{selected}(v), \mathit{sent}(v)) \wedge \\
&\quad \leq (\mathit{router}(v), \mathit{prefix}(v), \mathit{sender}(v), \mathit{received}(v), \\
&\qquad \mathit{bestSender}(v), \mathit{bestReceived}(v))
\end{aligned}
$$

While some specifications cannot be decomposed into this form, e.g. if the specification relates $\mathit{received}(v)$ and $\mathit{sent}(v)$, all specifications that we expressed for the evaluation of Bagpipe can.

The rest of this section describes examples of useful policies, and shows how they are expressed as specifications.

Block To External Specification Section 2.2 described the *BlockToExternal* specification, which prohibits internal routers of the AS under consideration from forwarding classified announcements to any external routers. An announcement is considered classified if it had the `BTE` (block to external) community set at the time that it was received by the AS. We can express this specification as:

$$
\begin{aligned}
&\mathit{BlockToExternal}(v) := \mathit{receiver}(v) \in R_e \rightarrow \\
&\qquad\qquad\qquad BTE \in \mathit{communities}(\mathit{original}(\mathit{sent}(v)))
\end{aligned}
$$

An AS's router configurations correctly implement this specification if and only if it is true that $\mathit{specHolds}(\mathit{BlockToExternal})$. Inlining the definitions of *specHolds* and *BlockToExternal*, as well as removing unused variables, leads to the formula below, which states that an AS's router configurations correctly implement *BlockToExternal* if and only if the most recently forwarded announcement a_o in any reachable router state of any internal router r is not classified.

$$
\begin{aligned}
&\forall\, (t : \mathit{trace})\ (r \in R_i)\ (p \in P)\ (o \in \mathit{out}(r)), \\
&\quad \texttt{let}\ \ a_o := \texttt{adjRIBsOut}(t, r, p, o) \\
&\quad \texttt{in}\ \ \ o \in R_e \rightarrow BTE \in \mathit{communities}(\mathit{original}(a_o))
\end{aligned}
$$

Note that the *BlockToExternal* specification is an invariant on an AS's `adjRIBsOut`. Further, *BlockToExternal* can be decomposed into an export specification $\pi_e(r, p, o, a_l, a_o) = o \in R_e \rightarrow BTE \in \mathit{communities}(\mathit{original}(a_o))$, and an import specification and selection ranking that always return *true*. We say that *BlockToExternal* is composed of only an export specification.

No Martian Specification The *BlockToExternal* specification restricts which announcements an AS can forward. The *NoMartian* specification restricts which announcements an AS can select.

The *NoMartian* specification prohibits internal routers from selecting a route announcement *selected* for martian prefixes *prefix*, i.e. invalid prefixes such as the private prefix `10.0.0.0/8` or the loop-back prefix `127.0.0.0/8` which should not be used to forward packets over the Internet. Formally:

$$\mathit{NoMartian}(v) := \text{martian}(\mathit{prefix}(v)) \rightarrow \mathit{selected}(v) = na$$

The specification holds iff $\mathit{specHolds}(\mathit{NoMartian})$, i.e.:

$$\begin{array}{l}\forall\,(t : \mathit{trace})\,(r \in R_i)\,(p \in P),\\ \quad \texttt{let}\ \ a_l := \texttt{locRIB}(t, r, p)\\ \quad \texttt{in}\ \ \ \text{martian}(p) \rightarrow a_l = na\end{array}$$

Note that the *NoMartian* specification is an invariant on an AS's `locRIB`, and is composed of only an import specification.

Gao-Rexford Specification According to the Gao-Rexford model [25], a widely-used description of AS behavior, there are three kinds of relationship that an AS can have with any of its neighbors: *customer*, *peer*, or *provider*. Customers pay the AS to forward packets, peers neither charge nor pay money to forward packets, and providers charge money to forward packets. To maximize profit, an AS's routers should thus prefer an announcement from (i.e. a route through) a customer over the announcement from a peer or provider, and should prefer the announcement from a peer over the announcement from a provider. This preference can be captured with a relation $<$, where $\mathit{peer} < \mathit{customer}$, $\mathit{provider} < \mathit{customer}$, and $\mathit{provider} < \mathit{peer}$. The *GaoRexford* specification prohibits a router from selecting an announcement a_i^* that is "worse" than any announcement a_i received from some neighbor i. Given a function $\mathit{relationship}(a)$ that returns the relationship of the neighbor from which a was received, *GaoRexford* can be defined as follows:

$$\begin{array}{l}\mathit{GaoRexford}(v) := \mathit{relationship}(\mathit{received}(v)) \leq\\ \qquad\qquad\qquad \mathit{relationship}(\mathit{bestReceived}(v))\end{array}$$

The specification holds iff $\mathit{specHolds}(\mathit{GaoRexford})$, i.e.:

$$\begin{array}{l}\forall\,(t : \mathit{trace})\,(r \in R_i)\,(p \in P)(i \in \mathit{in}(r)),\\ \quad \texttt{let}\ \ i^* := \texttt{inLocRIB}(t, r, p)\\ \qquad\ \ a_i := \texttt{adjRIBsIn}(t, r, p, i)\\ \qquad\ \ a_i^* := \texttt{adjRIBsIn}(t, r, p, i^*)\\ \quad \texttt{in}\ \ \ \mathit{relationship}(a_i) \leq \mathit{relationship}(a_i^*)\end{array}$$

Note that the *GaoRexford* specification is composed of only a selection ranking.

2.4 The Initial Network Reduction

As defined in Section 2.3, a specification holds if it evaluates to true for all network states reachable by any trace in the infinite set of possible traces. However, to verify *specHolds* using current automated solvers, it is necessary to reduce the problem to one without universal quantification over the infinite set of traces. This section describes the *initial network*

$specHolds(\tau : spec)$

$\iff$

$\forall\,(t : trace)\;(r \in R_i)\;(p \in P)\;(i \in in(r))\;(o \in out(r)),$

```
  let i* := inLocRIB(t, r, p)
      a_i := adjRIBsIn(t, r, p, i)
      a_i* := adjRIBsIn(t, r, p, i*)
      a_l* := locRIB(t, r, p)
      a_o := adjRIBsOut(t, r, p, o)
  in  τ({router := r; prefix := p; sender := i; received := a_i;
         selected := a_l*; bestSender := i*; bestReceived := a_i*;
         receiver := o; sent := a_o}) = true
```

$\iff$ — 1) rewrite RIBs

$\forall\,(t : trace)\;(r \in R_i)\;(p \in P)\;(i \in in(r))\;(o \in out(r)),$

```
  let i* := inLocRIB(t, r, p)
      a_i := adjRIBsIn(t, r, p, i)
      a_i* := adjRIBsIn(t, r, p, i*)
      a_l* := imp(r, i*, p, a_i*)
      a_o := exp(r, o, p, a_l*)
  in  τ({router := r; prefix := p; sender := i; received := a_i;
         selected := a_l*; bestSender := i*; bestReceived := a_i*;
         receiver := o; sent := a_o}) = true
```

$\impliedby$ — 2) generalize best neighbor

$\forall\,(t : trace)\;(r \in R_i)\;(p \in P)\;(i, i^* \in in(r))\;(o \in out(r)),$

```
  let a_i := adjRIBsIn(t, r, p, i)
      a_i* := adjRIBsIn(t, r, p, i*)
      a_l := imp(r, i, p, a_i)
      a_l* := imp(r, i*, p, a_i*)
      a_o := exp(r, o, p, a_l*)
  in  pref(a_l) ≤ pref(a_l*) →
      τ({router := r; prefix := p; sender := i; received := a_i;
         selected := a_l*; bestSender := i*; bestReceived := a_i*;
         receiver := o; sent := a_o}) = true
```

$\impliedby$ — 3) generalize received announcements

$\forall\,(r \in R_i)\;(p \in P)\;(i, i^* \in in(r))\;(o \in out(r))\;(a_i, a_i^* \in A),$

```
  let a_l := imp(r, i, p, a_i)
      a_l* := imp(r, i*, p, a_i*)
      a_o := exp(r, o, p, a_l*)
  in  transmittable(r, p, i, a_i) → transmittable(r, p, i*, a_i*) →
      pref(a_l) ≤ pref(a_l*) →
      τ({router := r; prefix := p; sender := i; received := a_i;
         selected := a_l*; bestSender := i*; bestReceived := a_i*;
         receiver := o; sent := a_o}) = true
```

$\iff$

$\texttt{INR}(\tau)$

```
where
  transmittable(r, p, i, a) :=
    a = na ∨
    ∃ (a_0 ∈ A) (ξ ∈ path(i, r)), a = transmit(ξ, p, a_0)
```

Figure 2.6: The Initial Network Reduction. This reduction soundly removes the universal quantification over the infinite set of traces from *specHolds*. The reduction proceeds in three consecutive steps. 1) *Rewrite RIBs* rewrites a_l^* and a_o in terms of a_i^*. 2) *Generalize best neighbor* strengthens *specHolds* with facts about selected announcements and then generalizes i^*. 3) *Generalize received announcements* strengthens *specHolds* with facts about received announcements and then generalizes a_i and a_i^*. Terms added in each step are blue.

reduction, which proves that a specification holds if it evaluates to true for all network states reachable by any trace in the *finite* set of traces that arise in the *initial network*. At a high level, the initial network of an AS corresponds to the network state where all routers have initialized their RIBs to na (corresponding to the state after line 8 in Fig. 2.1). We have proven this reduction in Coq. The full formalization of the BGP semantics and the reduction are out of scope for this chapter, but can be found in Chapter 3. Here we focus on the reduction and its intuition.

To understand the intuition behind the initial network reduction, consider an announcement a received by a router r via some trace t. To be received by r, announcement a had to be transmitted along some path (a sequence of connected routers) through the network. The *initial trace* t' of t with respect to a contains only those events in t that transmit a, but not those events that transmit any other announcements (i.e. t' transmits a in the initial network without any other announcements that could interfere).

For an announcement to be transmitted along a path, all the routers along the path have to select and forward the announcement. This means that in trace t: 1) for every router along the path the announcement was selected because it was better than all other announcements, and 2) the announcement was forwarded because it was different from the previously forwarded announcement.

If an announcement was transmitted in t along the path ξ, it will also be transmitted in t' along ξ because: 1) for every router along the path the announcement is selected because there are no other announcements that could be better, and 2) the announcement is forwarded because it is different from the initial announcement na stored in all RIBs.

This implies that if Bagpipe verifies that a specification holds for those announcements that can be transmitted via the *initial trace*, then the specification also holds for those announcements that can be transmitted via any trace. It thus suffices to only consider the finite set of initial traces, instead of the infinite set of all traces. Thus, an AS's behavior in the initial network is "maximal" in a sense: if an announcement will ever be selected or forwarded by a router in any network state, it will also be selected or forwarded in the initial network.

To illustrate, consider the full trace t corresponding to Fig. 2.7, where router e_0's data-plane can directly deliver packets for destinations in `128.208.7.0/24`:

1. r_1 receives m_1 from e_0.
2. r_1 imports m_1, resulting in a_2, and selects a_2.
3. r_1 exports a_2, resulting in m_3, and forwards m_3 to r_0.
4. r_0 receives m_3 from r_1.
5. r_0 imports m_3, resulting in a_5, and selects a_5.
6. r_0 receives m_6 from e_0.
7. r_0 imports m_6, resulting in a_7, and selects a_7.
8. r_0 exports a_7, resulting in m_8, and forwards m_8 to r_2.
9. r_2 receives m_8 (containing a_8).

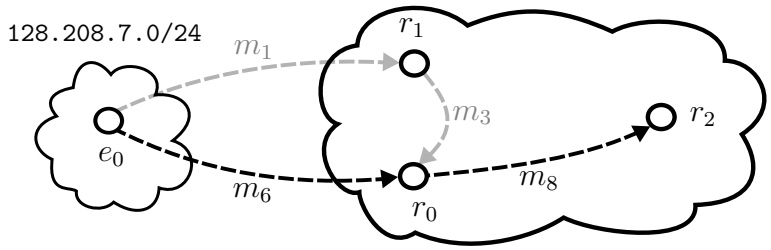

Figure 2.7: Initial Network Reduction Example.

Black events are in the initial trace t' of t with respect to the announcement a_8. Gray events are not in the initial trace t' (these events are also gray in the figure). Subscripted variables r, e, m, and a correspond to internal routers, external routers, update messages, and announcements respectively.

We focus on the announcement a_8 received by router r_2, i.e. $\texttt{adjRIBsIn}(t, r_2, \texttt{128.208.7.0/24}, r_0)$. The announcement a_8 is the result of transmitting the original update message m_6 along the path $\xi = [e_0, r_0, r_2]$ via the network trace t.

It follows that a_8 is also the result of transmitting m_6 along ξ via the initial trace t' consisting only of the black events in trace t. t' is still legal despite the fact that r_o does not receive m_3 because there is no better announcement than a_7 for r_0 to select, and the value of r_0's RIBs is *na* and thus different from a_8

The following paragraphs and Fig. 2.6 use the above insight to precisely explain the *initial network reduction* which eliminates *specHolds*'s quantification over all traces. As mentioned earlier, we have proven this reduction in Coq, but the full formalization of the BGP semantics and the reduction are out of scope for this chapter. The initial network reduction proceeds in three steps.

1. Rewrite RIBs By the aforementioned insight, it suffices to only consider the initial traces for received announcements in the `adjRIBsIn`, but to verify a specification we have to also consider the announcements in the `adjRIBsOut` and `locRIB`. This step eliminates *specHolds*'s dependence on `adjRIBsOut` and `locRIB`. This is achieved by rewriting a) the forwarded announcements in the `adjRIBsOut` in terms of `locRIB` and b) the selected announcements in `locRIB` in terms of `adjRIBsIn`. These rewrites are possible because the `adjRIBsOut` is computed from the `locRIB`, and the `locRIB` in turn is computed from the `adjRIBsIn`, by the algorithm described in Fig. 2.1.

a) The announcement $\texttt{adjRIBsOut}(t, r, p, o)$ forwarded by router r to some neighbor o is computed by applying the export program *exp* to the announcement $\texttt{locRIB}(t, r, p)$ selected by r. $\texttt{adjRIBsOut}(t, r, p, o)$ is therefore equal to $exp(r, o, p, \texttt{locRIB}(t, r, p))$.

b) The announcement $\texttt{locRIB}(t, r, p)$ selected by router r is computed by first applying the import program *imp* to each announcement $\texttt{adjRIBsIn}(t, r, i, p)$ received by a neighbor

i of r, and then selecting the "best" one (as described in Fig. 2.1). Given the neighbor i^* from which the "best" announcement was selected, $\texttt{locRIB}(t, r, p)$ is therefore equal to $imp(r, i^*, p, \texttt{adjRIBsIn}(t, r, i^*, p))$.

The function $\texttt{inLocRIB}(t, r, p)$ computes the neighbor i^*. $\texttt{inLocRIB}(t, r, p)$ first passes all announcements in the $\texttt{adjRIBsIn}(t, r, i, p)$ to the *imp* program, and then determines the neighbor with the "best" imported announcement.

2. Generalize Best Neighbor Note that the `inLocRIB` function still depends on the trace t. This step eliminates this dependence.

Consider the announcement a_l^* selected by the router r. A router always chooses the announcement a_l^* which has *pref* greater or equal to the *pref* of any other received and imported announcement a_l.

Therefore, if Bagpipe verifies that a specification holds for all received and imported announcements that have greater or equal *pref* than a_l, no matter from which neighbor they were received, the specification also holds for the announcement a_l^* received from neighbor i^*.

This step applies this insight by strengthening *specHolds* with the fact $pref(imp(r, i, p, a_i)) \leq pref(a_l^*)$, and then generalizing i^* to eliminate the dependence on t.

For those readers familiar with the BGP specification, note that Bagpipe fully supports tie-breaking on announcements with equal *pref*, e.g. tie-breaking on MED, and OSPF cost. This support stems from the fact that a specification must hold for all announcements with equal *pref*. This means that Bagpipe may falsely claim that a specification does *not* hold for a selected announcement a_l^* which can actually never be selected because a_l^*'s OSPF cost is higher than that of all other announcements, but will never falsely claim that a policy holds.

3. Generalize Received Announcements We are finally ready to apply the insight mentioned in the beginning of this section. This step replaces *specHolds*'s use of received announcements in the `adjRIBsIn`, which requires quantification over all traces, with any announcements that can be transmitted in the empty network. Note that there are two received announcements a_i and a_i^* that need consideration.

An announcement a received by router r from neighbor i for some prefix p is transmittable in the empty network $transmittable(r, p, i, a)$ in two ways. Either a is the initial value *na* stored in r's `adjRIBsIn`, or a_i is the result of transmitting some original announcement a_0 through the initial network along some path $\xi : path(i, r)$ that ends in router i followed by router r.

$transmit(\xi, p, a_0)$ computes the announcement that results from forwarding an announcement a_0 for prefix p along some path ξ in the initial network absent of any other announcements, i.e. it applies the appropriate *imp* and *exp* programs of every router along ξ to a_0.

Bagpipe exhibited no false positives in our evaluation. One reason is that modulo *pref* tie-breaking, the initial network reduction is *complete* for specifications that only depend on either a_i or a_i^* (but not both), because if either a_i or a_i^* can be received by r via the initial trace, then by definition, there exists a trace via which r receives either a_i or a_i^*. Examples

of specifications that depend on only a_i or a_l are *NoMartian* and *BlockToExternal*. If a specification depends on both a_i and a_i^*, it is possible that no trace exists that forwards both a_i and a_i^* to r, but we have not observed such cases in practice.

The goal of the initial network reduction was to eliminate *specHolds*'s quantification over any infinite sets, specifically the set of all traces. The initial network reduction achieves this goal, because INR only quantifies over finite sets. The set of prefixes P is large but finite. The set of announcements A is large but finite, as the BGP specification restricts the maximal announcement size to 4096 bytes. The set of routers R_i, $in(r)$, and $out(r)$ are also finite.

For ASes in a full-mesh configuration, meaning that each internal router is directly connected to all other internal routers[4], the set of paths $path(i, r)$ is also finite. This follows from the fact that an internal router r either received announcement a_i from some external router i — in which case the path is $[i, r]$ — or from an internal router i (other than r) which in turn received the announcement from an external router r_e — in which case the path is $[r_e, i, r]$. The set of paths is thus:

$$path(i,r) := \{[i,r] \mid i \in R_e\} \cup \\ \{[r_e,i,r] \mid i \in R_i \setminus \{r\} \wedge r_e \in R_e \wedge r_e \in neighbor(i)\}$$

Routing loops inside the AS are impossible, as routers do not forward announcements received from internal neighbors to internal neighbors.

We refer to the formula resulting from the initial network reduction as $\text{INR}(\tau)$. Because $\text{INR}(\tau)$ only quantifies over finite sets, it enables automatic verification of *specHolds* using current solvers.

2.5 Bagpipe Implementation

Bagpipe verifies a specification τ by searching for counterexamples to $\text{INR}(\tau)$. While this search space is finite, it is still far too large to permit naive brute-force search. This section describes how Bagpipe searches this space efficiently.

2.5.1 Symbolic Search using Rosette

Bagpipe uses Rosette [82] to symbolically search for counterexamples. Rosette extends the Racket language with symbolic values, assertions, and a `verify` function. `(verify e)` attempts to assign a concrete value to every symbolic value in e such that an assertion in e is violated. Rosette implements `verify` by reducing the search for a failure-inducing assignment to a satisfiability query, which is then discharged by an off-the-shelf SAT or SMT solver. Once the solver returns, its output is automatically lifted to a concrete value for each symbolic value. These concrete values are then used to provide counterexamples.

[4]Some large ASes avoid the performance penalty of a full-mesh configuration by using *route reflectors*, routers that exist to propagate messages between multiple connected components of an AS's topology. Bagpipe does not currently support this optional extension of the BGP specification.

```
(define (bagpipe τ) (verify (begin
  (define r (symbolic R_i))
  (define p (symbolic P))
  (define i i* (symbolic (in r)))
  (define o (symbolic (out r)))
  (define a_i a_i* (symbolic A))
  (define a_l (imp r i p a_i))
  (define a_l* (imp r i* p a_i*))
  (define a_o (exp r o p a_l*))
  (assert
    (implies
        (transmittable r p i a_i) (transmittable r p i* a_i*)
        (≤ (pref a_l) (pref a_l*))
      (τ {router := r; prefix := p; sender := i; received := a_i;
          selected := a_l*; bestSender := i*; bestReceived := a_i*;
          receiver := o; sent := a_o}))))))
```

Figure 2.8: Bagpipe implementation in Rosette. The core of Bagpipe is a translation of INR(τ) to a program `(begin ...)`. This program uses symbolic variables instead of universal quantification. Rosette's `verify` function attempts to assign a concrete value to every symbolic value in the program such that an assertion in the program is violated. If no such assignment is found, INR(τ) is valid, and the specification τ holds.

Figure 2.8 shows the implementation of Bagpipe in Rosette. The core of Bagpipe is a translation of INR(τ) to a Rosette program. The universally-quantified variables in INR(τ) are translated to symbolic values, which are used as inputs to the assertion that τ holds over all valid pairs of announcements.[5]

Users of Bagpipe express specifications in Racket as boolean predicates over active announcements. For example, a user would express the *NoMartian* specification from Section 2.3 as follows:

```
(define NoMartian (v)
  (implies (martian (prefix v))
           (= (selected v) na)))
```

In Bagpipe, routers are configured using the *imp* and *exp* programs. In practice, routers are configured using router configuration languages; most real-world routers use languages developed by Juniper and Cisco. An *interpreter* bridges the gap between Bagpipe and real-world configurations; it is a program that takes a router configuration and inputs (e.g. router, prefix, announcement) and returns the result (an announcement) of running the router configuration. Bagpipe includes interpreters for Juniper and Cisco configurations. These interpreters consist of a parser that generates an AST, and an execution engine that can run

[5]The translation of *transmittable* into Rosette is omitted due to space reasons, but it is straightforward.

an AST given some inputs. Rosette lifts the execution engine to a symbolic execution engine that can run an AST symbolically on all inputs. Bagpipe also infers the network topology of the AS from router configurations.

Bagpipe's interpreters skip commands unrelated to BGP (e.g., configuration commands for other protocols including IGMP, MPLS, and ISIS), and ignore low-level BGP configuration details (e.g., maximum update message TCP packet size). This could introduce unsoundness (e.g., if an AS operator accidentally configured maximum TCP packet size to be 0, then all update messages would be dropped). The interpreters also currently do not handle some BGP-related commands, such as IP broadcasting.

In our experiments, we found that Bagpipe's current interpreters are sufficient to handle hundreds of thousands of lines of industrial BGP configurations. Extending Bagpipe's interpreters to support additional BGP-conforming features or configuration languages requires no changes in the main algorithm, because Bagpipe models import and export rules as arbitrary functions. Such extensions may however require substantial engineering efforts in the interpreters, as existing configuration languages are often proprietary and contain a vast number of features.

2.5.2 Predicate Abstraction

The set of possible announcements is finite, since BGP restricts the maximal size of announcements to 4096 bytes. Even when represented symbolically, this is still a large search space. Therefore, Bagpipe also implements a form of predicate abstraction [27] by coalescing *aspath* and *communities* values that induce the same control flow in the BGP configuration.

Bagpipe exploits the fact that the Juniper and Cisco configuration languages use regular-expression predicates to branch on announcement attributes. The following example shows two such predicates contained in the Internet2 configurations:

```
as-path PRIVATE ".* (64512-65535) .*";
community LHCONE-DO-NOT-ANNOUNCE-AS members 65010:*;
```

`PRIVATE` matches all AS-paths that contain at least one AS with an AS number in the range 64512 – 65535. `LHCONE-DO-NOT-ANNOUNCE-AS` matches all communities whose first 16 bits encode the number 65010. The set of predicates in a configuration is finite and usually fairly small. All configurations of Internet2 combined, for example, contain only 73 community predicates.

Bagpipe implements predicate abstraction by automatically discovering all regular-expression predicates used in router configurations, and replacing the *aspath* and *communities* data contained in an announcement with a bit-vector that contains a bit for every predicate over *aspath* or *communities*. These bit-vectors are represented symbolically during Bagpipe's search for counterexamples.

This approach is sound, but incomplete. Consider for example a configuration with two predicates: predicate Φ matches every *aspath*, and predicate ϕ matches exactly one specific

aspath. Bagpipe would explore a branch on $\Phi(a) \wedge \neg\phi(a)$ which cannot be executed in reality because for every a, $\Phi(a)$ implies $\phi(a)$. We did not see such false positives in our evaluation.

2.5.3 Parallelization through Hoisting

Bagpipe hoists certain symbolic values out of the program passed to Rosette's symbolic search, and enumerates them instead of representing them symbolically. For example, Bagpipe hoists `(symbolic` R_i`)` out of the symbolic execution, manually enumerating the set R_i instead. Rosette is then invoked for each enumerated value.

```
(define (bagpipe τ) (for/each (r R_i)
  (verify (begin ...r...))))
```

Bagpipe parallelizes the resulting loop across multiple nodes in a cluster. Bagpipe hoists all variables except for announcements and prefixes, i.e. r, o, i, i^*, and some variables in *transmittable*. Bagpipe hoists these sets because their relatively small size and minimal structure are not exploitable by the symbolic execution engine.

The number of calls made by Bagpipe to Rosette can be computed by considering the magnitude of all of the sets that Bagpipe has to enumerate. To verify a policy, Bagpipe has to enumerate every internal router r, all the neighbors to which r can forward an announcement a_o, and all the paths along which the two received announcements a_i and a_i^* could have been transmitted to r.

The number of neighbors to which r can forward announcement a_o is $|out(r)|$. There are three kinds of paths along which an announcement could have been transmitted to router r. (1) The announcement was directly transmitted from one of r's external neighbors e to r. There are $n(r) = |\{e \in R_e | e \in neighbor(r)\}|$ such paths. (2) The announcement was transmitted from an external neighbor through an internal neighbor i to r. There are $\sum_{i \in R_i \setminus \{r\}} n(i)$ such paths. (3) The announcement is the value na with which the `adjRIBsIn` for an incoming neighbor was initialized. There are $|in(r)|$ such cases. The total number of calls Bagpipe makes to Rosette are thus:

$$\sum_{r \in R_i} |out(r)| \left(|in(r)| + n(r) + \sum_{i \in R_i \setminus \{r\}} n(i) \right)^2$$

Assuming that an AS has at least one external neighbor, the asymptotic number of Rosette calls made by Bagpipe is $O(|R_i|^4 |R_e|^3)$, where R_i is the set of internal routers and R_e is the set of all external neighbors. This stems from the fact that $O(|out(r)|) = O(|in(r)|) = O(|R_i| + |R_e|)$, and $O(n(r)) = O(|R_e|)$.

2.5.4 Omitting Unused Specification Arguments

There are specifications that do not use all of their arguments; *BlockToExternal*, for example, uses neither the *sender* argument nor the *received* argument (whose value depends on *sender*).

Enumerating these unused arguments would result in many calls to Rosette which are guaranteed to be equivalent. To avoid this duplication, Bagpipe does not enumerate certain combinations of unused arguments, thereby reducing the number of calls to Rosette. In practice, we found that useful specifications use one of the following three combinations of unused variables (all supported by Bagpipe):

- Bagpipe does not enumerate *sender* and *receiver* if arguments *sender*, *received*, *receiver*, and *sent* are unused. This is the case for all specifications composed of only an import specification (e.g. the *NoMartian* specification). In this case, Bagpipe calls Rosette $O(|R_i|^2|R_e|)$ times.
- Bagpipe does not enumerate *sender* if arguments *sender* and *received* are unused. This is the case for all specifications composed of only an export specification (e.g. the *BlockToExternal* specification). Bagpipe calls Rosette $O(|R_i|^3|R_e|^2)$ times.
- Bagpipe does not enumerate *receiver* if arguments *receiver* and *sent* are unused. This is the case for all specifications composed of only a selection ranking (e.g. the *GaoRexford* specification). Bagpipe calls Rosette $O(|R_i|^3|R_e|^2)$ times.

2.6 Evaluation

This section evaluates Bagpipe by answering the following questions. *Expressiveness:* Can Bagpipe specify policies for common configuration scenarios? *Efficiency:* How long does Bagpipe take to verify specifications? *Effectiveness:* How many bugs does Bagpipe find, and how many false positive does Bagpipe produce?

2.6.1 Juniper TechLibrary Scenarios

We evaluated Bagpipe on 10 configuration scenarios from the Juniper TechLibrary. Specifically, we evaluated Bagpipe on 10 scenarios described in the *Basic BGP Configuration* [9] and *Configuring Routing Policies* [40] sections of the Juniper TechLibrary, which is the official technical documentation for Juniper products. We used all scenarios from the *Basic BGP Configuration*, but did not use 25 scenarios from *Configuring Routing Policies* because: 5 scenarios require extensions or optional features of the BGP specification RFC 4271 that are inconsistent with Bagpipe's model of BGP (specifically exporting routes that are not selected, and delaying selection of announcements to reduce oscillation), 5 scenarios are unrelated to BGP verification (e.g., logging the number of forwarded announcements), and 15 scenarios require currently unsupported Juniper features (e.g., policy subroutines) that we believe could be implemented in Bagpipe without changing Bagpipe's model of BGP.

Each scenario describes a set of AS operator objectives, and it provides router configurations for an entire example AS that achieves these objectives. For each scenario, we expressed a specification and verified it against the scenario's configurations. Bagpipe verified each specification in less than one minute.

The following list provides the name of each scenario, along with a description and code size of the specification that we expressed and verified (no line is longer than 80 characters):

1. *Configuring Internal BGP Peering.* All internal routers share their route announcements with all other internal routers (5 lines).
2. *Using AS Path Regular Expressions.* Routers block announcements whose AS path matches a given set of regular expressions (4 lines).
3. *Disabling Suppression of Route Advertisements.* Route announcements are sent back to the neighbor from which they were received (5 lines).
4. *Configuring Policy Chains and Route Filters.* The prefix of exported announcements matches the given chained route filters (10 lines).
5. *Configuring a Conditional Default Route Policy.* A default route is exported (3 lines).
6. *Configuring Communities in a Routing Policy.* Routers set appropriate local preferences according to an announcement's communities (8 lines).
7. *Rejecting Known Invalid Routes.* Known invalid routes are rejected (3 lines).
8. *Configuring External BGP Peering.* A router is connected only to external routers in a given set (4 lines).
9. *Configuring Routing Policy Prefix Lists.* The prefix of exported announcements matches the given prefix lists (12 lines).
10. *Using Routing Policy to Set a Preference Value for BGP Routes.* Any local preference set by external neighbors is replaced with a default value (3 lines).

To ensure Bagpipe detects specification violations, we also modified each scenario's configurations to violate its objectives. In each case, Bagpipe detected the violation in under a minute.

Section 2.3 showed that Bagpipe supports policies found in the literature, such as the Gao-Rexford model [25] and prefix-based filtering [56]. Section 2.6.2 shows that Bagpipe can also express policies inferred from real AS configurations.

2.6.2 Real AS Configurations

We inferred and verified specifications from the configurations of three ASes: the nation-wide ISP Internet2, the regional ISP BelWü, and the local ISP Selfnet. These configurations total over 240,000 lines of Cisco and Juniper code.

For each inferred specification, Figure 2.9 summarizes the time required to verify the specification, the number of searches that were performed by Rosette (which are performed in parallel), the arguments passed to the policy that are not used (see Section 2.5.4), and the number of import and export programs that violate the specification. *Bagpipe did not produce false positives on any benchmark.*

Timings are on Amazon EC2 with 2 instances of type `c3.8xlarge`, each with 32 virtual-cores and 60 GB of memory. The experiments ran for a total of 82h, the cost for which is about $30 using EC2 spot instances.

AS	Policy	Time	Solver Calls	Unused Arguments	Violations	False Positives
Internet2	*NoMartian*	1,178s (20min)	3,114	*sender* and *receiver*	0	0
Internet2	*NoMartian* (no checks)	1,194s (20min)	3,114	*sender* and *receiver*	N/A	0
Internet2	*BlockToExternal*	28,594s (8h)	115,330	*sender*	5	0
Internet2	*GaoRexford*	260,790s (72h)	971,680	*receiver*	14	0
BelWü	*NoMartian*	2,106s (35min)	1516	*sender* and *receiver*	N/A	0
BelWü	*TagMartian*	2,165s (36min)	1516	*sender* and *receiver*	0	0
BelWü	*RemoveOwnASN*	1,838s (31min)	1516	*sender* and *receiver*	0	0
Selfnet	*StaticExport*	2s	9	none	0	0

Figure 2.9: Real AS Configuration Case Study Results. *Solver Calls* is the number of calls to Rosette's solve function. *Unused Arguments* indicates the arguments passed to the policy that are not used. *Violations* is the number of specification violations found. Bagpipe did not issue false positives in any experiment.

Nationwide ISP: Internet2 The Internet2 AS connects educational, research, and government institutions spread throughout the US. We have access to the full configuration of Internet2's 10 BGP routers [36]. These routers are connected to 274 external neighbors. The configurations total 100,651 lines of Juniper code. We verified four policy specifications for Internet2, described below.

Internet2's configurations contain checks to block the import of announcements with martian prefixes. We thus inferred that it is Internet2's policy to never import martian prefixes. It takes Bagpipe 1,178s (20min) to verify that Internet2 correctly implements the *NoMartian* specification.

Internet2 contains dedicated sanity checks in 237 out of the 274 *imp* programs. After removing these sanity checks from the configurations, it takes Bagpipe 1,194s (20min) to check the *NoMartian* specification for Internet2 (indicated by *no checks* in Fig. 2.9). Because Internet2 performs a large variety of other checks on announcements, the specification continues to hold for 189 neighbors. This result implies that Bagpipe's ability to verify specifications is not only useful to increase confidence in the correctness of router configurations, but also to safely remove unnecessary sanity checks — 152 in this case.

From Internet2's configurations, it appears to be Internet2's policy not to forward an announcement to external routers if the announcement has the `BTE` community set. It takes Bagpipe 28,594s (8h) to check the *BlockToExternal* specification. The configurations of 5 neighbors do not adhere to the *BlockToExternal* specification. We notified Internet2 of these violations, but have not yet received a response.

Internet2 appears to operate according to a refined version of the Gao-Rexford model discussed in Section 2.3. We manually classified neighbors either as *customer* or *peer* using Internet2's pricing structure [37] and comments found in the configurations. None of Internet2's neighbors is a *provider*. We refined the usual *GaoRexford* specification to support Internet2's advanced policies, such as blocking announcements for invalid prefixes and allowing both customers and peers to influence Internet 2's preference of an announcement by setting certain communities. For example, a peer can set the `HIGH_PEERS` community to

increase an announcement's preference. It takes Bagpipe 260,790s (72h) to check the refined *GaoRexford* specification. The configurations of 14 neighbors do not adhere to the refined *GaoRexford* specification. We have contacted Internet2 about these violations, but have not yet received a response.

Regional ISP: BelWü We have access to the BGP related configurations for three BGP routers[6] of the regional ISP BelWü [8]. These routers are connected to 300 external neighbors. The configurations total 143,657 lines of Cisco code.

At the time of our experiments, it was BelWü's policy to monitor all announcements for martian prefixes by tagging them with a particular community, and to use this monitoring information to evaluate the impact of martian filtering on BelWü's existing routes. Because of good results, BelWü is planning to enable strict martian filtering in the near future.

Bagpipe took 2,106s (35min) to check the *NoMartian* specification and, as expected, revealed that BelWü imports martian prefixes (from 268 neighbors). We also expressed the specification *TagMartian*, which requires BelWü to tag all announcements for martian prefixes. Bagpipe took 2,165s (36min) to verify that BelWü correctly implements *TagMartian*.

It is BelWü's policy to remove, from all received announcements, communities that contain the ISP's own AS number. We call this policy *RemoveOwnASN*. It takes Bagpipe 1,838s (31min) to verify that BelWü correctly implements *RemoveOwnASN*.

The configurations did not indicate BelWü's Gao-Rexford relationships (*customer*s, *peer*s, and *provider*s), and we did thus not verify a *GaoRexford* policy. Verifying such a policy would require 619,258 solver calls.

Local ISP: Selfnet We also analyzed the 66 lines of Juniper configuration for the sole BGP router of the local ISP Selfnet [70]. This router has only a single BGP neighbor, and the ISP's policy is to only export announcements with the prefixes that it actually owns. We call this policy *StaticExport*. Bagpipe took 2s to verify that Selfnet correctly implements *StaticExport*.

2.6.3 Potential False Positives

Bagpipe has three potential sources of false positives:

1) When two announcements are tied on local preference, BGP uses complex rules to choose between them. Bagpipe conservatively and soundly models tie breaking as non-deterministic choice, rather than completely modeling the details of lower-level protocols like OSPF that these rules use (Section 2.4).

2) Bagpipe uses predicate abstraction to represent announcements. Removing this source of incompleteness would dramatically increase the size of Bagpipe's search space (Section 2.5.2).

---

[6]Our experiments for these configurations are to demonstrate scaling, and assume that only these three routers are operating in the AS.

3) The initial network reduction forwards announcements in the initial network, which is sound but could lead to false positives (e.g., when two announcements can be individually forwarded in the initial network but interfere with each other in reality). We do not know how to eliminate this source of incompleteness while keeping the search space finite (Section 2.4).

We found, via manual inspection, that *none* of the counter examples returned by Bagpipe during our evaluation were due to false positives.

2.7 Related Work

In this section, we address related work in network configuration checking. We also briefly discuss software-defined networking and prior work on SMT-based tools.

Network Analysis ***rcc*** [21] is a tool to find bugs in BGP configurations, which has been adopted by many AS administrators. It attempts to find violations of route validity and path visibility by inferring inter-AS relationships from the configuration itself (the input to the tool is a set of configurations from all of the routers in an AS). Unlike Bagpipe, ***rcc*** does not provide strong guarantees about the checked configurations; there are both false positives and false negatives in the configuration errors flagged by the tool.

Batfish [22] is a Datalog-based network configuration analysis tool. Router configurations, topology descriptions, and a particular set of received BGP announcements are translated into Datalog facts which are processed by Datalog rules to generate routing-tables. Z3 is then used to verify first-order properties over the generated routing-tables. Bagpipe and Batfish make different design decisions, and are thus able to verify different properties. Bagpipe verifies a restricted set of routing-table invariants (described in Section 2.3.2) with respect to *any* set of received announcements, whereas Batfish verifies arbitrary first-order logic formulas over routing-tables with respect to a particular set of BGP announcements. Great care has been taken in Bagpipe that all invariants can be translated to SAT formulas which can be decided in exponential time, whereas Batfish's formulas are generally undecidable.

Header Space Analysis (HSA) [41] verifies a data plane's packet forwarding behavior in a similar way to Bagpipe (which verifies a control plane's announcement forwarding behavior). Contrary to Bagpipe, HSA uses a custom symbolic search algorithm instead of an SMT solver; and HSA verifies specifications that restrict the forwarding of packets independent of any other packets in the network (like Bagpipe's import/export specifications), but cannot verify specifications that restrict the forwarding of packets depending on other packets in the network (like Bagpipe's selection rankings).

BGP Simulation C-BGP [64] is a BGP simulator. Given a topology and a set of configurations, it determines how traffic will be routed. AS operators can use it both for debugging existing problems and for testing potential new configurations. C-BGP and Bagpipe are potentially complementary; an AS operator could test configurations using C-BGP and then

verify them using Bagpipe to guarantee that the network is configured to correctly handle any set of received path announcements.

SDN Software defined networking is a new paradigm for local networks in which router configuration is controlled by a single program running on a master router. There has been a large amount of work on verifying the behavior of software-defined networks, including language support [57, 1], model-checking [3, 18], and full formal verification [31]. SDN has thus far not been used to control BGP-speaking border routers, but even if current BGP configuration languages are supplanted by SDN, tools like Bagpipe will still be useful to ensure that configurations respect AS policies.

SMT-Based Tools SAT and SMT solvers have been applied in a wide range of automated tools for bug finding [15, 11, 19], verification [45, 46, 78], program synthesis [76, 42], and fault localization [39]. Bagpipe builds on Rosette [81], a programming language designed for easy creation of such tools. In particular, Rosette is equipped with a symbolic compiler that can efficiently reduce a verification, synthesis, or fault localization query about all bounded executions of a program to an SMT formula. Because the initial network reduction enables Bagpipe to treat BGP configurations as finite programs, we can use Rosette's bounded reasoning facilities for sound and scalable verification of BGP policies.

2.8 Conclusion

We presented Bagpipe, a tool to automatically verify that router configurations correctly implement AS operators' BGP policies. To make this verification possible, we introduced the *initial network reduction*, which reduces verification of BGP policies from checking an infinite set of traces to checking a finite set of initial traces, thus enabling Bagpipe to use current constraint solvers effectively. Building on this reduction, the Bagpipe implementation additionally employs a novel combination of search space partitioning, unused variable omission, symbolic execution, and predicate abstraction. We evaluated Bagpipe on 10 configuration scenarios from the Juniper TechLibrary and three ASes with a total of over 240,000 lines of configuration, finding that Bagpipe will scale to the complexity and scope of real-world AS configurations.

Future work will extend Bagpipe to support additional BGP features (e.g. route reflectors), incrementalize verification for configuration updates, and use Rosette's synthesis features to automatically generate configurations that correctly implement policies.

Chapter 3

BGP SEMANTICS

3.1 Introduction

The Internet is a collection of interconnected networks run by universities, corporations, regional ISPs, and nation-wide ISPs. These networks, collectively known as Autonomous Systems (ASes), use the Border Gateway Protocol (BGP) to exchange route announcements. A route announcement describes a path that packets (e.g. sent via TCP) can take to travel across the Internet. Each AS configures its BGP routers to restrict how route announcements are used and exchanged. For example, to avoid unprofitable routes, an AS codifies its contracts with other ASes in its BGP router configurations.

Router configuration is a challenging and error-prone task. Large ASes maintain millions of lines of frequently-changing configurations that run distributed across hundreds of routers [36, 84]. Router misconfigurations are common and have led to highly visible failures affecting ASes and their billions of users. For example, in 2009, YouTube was inaccessible worldwide for several hours due to a misconfiguration in Pakistan [10]. In 2010 and 2014, China Telecom hijacked significant fractions of international traffic [17, 74, 54, 51]. Goldberg surveys several additional major outages and their causes [26]. Less visible is the high cost ASes pay every day to develop and maintain configurations.

Given BGP's vital role, there exist router configuration guidelines [25, 29, 12, 89], checkers that statically check for router misconfigurations [21, 22, 7], and simulators that dynamically check for router misconfigurations [64, 66, 59, 55]. They provide no guarantees about the absence of certain router misconfigurations, because they are based on simplified semantics of BGP or no semantics at all. For example, Gao & Rexford (GR) [25] provide configuration guidelines. These guidelines were hugely successful, as they formalized existing best practices in router configuration, provide monetary benefit to the individual ASes, and are proven to prevent Internet-wide BGP divergence. However, GR's guidelines do not achieve their goal in realistic scenarios, because their proof is based on a simplified BGP semantics that does not accurately model the route exchange within an AS.

To improve upon this situation, this chapter presents the first mechanized formal semantics of the BGP specification RFC 4271 [65]. In contrast to previous semantics [25, 29, 89], our semantics is fully formal (it is implemented in the Coq proof assistant) and models all required features of the BGP specification modulo low-level details such as bit representation of update messages and TCP [1].

---

[1] Our semantics is open-source, available online and in the supplemental material.

Three case studies show how to use our semantics as a basis for reliable proofs, checkers, and simulators that help BGP administrators avoid router misconfiguration.

1. Formalizing and extending Gao & Rexford's proof GR [25] proposed a set of guidelines for BGP router configuration, and they proved Internet-wide route convergence if these guidelines are implemented by every AS on the Internet. This is rightly one of the most celebrated results in networking, and the guidelines are widely followed.

We attempted to formalize GR's pen-and-paper proof in Coq and discovered that it is flawed: GR's original guidelines are not sufficient to ensure intra-domain convergence. Their proof makes simplifying assumptions about the BGP protocol. For example, every router has access to all the routes received by every router within the same AS; routes are not sent over a network but are instantly accessible whenever a router is "activated"; and route announcements cannot be withdrawn. All these assumptions are frequently violated in practice.

Proving convergence requires additional restrictions on BGP configurations. If every AS follows these extensions to GR's configuration guidelines, then convergence is guaranteed. Our proof is formal, mechanized, and uses our semantics of RFC 4271, which models both intra-domain and inter-domain routing and eliminates the aforementioned simplifying assumptions.

2. Verifying the soundness of the Bagpipe tool Bagpipe defines a declarative domain-specific language that enables BGP administrators to express control-plane specifications, such as "*my routers will never accept routes for invalid IP addresses*", "*my routers will always forward certain routes to other ASes*", and "*my routers will always prefer routes from customers over routes from providers*". Given a specification expressed in this language, Bagpipe automatically verifies that an AS's router configurations satisfy the given specification.

Although Bagpipe found 19 apparent errors in three ASes with over 240,000 lines of Cisco and Juniper BGP configuration, the Bagpipe tool itself had not been verified. Therefore, its results might not be trustworthy. Using our semantics, we found 2 bugs in Bagpipe and then formally verified that the fixed Bagpipe is sound, i.e. it will never falsely claim that an AS implements a specification. The proof led to the discovery of the Initial Network Reduction.

3. Testing C-BGP C-BGP [64] is a widely-used BGP simulator. It outputs a trace that captures all the route announcements exchanged by the routers in a simulated BGP network, and the routes installed in each router's routing information bases. We compared C-BGP against our semantics via differential testing [20] on BGP networks with randomly-generated topology, router configurations, and initial routes.

The tests revealed 2 bugs in which C-BGP violates the BGP specification. For example, C-BGP occasionally sends duplicate announcements even when the routes they are advertising have not changed. This is not permitted by Section 9.2 of the BGP specification, and it is therefore rejected by our semantics. Sending redundant messages can have serious consequences, e.g. it may impact BGP convergence times by triggering route flap dampening.

We reported this bug to the C-BGP maintainer, who acknowledged that the current behavior is incorrect and fixed the bug.

Validating the BGP semantics Creating a formalization is an intellectually challenging task, and it is all too easy to make mistakes during the process. Our case studies revealed errors in configuration guidelines, a checking tool, and a simulator, but might our semantics themselves be flawed?

Our proofs and case studies are useful in their own right, but they also validated that our formalism does not suffer from several common problems.

To prevent reasoning and modeling errors, we encoded it in the Coq proof assistant (Section 3.2). This guarantees that we did not make any unstated or incorrect assumptions. It also guarantees that there are no missing steps or outright errors in the proofs.

To show that the model is sufficiently detailed, we used it to prove the correctness of important BGP-related properties. In particular, we reasoned about the correctness of protocols (Section 3.3) and we reasoned about the correctness of tools (Section 2.4). Our model captures all the aspects of BGP that are necessary for both of those scenarios.

To show that the model accurately reflects reality and is sufficiently complete, we compared it against another BGP implementation — the C-BGP simulator. Except for the C-BGP bugs, our model behaves the same as the C-BGP simulator over a test suite (Section 3.5). This shows that the proofs in the previous sections did not succeed degenerately; for example, if the model had incorrectly forbidden all forwarding, then it would be vacuous to prove that no invalid announcement is forwarded.

Contributions This chapter's contributions include

- The first formal semantics of BGP, which simplifies and generalizes the informal RFC 4271.
- Three case studies that rectify BGP configuration guidelines, verify a BGP checker, and test a BGP simulator.
- Verification techniques, such as a library for co-inductive sequences, parameterization techniques, and integration of formal semantics with tests.

3.2 BGP Semantics

This section presents the first formal semantics of the BGP specification RFC 4271 [65]. RFC 4271 is 104 pages long and suffers both from underspecification (ambiguities) and overspecification (detailed procedural definitions).

Our semantics is formalized in 287 non-comment lines of Coq code. Our specification may be simple, but it was hard to achieve that simplicity! It would have been infeasible and counterproductive to formalize RFC 4271 as written — it contains too much detail. Rather, we iteratively refactored it, introducing some new concepts such as an *injected* neighbor and *na* values. Simply ignoring details of the RFC can introduce a loophole and the potential

for error. Instead, we abstracted away many details. This approach not only simplified our model of RFC 4271, but it also yielded a pluggable architecture that accommodates both official and vendor-specific extensions to BGP. Our case studies informed our selection of details to abstract, because the wrong abstraction prevents proofs.

Our semantics models the full BGP specification (RFC 4271) except for low-level details (bit representation of update messages, version negotiation, establishing BGP connections, and TCP). It models some extensions, such as the communities attribute and route reflectors. It does not model all optional features, such as route aggregation. We believe that our semantics could be extended with these additional features (see Section 3.2.3).

This section primarily describes our formalism, but notes a few of the places where its organization differs from the specification.

3.2.1 Network Semantics

The Internet consists of a network of routers that forward data packets toward their destination IP addresses. Routers announce routes — a path through other routers to a destination — by sending BGP *update messages* to one another. An update message means "I can forward packets to the following destinations." To a first approximation, a router sends an update message to its neighbors if it can directly deliver packets to a certain destination, or if it has learned of a new route by receiving a route announcement via an update message. The rules governing this behavior, which is controlled by router configurations, are made more precise in Section 3.2.1.

A router's *control plane* selects and forwards routes. The control plane runs separately and asynchronously from the router's *data plane*, which forwards packets using the routes selected by the control plane. BGP operates on the control plane. Our semantics therefore models only the control plane, not the data plane.

Topology

The top part of Fig. 3.1 shows how our semantics represents the network topology.

The Internet's routers are operated by *Autonomous Systems* (ASes) such as universities, corporate networks, regional ISPs, and nationwide ISPs. ASes are identified by globally-unique AS numbers (ASNs).

A router is represented as an IP address. A set of IP addresses with the same first $n \in [0, 33)$ bits is called a *prefix*. Each BGP router r is connected to a set of other BGP routers called r's neighbors. BGP connections are symmetric, i.e., if r is a neighbor of r', then r' is also a neighbor of r. A dummy neighbor called *injected* "injects" new routes, such as to destinations the router is directly connected to.

The function *md* assigns a mode to every connection. Connections between routers owned by the same AS are in *ibgp* (internal BGP) mode, and connections between routers owned by different ASes are in *ebgp* (external BGP) mode.

$IP := [0,256) \times [0,256) \times [0,256) \times [0,256)$ — IP addresses
$P := IP \times [0,33)$ — prefixes (sets of IP addresses)

$R \subseteq IP$ — routers
$ASN : Type$ — AS number
$asn : R \to ASN$ — router's AS number

$C \subseteq R \times R$ — connections between routers
$in(r) := \{s \mid (s,r) \in C\} \cup \{injected\}$ — r's incoming connections
$out(r) := \{d \mid (r,d) \in C\}$ — r's outgoing connections
$md(s,d) =$ `if` $asn(s)$=$asn(d)$ `then` $ibgp$ `else` $ebgp$ — link mode

$A = \{$ — BGP attributes
  $aspath : list(ASN)$; — AS path
  $pref : uint32$; — local preferences
  $communities : \mathcal{P}(uint32)$; — communities ($\mathcal{P}$ is powerset)
  ...
$\} \cup \{na\}$ — used when there is *no available* route to a prefix

$M = P \times A$ — update message

Figure 3.1: General Semantics Definitions.

Update Messages

A BGP router announces a route to another router by sending an *update message* (RFC 4271, §4). Each update message contains the set of destination IP addresses p for which the route is being announced, as well as attributes a that provide additional information about the route. The attributes (RFC 4271, §5) include 3 fields that are relevant to us. (1) An AS *aspath* is the list of AS numbers of every AS traversed by the update message. For example, if a router for AS#67 receives an update from AS#3 stating, "Address i can be reached by traversing AS#3 then AS#14," then the router might send its own announcement stating "Address i can be reached by traversing AS#67 then AS#3 then AS#14." (2) A local preference *pref* is an integer that influences route selection. (3) A *community* is a 32-bit integer that is used by ASes to announce additional information about a route, but is uninterpreted by the BGP protocol. Our semantics is parametric over the attribute's fields, and thus can be extended to support additional fields as described in Section 3.2.3.

A new update message overrides any previous announcement from the same neighbor with the same set of destinations. If the new update message's attributes are set to *na* (not available), the old route is withdrawn. If the new update message's attributes are available (i.e., a record), the old route is withdrawn and replaced by the new route.

$N := (C \to \mathit{list}(M)) \times ((r : R) \to S(r))$ — network sate
$U : \mathit{Type}$ — uninterpreted router state
$S(r)$ `:= {` — router state
  `adjRIBsIn` : $\mathit{in}(r) \times P \to A$; — received messages
  `locRIB` : $P \to A$; — selected messages
  `adjRIBsOut` : $\mathit{out}(r) \times P \to A$; — sent messages
  `uninterpreted` : $P \to U$ — uninterpreted state
`}`

Figure 3.2: Network State. The network state N consists of link state and router state. Link state keeps track of update messages currently in-flight $\mathit{list}(M)$ for each connection C. Router state $S(r)$ keeps track of received, selected, and sent messages, as well as uninterpreted state U used to store other protocols' routing information.

Network State

BGP is a stateful protocol. Figure 3.2 describes how our semantics models the network state, which consists of link state Γ and router state Σ.

The *link state* $\Gamma = C \to \mathit{list}(M)$ keeps track of all the update messages currently in-flight in the network. BGP sends messages via a TCP connection, thus messages on a link from one router to another are ordered.

The state at each router r is called the *router state* $S(r)$. Each BGP router remembers all the update messages that it has received; which route it has chosen to perform packet forwarding on the data plane when two of its neighbors each offered a route to the same destination; and which routes it has advertised to its neighbors. This state is stored in three tables, known as *Routing Information Bases* (RIBs) (RFC 4271, §3.2). Withdrawn routes are removed from the RIBs. A BGP router also maintains *uninterpreted state* U, such as state from other routing protocols like OSPF.

The router state is updated every time that a router receives a new update message. Our semantics models this using the *handle* function (Section 3.2.2), which takes a router's state and returns the router's new state and the messages to be sent to the router's neighbors.

The `adjRIBsIn` field maps every incoming neighbor i (including *injected*) and prefix p to the attributes of the update message most recently received from r's neighbor i for prefix p. If a neighbor withdraws a route, or if no update message has yet been received for a neighbor and prefix (e.g., whenever the router is restarted), the `adjRIBsIn` maps that neighbor and prefix to *na* (not available). In practice, the vast majority of entries in any RIB map to *na*.

The `locRIB` field maps every prefix p to the attributes of the update message selected by the router to perform packet forwarding. The BGP protocol handles prefixes independently. If two prefixes overlap, then one is always contained inside the other. BGP may install routes for both prefixes, and the dataplane then chooses the most specific prefix (the one

$\boxed{trace(N)}$

$$\frac{\Gamma, \Sigma \rightsquigarrow \Gamma', \Sigma' \qquad trace(\Gamma', \Sigma')}{trace(\Gamma, \Sigma)} \text{ STEP}$$

$\boxed{N \rightsquigarrow N}$

$$\frac{}{\Gamma, \Sigma \rightsquigarrow \Gamma, \Sigma} \text{ SKIP}$$

$$\frac{\begin{array}{cc}(s,r) \in C & \sigma = (\texttt{adjRIBsIn}(\Sigma[r], s, p) := a) \\ \Gamma(s,r) = (p,a) :: ms & \Gamma', \sigma' = handle(r,p,\sigma)\end{array}}{\Gamma, \Sigma \rightsquigarrow \Gamma[(s,r) := ms] \mathbin{++} \Gamma', \Sigma[r := \sigma']} \text{ FWD}$$

$$\frac{\begin{array}{cc}& \sigma = (\texttt{adjRIBsIn}(\Sigma[r], injected, p) := a) \\ (p,a) \in M & \Gamma', \sigma' = handle(r,p,\sigma)\end{array}}{\Gamma, \Sigma \rightsquigarrow \Gamma \mathbin{++} \Gamma', \Sigma[r := \sigma']} \text{ INJ}$$

$$\frac{\begin{array}{cc}& \sigma = (\texttt{uninterpreted}(\Sigma[r], p) := u) \\ (p,u) \in P \times U & \Gamma', \sigma' = handle(r,p,\sigma)\end{array}}{\Gamma, \Sigma \rightsquigarrow \Gamma \mathbin{++} \Gamma', \Sigma[r := \sigma']} \text{ UPD}$$

Figure 3.3: Network Semantics. *handle* implements a router's message-forwarding logic (see Fig. 3.4), and returns new messages to be sent Γ' plus the router's updated state σ'. $\Gamma \mathbin{++} \Gamma'$ is a notation for point-wise list append, defined as $(\lambda c.append(\Gamma(c), \Gamma'(c)))$. The notations $(\texttt{adjRIBsIn}(\sigma, i, p) := a)$ and $(\texttt{uninterpreted}(\sigma, p) := u)$ both return a copy of state σ, with `adjRIBsIn` and `uninterpreted` updated respectively.

that represents fewer IP addresses), even if the other route is more preferable according to the router's cost function.

The `adjRIBsOut` field maps every outgoing neighbor o and prefix p to the attributes of the update message most recently sent to r's neighbor o for prefix p.

Our semantics is parametric over the `uninterpreted` state U, but it is intended to store the routing information needed by other protocols.

Traces

Our semantics models an execution of the BGP protocol as a coinductive sequence of network state transitions (a trace), starting from some initial network state. The BGP protocol might

never converge — for instance, update messages might be sent in an infinite loop around a network. A trace is thus modeled as a coinductive infinite sequence of transitions.

BGP routers can only perform a fixed set of actions to *transition* from some network state (Γ, Σ) to the next network state (Γ', Σ'). Our semantics models these transitions with the $(\Gamma, \Sigma) \rightsquigarrow (\Gamma', \Sigma')$ relation defined in Fig. 3.3. Most transitions invoke the *handle* function, which implements a router's message-forwarding logic (described later in Fig. 3.4).

Skipping The SKIP transition models cases where the BGP protocol converges and eventually stops sending messages; the rest of the trace contains infinitely many SKIP events.

Forwarding The FWD transition marks a point in an execution of the BGP protocol when a BGP router r has received a route announcement via an update message $m = (p, a)$ from some neighbor s. This is only possible if, at the beginning of the transition, the link state for the connection between s and r is a list with m as its first element. The router r first installs the received message in the `adjRIBsIn` for its neighbor s, and then processes this new route by calling the *handle* function. The message m is removed from the link that it was received from. For each connection c, the new messages for c are appended to the existing messages for c in Γ.

Injection Some BGP routers know how to directly deliver packets for a destination prefix p. In practice, routers learn of this fact either via static configuration, or via some other network protocol such as OSPF [16]. RFC 4271 is vague in its description of how BGP routers should implement injected routes (RFC 4271, §9.4). Our semantics models it using the dummy *injected* neighbor and the INJ transition, which is largely similar to the FWD transition.

Note that *any* route can be injected, and any injected route can be withdrawn with *na*; *handle* provides a configuration hook to filter unwanted injected routes. The proofs are kept simple by re-using an existing element in the formalism rather than creating a new mechanism.

We considered several alternatives to this design. Adding injected routes only on startup wouldn't handle run-time discovery of routes via IGP. Adding injected routes as selected routes (that is, to `locRIB` rather than to `adjRIBsIn`) would permit them to be overwritten by newly-arriving routes for the given prefix, or would need a "sticky" bit to prevent that problem.

Uninterpreted State Update The UPD transition marks a point in an execution of the BGP protocol when a BGP router r changes its uninterpreted state for prefix p. The router r then reprocesses all the routes of prefix p by calling the *handle* function. Storing a router's uninterpreted state for each prefix, instead of having just one uninterpreted state per router, has the benefit that a router can avoid reprocessing update messages for all other prefixes, if the update will only affect processing for a subset of prefixes.

$imp : (r : R) \to in(r) \to P \to A \to A$ — configure import
$exp : (r : R) \to in(r) \to out(r) \to P \to A \to A$ — configure export
$dec : (r : R) \to U \to (in(r) \to A) \to in(r)$ — select message

$handle : (r : R) \to P \to S(r) \to ((C \to list(M)) \times S(r))$
$handle(r, p, \sigma) :=$
  `let` $\sigma_i := \texttt{adjRIBsIn}(\sigma)$
      $I := \lambda i.imp(r, i, p, \sigma_i(i, p))$
      $i^* := dec(r, \texttt{uninterpreted}(\sigma, p), I)$ — selected neighbor
      $a^* := I(i^*)$ — selected attribute
      $\sigma_l := \texttt{locRIB}(\sigma)[p := a^*]$ — new state's `locRIB`
      $\sigma_o := \lambda(o, p').$`if` $p' = p$ `then` $exp(r, i^*, o, p, a^*)$
                  `else` $\texttt{adjRIBsOut}(\sigma, o, p')$
      $\Gamma := \lambda(s, d).$`if` $s = r \wedge \texttt{adjRIBsOut}(\sigma, d, p) \neq \sigma_o(d, p)$
                `then` $[(p, \sigma_o(d, p))]$ `else` $[]$
  `in` $(\Gamma, \{\texttt{adjRIBsIn} := \sigma_i; \texttt{locRIB} := \sigma_l; \texttt{adjRIBsOut} := \sigma_o;$
        $\texttt{uninterpreted} := \texttt{uninterpreted}(\sigma)\})$

Figure 3.4: Router Semantics. *handle* defines how a router processes attributes stored in its Adj-RIBs-In. First, r imports all attributes from the `adjRIBsIn` σ_i. Second, r chooses the best imported attribute a^* from neighbor i^* and stores it in the `locRIB` σ_l. Third, r exports a^* to all its neighbors, storing the result in its `adjRIBsOut`.

3.2.2 Router Semantics

This section describes how a BGP router processes routes in its Adj-RIBs-In.

Router Processing

The BGP specification (RFC 4271, §9) requires a router to execute several steps in response to a change of its Adj-RIBs-In, e.g., because of an incoming message. The router's configuration customizes some of these steps. The steps are:

1. The configurable *import* step modifies the attributes of each received message for the given prefix, resulting in imported messages. This step can, for example, filter incoming messages with invalid prefixes.
2. The *decision* step selects a single message from all imported messages for the given prefix. This step can, for example, prefer routes through paying neighbors over routes through neighbors that expect to be paid.
3. The configurable *export* step modifies the selected attributes for the given prefix, resulting in an exported message. This message is sent to the router's neighbors. This step can, for example, block update messages to competitors.

These steps are implemented as follows by the *handle* function (Fig. 3.4). $handle(r, p, \sigma)$ is invoked whenever router r with state σ receives an update message for prefix p.

1. Attributes are imported with a call to the *imp* function, which takes the current router r, prefix p, and neighbor i, as well as the attributes stored in σ_i for that neighbor and prefix, and returns possibly-modified attributes. The *imp* function is one of BGP's hooks for customization, defined in a router's configuration. The function is usually written in either the Juniper or Cisco configuration language, which are loop-free imperative programming languages with domain-specific syntax and semantics. Our semantics is parametric over the particular language used, and just requires that the configuration language can be denoted to a mathematical function. The advantages of this approach are described in Section 3.2.3. An implementation of a router can avoid re-importing old announcements by caching the result of old announcement imports. The *imp* function can discard a message by modifying the message's attributes to *na*. For example, to discard messages for invalid prefixes, a router can be configured with the following *imp* function: $\lambda r\, i\, p\, a.\, \texttt{if}\ \mathit{invalid}(p)\ \texttt{then}\ \mathit{na}\ \texttt{else}\ a$.

2. The *dec* function (Section 3.3.2) selects the incoming neighbor i^* whose imported update message attributes a^* should be used for routing packets on the data plane. Because *dec* depends on the router's uninterpreted state, it is necessary to reprocess the Adj-RIBs-In every time that the uninterpreted state changes. The router then stores the selected attributes in the Loc-RIB. In our semantics, instead of directly updating the state, the router returns a new state which is a modified copy of the original state.

3. The router exports the selected attributes to all its neighbors. The *exp* functions are configured in a similar fashion to the *imp*. The router then stores all the exported attributes in the Adj-RIBs-Out σ_o.

Finally, the router decides which update messages to send. We model the sent messages as a mapping from every connection (even those that are not connected to r, in order to simplify the formalism) to a list of sent messages. The router sends a message if the Adj-RIBs-Out σ_o for the destination neighbor d has just changed.

Configuration Restrictions

The assumptions and requirements on BGP routers and configurations are sprinkled throughout the 104-page specification and often written procedurally. A key reason for our success is that our semantics does not hard-code exactly the complex procedures in the specification (e.g., see Section 3.3.2), but instead we determined the emergent properties and formalized them.

We abstracted out some requirements of RFC 4271 (Fig. 3.5). A user of our semantics can enable or disable these requirements, or add new ones. This enables modeling specifics of a particular AS, such as its topology or which features are used in its BGP configuration files. It also enables modeling of routers that violate the specification, which is common. For example, the Juniper router configuration language used by Juniper-manufactured routers does not enforce restrictions 5 and 6, and allows arbitrary manipulations of the AS path.

1. $\forall r\ i\ p.\ imp(r, i, p, na) = na$
2. $\forall r\ i\ o\ p.\ exp(r, i, o, p, na) = na$
3. $\forall r\ i\ p\ a.\ md(i, r) = md(r, o) = ibgp \rightarrow exp(r, i, o, p, a) = na$
4. $\forall r\ i\ p\ a.\ asn(r) \in aspath(a) \rightarrow imp(r, i, p, a) = na$
5. $\forall r\ i\ o\ p\ a.\ md(r, o) = ebgp \rightarrow$
   $aspath(exp(r, i, o, p, a)) = asn(r) :: aspath(a)$
6. $\forall r\ i\ o\ p\ a.\ md(r, o) = ibgp \rightarrow aspath(exp(r, i, o, p, a)) = aspath(a)$

Figure 3.5: Rule Restrictions. BGP requires that the *imp* and *exp* rules create no attributes "out of thin air" (1,2), avoid forwarding loops (3,4), and extend paths appropriately (5,6).

The C-BGP router configuration language [64], unlike the specification, ensures that update messages are not sent back along the connection that they came from.

Rules 4–6 are required by the BGP spec but are not needed for our proofs. Therefore, our proofs hold even for routers or configurations that violate them.

3.2.3 Design Decisions

This section mentions design principles that we found essential in formalizing a complex specification like RFC 4271.

Parameterization Our BGP semantics is parametric over attributes (Section 3.2.1), attribute preference (Section 3.3.2), uninterpreted state (Section 3.2.1), and router configuration languages (Section 3.2.2). Our proofs hold for every possible parameter instantiation.

The attributes and preference are hard-coded in the BGP specification's definition of import/select. As noted above, these requirements are not satisfied by real-world implementations.

The uninterpreted state is primarily used for the OSPF (or some other IGP) protocol. The BGP specification requires that selection considers interior cost such as the number of hops, within an AS, between each pair of border routers. This suggests that formalizing BGP also requires formalizing all of OSPF, and other protocols that the specification relies upon — ignoring these, or making assumptions, would yield an unsound semantics. To avoid this problem, our semantics of BGP is parametric over the implementation of the IGP protocol. A proof requiring more detail can express additional requirements that are satisfied by any execution of IGP.

The router configuration languages are already parameterized in RFC 4271.

The fact that our semantics is parameterized has several advantages. 1) It *simplifies* the semantics' definition, e.g., we did not have to formally model all the languages used to configure routers in practice. 2) It enables *extending* the semantics; see immediately below. 3) It enables reasoning about *non-compliant* router implementations, e.g., actual router implementations have their own attribute selection algorithm.

We believe we have come up with an appropriate level of parameterization, and our case studies support this belief. Our semantics is specific enough to prove important properties. It is general enough to be easy to define and reason about, and to match realistic implementations that do not follow the specification precisely.

Our initial goal was only to support the BGP specification, RFC 4271, and we introduced parameterization to make that goal manageable. However, by parameterizing our semantics over uninterpreted state, attributes, selection, and import/export rules, our semantics supports several extensions to BGP.

(1) Communities (RFC 1997) are additional predefined BGP attributes. Our semantics supports communities by being parametric over attributes.

(2) Route Reflectors (RFC 4456) help an AS scale up to more routers. RFC 4271 requires full-mesh configuration — each router is connected to all other internal routers. Route reflectors are dedicated internal routers which serve primarily to forward announcements between other internal routers. This architecture reduces the number of connections required between routers. Our semantics supports router reflectors by being parametric over attributes, selection, and export.

(3) Confederations (RFC 5065) serve the same purpose as route reflectors: to reduce the number of required connections. Instead of having dedicated "repeater routers" (reflectors), they instead partition the AS into multiple sub-ASes (confederations) which are individually each in a full-mesh configuration and have connections between confederations to propagate routes throughout the larger AS. Our semantics supports confederations by being parametric over the same properties that enable route reflectors.

Modeling Networks as Distributed Systems Our semantics of BGP borrows from modeling techniques for distributed systems. We use traces to model the evolution of the BGP network, and we use modal logic to reason about these traces.

Like previous work on distributed systems, our semantics models the network using a small step semantics. However, instead of using a classical logic approach and modeling a step as a proposition which indicates whether a network can step from one state to another (as done in Verdi [98]), we use a constructive logic approach, and model a step as a type which contains an element for every possible way that a network can step from one state to another. Using this model, a trace is then just a sequence of elements in the step type.

The semantics of BGP requires the notion of an infinite trace. We thus model traces as co-inductive sequences of steps. Coq has automation support for reasoning about inductive types, but not for co-inductive types. Therefore, we created a small, reusable library that encapsulates the co-inductive definition of traces behind a simple interface inspired by the "eventually" concept from modal logic.

For example, in the GR proof, we frequently had to prove liveness properties, i.e., that a given trace t will establish a certain property P at some point in the future, or formally: there exists a natural number n, such that P holds for every step after the nth step. Instead of reasoning about liveness properties directly, and thus co-inductive traces, we hid the definition

of such liveness properties in a quantifier $eventually(P) = \exists n, \forall i > n, P(t[i])$, and then only reasoned using lemmas proven about eventually, like: $eventually(P) \wedge eventually(Q) \rightarrow eventually(\lambda s.P(s) \wedge Q(s))$.

An alternative to a co-inductive definition of infinite sequences would have been to define a trace as a relation from Nat to state, but that would have complicated other parts of the proof.

Unification of Concepts The BGP specification has no notion of a "not available" value *na*. Our semantics introduces this to unify multiple concepts from the BGP specification: uninitialized entries, when a RIB does not contain a route for a certain (neighbor and) prefix; withdrawal of a route by an update message or by another protocol such as OSPF; dropping of a route by the *handle* routine, whether on import or export. Proofs are simpler because there is no need to represent the concepts separately, and because the RIB tables are total.

3.2.4 Discussion

The direct outcome of this part of our work is the first mechanized semantics for BGP: a recasting of an imprecise English specification into an unambiguous, simple formalism implemented in Coq. Rather than merely transliterating; the formalism adds or changes some abstractions such as not-available and injection. Rather than hard-coding procedures that are given step-by-step in the spec, we identified key properties that they ensure and made the specification parametric over them. This enables these requirements to be relaxed at will, and it identifies which ones are really important for a particular proof. It also makes the semantics extensible.

Methodological contributions that can be applied to other verification tasks include our eventually library to handle modeling infinite traces as co-inductive sequences of states and the way that we abstracted out and parameterized over required properties.

3.3 Proof of Gao & Rexford's Guidelines

Unlike most other networking protocols, BGP has no built-in mechanism to guarantee network convergence: routers may indefinitely flip-flop between routes, wasting resources and failing to properly forward packets. A large number of papers have been published to address the problem of BGP divergence [29, 12, 91]. The most influential work is Gao & Rexford's (GR's) configuration guidelines [25], which state constraints on how a network operator should configure its BGP routers. These guidelines capture important intuitions about BGP, they formalized existing best practices in router configuration, and they provide monetary benefit to the individual ASes. There is another benefit as well: GR stated, and informally proved, a theorem that if every AS follows the guidelines, then all BGP routers in the Internet will converge to a steady state (in the absence of routers/links entering/leaving the network).

When we attempted to formalize the proof, we discovered that the theorem is false, and the manual proof rested on unfounded assumptions. In particular, it made the assumption

that the BGP protocol does not diverge within an AS, which Vissicchio et al. [87] have shown to be false.

The BGP specification might seem to prevent divergence, because if a router r receives a message from an internal router ir_1, the BGP specification forbids it from forwarding that message to another internal router ir_2. (There would be no need, since ir_1 is directly connected to ir_2.) It is reasonable to conclude that this rule exists to prevent intra-AS divergence, but if so the BGP specification authors failed in their aim and GR did not notice the failure.

This divergence can be prevented by adding a fourth guideline. Like the GR's original three guidelines, this forbids certain BGP configurations. Section 3.3.1 states all four configuration guidelines. We used the Coq proof assistant and our BGP semantics to formally prove that the four configuration guidelines are adequate to prevent divergence throughout the Internet. Compared to GR's original proof, ours uses a more realistic model without GR's simplifying assumptions, ours proves internal convergence, and ours states and proves new invariants about BGP behavior such as the Gliding Path Invariant.

This section is organized as follows. Section 3.3.2 states GR's configuration guidelines and the extension that prevents internal AS divergence. Section 3.3.2 formalizes the proof's assumptions about the BGP network's topology. Section 3.3.3 presents our formal proof that any BGP network implementing the guidelines will eventually converge. Section 3.3.4 compares our proof with the one provided by GR.

3.3.1 Configuration Guidelines

An AS x usually has one of three kinds of business relationships with each of its neighboring ASes y. 1) x is the customer of the provider y, and pays y for every packet sent between the two (in either direction). Customers are usually smaller ASes that pay the larger provider AS to get access to the entire Internet. 2) x is the provider of the customer y, and gets payed by x for every packet sent between the two. 3) x and y are peers that do not pay each other for any packet sent between the two.

GR's proof requires that every AS x configures all its routers to follow three guidelines.

Import Guideline: Routes from customers are preferred over routes from peers or providers. Formally, attributes imported by a router r from a customer i_c must have strictly higher preference than attributes a' imported by any router r', of the same AS, from a peer or provider i_p, i.e. $imp(r, i_c, p, a) > imp(r', i_p, p, a')$ ($>$ is defined in Section 3.3.2). This guideline not only aids the Internet's global convergence, it also makes financial sense to the individual AS. By preferring update messages from customers, an AS's routers will prefer to forward packets to paying customers, leading to increased revenue for the AS.

Export Guideline: Only routes involving customers are exported. Formally, a router r of x may only export update messages m that satisfy one of the following three conditions: m is being forwarded to a customer, or m was directly or indirectly received from a customer of x, or m was injected. Router r received a message from a customer indirectly, if some router r' of x received the message from a customer and subsequently forwarded it inside the

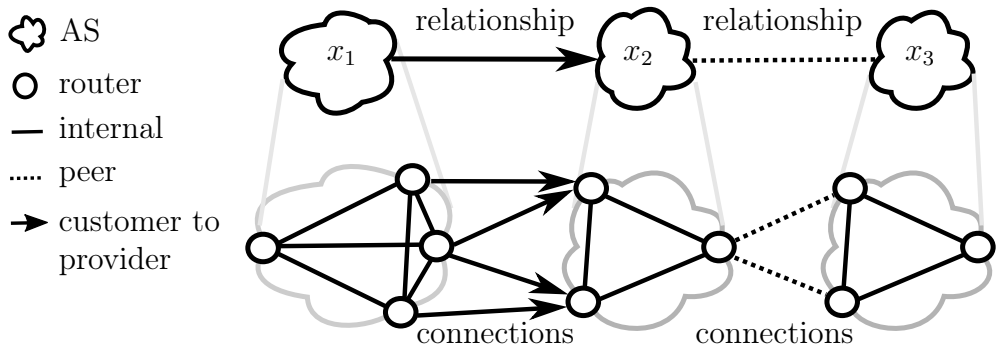

Figure 3.6: AS Router Endomorphism. Every AS has at least one router. The connections between routers of any two ASes, match the relationship between the ASes.

AS to the router r. This guideline not only aids the Internet's global convergence, it also makes financial sense to the individual AS. By blocking update messages that do not involve customers, the AS avoids establishing routes that do not involve customers, and thus avoids having to forward packets that neither the sender nor receiver is willing to pay for.

Injection Guideline: Injected routes are preferred over all other routes. Formally, AS x_0's injected attributes a_0 must have strictly higher preference than attributes a imported by any router r, of the same AS, from any external neighbor i for the same prefix p, i.e. $imp(r_0, injected, p, a_0) > imp(r, i, p, a)$. By preferring injected routes, and exporting them to all neighbors, an AS makes the route's prefix available outside the AS, which is usually the intended behavior.

We extend GR's guidelines with one more:

Internal Guideline: A router is required not to change the preference of attributes imported or exported internally. Formally, for any internal neighbor n of r, $imp(r, n, p, a) \simeq a$ and $exp(r, i, n, p, a) \simeq a$ ($\simeq$ is defined in Section 3.3.2). This guideline extension is crucial for our proof (see Section 3.3.3), but was overlooked by GR's original proof due to their simplified semantics of BGP. Vissicchio [87] discusses internal policy guidelines in detail.

3.3.2 Convergence Assumptions

This section formalizes GR's assumptions about the BGP network's topology. Unless otherwise noted, any assumption made by our proof is also an assumption made by GR's original proof.

Topology Restrictions

ASes have various business relationships with one another. This proof assumes one of exactly three relationships between any two ASes: *Customer to Provider*: the customer AS pays the provider AS for every packet sent between the two (in either direction). Customers are usually smaller ASes that pay the larger provider AS to get access to the entire Internet.

Peering: one AS neither pays nor charges the other AS for any packet sent between the two. The peering relationship is symmetric. *No Relationship*: the two ASes do not directly send packets from one to the other.

This proof assumes that the connections between routers of any two ASes match the relationship between the ASes. This is illustrated in Fig. 3.6. Formally, there is a endomorphism between the ASes with their relations, and routers with their connections. In other words: every router belongs to exactly one AS, and every AS has at least one router; any two ASes that are in a customer-to-provider relationship only maintain customer-to-provider connections between their routers, and there exists at least one such connection; any two ASes that are in a peering relationship only maintain peering connections between their routers, and there exists at least one such connection; any two ASes that are not in a relationship maintain no connections between their routers.

This proof assumes that each AS is in a full-mesh configuration: each router is directly connected to all other routers of the same AS. This simplifies the proof of convergence, but in practice some large ASes avoid the performance penalty of a full-mesh configuration by using route reflectors [6]. Route reflectors are an optional RFC extension, supported by our semantics but not by this proof.

This proof assumes that the customer-to-provider relationships form a Directed Acyclic Graph (DAG). See Fig. 3.10 for an example. This means that each AS can be assigned a level (its *AS level*), such that any AS is only a provider to lower-level ASes, and only a customer to higher-level ASes. This assumption is realistic. ASes on the lowest level are usually universities or companies that require a provider to forward and receive packets from the rest of the Internet, but are not themselves providers to any other ASes. On the second lowest level are Internet Service Providers like Internet2, which is a provider to many universities in the US, but is not large enough to route all prefixes without talking to a provider. On the highest level are ASes like (the cleverly named) Level3. These can route any prefix without talking to any provider. Our proof even holds for DAGs consisting of multiple disconnected components, though this is not usually the case for the Internet.

Lastly, this proof assumes that the customer-to-provider relationships are well-founded, i.e. every customer's direct or indirect provider can be reached via a finite number of ASes. Similarly, the provider-to-customer relationships must be well-founded.

Network Assumptions

This proof assumes *fairness*: update messages sent by a BGP router are eventually delivered. Formally, this proof assumes that for every link state in a trace, every non-empty connection is associated with a natural number n such that after n transitions in the trace, the trace contains a FWD transition that delivers the update message on that link.

This proof further assumes that there exists exactly one AS that knows how to directly route a certain prefix. Formally, for every prefix p, there exists exactly one AS $x_0(p)$ with exactly one router $r_0(p)$ that injects the route for that prefix with attributes $a_0(p)$. This is usually the case on the Internet, but ASes may violate this assumption due to misconfiguration [10].

This proof further assumes that after an initial phase where injections can happen, no new injections will happen, and no existing injections will be withdrawn (i.e. the trace will contain no INJ transitions). This is an unrealistic assumption. The Internet as a whole frequently injects new routes and withdraws already injected routes, such that the Internet is never in a truly stable state. However, because the BGP protocol handles prefixes independently, there is usually enough time for any particular prefix to become stable, even if some other prefixes are still converging.

Finally, the proof assumes that none of the routers will ever update their uninterpreted state (i.e. the trace will contain no UPD transitions). This is again an unrealistic assumption, but works in practice because only few routers have to choose between routes based on a router's uninterpreted state.

Route Selection

Both GR's original proof and our extended proof of BGP convergence assume that BGP routers select update messages according to a preference relation.

The BGP specification requires a BGP router r with uninterpreted state u to select attributes a from neighbor i over attributes a' from neighbor i', iff (in order of priority) (RFC 4271, §9.1.2.2):

1. a has a higher local preference value (*pref*) than a'
2. a has a shorter AS path (*aspath*) than a'
3. a has a better origin value than a'
4. a has a lower MED value than a'
5. i is non-internal, and i' is internal
6. r assigns a lower interior cost to sending traffic to i than to i' (e.g. computed by the OSPF protocol and stored in u)
7. i has a lower router identifier value than i'
8. i has a lower router address than i'

Attributes with the *na* value are never selected if any other attributes are available. Router implementations and extensions of BGP often implement a customized version of this selection algorithm, e.g. they may insert an additional check or change the order of the checks.

If routers are configured to ignore MED or compare MED across all advertisements (which both GR's original proof and our extended proof assume) then the above selection algorithm describes a preference relation [25]. Our proof is parametric over the exact implementation, and just requires that a router select attributes according to some preference relation. The advantages of a parameterized preference relation are described in Section 3.2.3.

Below, we first describe the preference relation $(i, a) \succeq_u (i', a')$ that this proof is parametric over. It compares attributes a and a' received from the neighbors i and i' by a router with uninterpreted state u. Second, we describe the function *dec* which selects attributes according

to this preference relation. Third, we describe a preference relation $a \geq a'$ that compares attributes a and a', without the need to know the neighbors from which they were received or the uninterpreted state.

Preference Relation for Incoming Attributes This proof is parametric over the exact implementation of the preference relation $(i, a) \succeq_u (i', a')$, it just requires that for any router r with uninterpreted state u, the $\preceq_u$ relation is a total preorder, i.e. it is reflexive ($(i, a) \preceq_u (i, a)$), transitive ($(i, a) \preceq_u (i', a') \rightarrow (i', a') \preceq_u (i'', a'') \rightarrow (i, a) \preceq_u (i'', a'')$), and total ($(i, a) \preceq_u (i', a') \vee (i', a') \preceq_u (i, a)$). The preference relation demanded by the BGP specification is in fact a total preorder.

Further, our semantics requires that for any router r with uninterpreted state u the preference relation is partially antisymmetric. If attributes a from neighbor i and attributes a' from neighbor i' are equally preferred, then their neighbors must be the same, i.e. $(i, a) \approx_u (i', a') \rightarrow i = i'$. In other words, attributes received from different neighbors are never tied on preference (they are totally ordered), i.e. $i \neq i' \rightarrow ((i, a) \prec_s (i', a') \vee (i', a') \prec_s (i, a))$. We say that attributes a from neighbor i and attributes a' from neighbor i' are equally preferred $(i, a) \approx_u (i', a')$ iff $(i, a) \preceq_u (i', a') \wedge (i', a') \preceq_u (i, a)$. Because each neighbor has a unique router address, the preference relation demanded by the BGP specification is in fact partially antisymmetric. However, it is not antisymmetric. For example, consider two announcements a and a' that differ in their communities, and were received from the same neighbor i. They will have the same preference, but will not be equal. Thus, $(i, a) \approx_u (i, a') \not\rightarrow (i, a) = (i, a')$.

Incoming Neighbor Selection Our semantics of BGP selects the route to forward packets using the $dec(r, u, I)$ function. This function chooses the neighbor with the most preferable attributes, given a mapping I from every neighbor of router r to attributes, and the router's uninterpreted state u i.e. $\forall i, I(dec(r, u, I)) \succeq_u I(i)$. The selected neighbor is unique, because attributes from different neighbors cannot have the same preference (partial antisymmetry). The selected neighbor exists, because there is at least one neighbor (*injected*), and because $\preceq_u$ is total.

While the *dec* function cannot be configured directly, it can be configured indirectly by configuring the *imp* function to appropriately change an update message's *pref* field. For example, a router can be configured to select the cheapest available route to forward packets (ASes usually receive or pay money whenever they send a packet to another AS), by configuring the *imp* function to set a higher *pref* value for cheaper routes.

Preference Relation for Attributes The preference relation $\preceq_u$ defines how to compare attributes at the same router, but it is sometimes necessary to also compare attributes at different routers with different uninterpreted states. Fortunately, the checks 1-4 performed by $(i, a) \preceq_u (i, a')$, up to (but not including) the non-internal vs internal check, only depend on the attributes a and a', not the routers i, i', and the uninterpreted state u. We can thus say that attributes a are preferred over attributes a' ($a \geq a'$), if a is preferred over a' up to

the non-internal vs internal check. The *na* value has strictly lower preference than any other attributes. This relation is a total preorder, but it is not antisymmetric and thus not an order (i.e. two different attributes may be equally preferred). We say that attributes a and attributes a' are equally preferred ($a \simeq a'$) iff $a \leq a' \wedge a' \leq a$.

Because attribute preference is derived from the preference demanded by the BGP specification, if an announcement a from neighbor i is preferred over announcement a' from neighbor i', then a is preferred over a', i.e. $(i, a) \succeq_u (i', a') \rightarrow a \geq a'$ (the converse is not necessarily true).

Further, because this preference relation performs all checks up to the non-internal vs internal check, if an announcement a_e is preferred over an announcement a_i, and given a non-internal neighbor i_e and an internal neighbor i_i at some router r with uninterpreted state u, then a_e received from i_e is preferred over a_i received from i_i, i.e. $a_e \geq a_i \rightarrow (i_e, a_e) \succeq_u (i_i, a_i)$.

3.3.3 Convergence Proof

The goal of this section is to show that any BGP network will converge, assuming it follows the assumptions and guidelines from the previous Section 3.3.2.

A BGP network *converges*, iff for every execution trace of the BGP protocol from some valid initial state, the BGP network will eventually become stable.

Some proposition P happens *eventually*, iff there exists a natural number n, such that P holds for every network state of the trace after the first n transitions. This terminology is inspired by modal logic.

A BGP network will eventually become *stable*, iff every AS of the BGP network will eventually become stable for every prefix. An AS will eventually become stable for a prefix p, iff every router of the AS will eventually become stable for prefix p. A router r will eventually become *stable* for prefix p, iff it will eventually select some attributes for prefix p that will never change. Formally, there exist best attributes a_r and a best incoming neighbor i_r, such that: 1) r's Adj-RIBs-In for p and i_r^* will eventually contain a_r, and 2) the imported a_r will eventually be preferred over any other attributes a for p imported from any other incoming neighbor i of r. r's Loc-RIB will thus eventually contain the imported attributes a_r, and all of the router's outgoing connections will eventually become stable for p.

The connection from one router s to another router d will eventually become stable for prefix p, iff the list of messages for prefix p on the connection will eventually be empty. Once stable, d's Adj-RIBs-In for s will contain the attributes from s's Adj-RIBs-Out for d.

An initial state is *valid*, iff all RIBs map to *na*, and there are no update messages on any connection; except that for any prefix p the injecting router $r_0(p)$ has injected its announcement $a_0(p)$.

Going forward, we prove all statements of stability for some given BGP network that complies with Section 3.3.2, some given execution trace of that network, and some given prefix p. Our final theorem will combine these proofs into a proof that holds for *all* compliant BGP networks, traces, and prefixes.

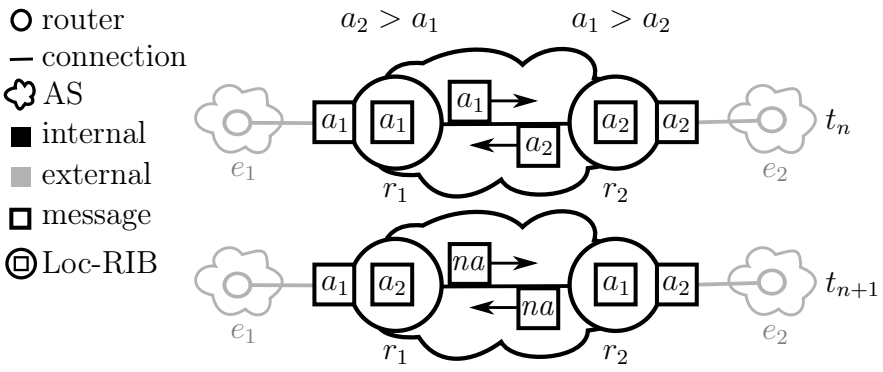

Figure 3.7: Divergence Example. The AS with router r_1 and r_2 can transition between the network state at t_n and t_{n+1} forever, and is thus not guaranteed to eventually become stable.

The proof that a BGP network will converge proceeds in two steps. Section 3.3.3 proves that a single AS will eventually become stable, assuming that some of the incoming connections to the AS will eventually become stable. Section 3.3.3 uses the local convergence proof to show that a BGP network converges globally. This proof follows Gao & Rexford's original proof [25].

Internal Stability

The goal of this section is to show that an AS will eventually become stable, assuming that some of the incoming connections to the AS will eventually become stable. Note that GR's simplified semantics of BGP does not accurately model interactions between routers within a single AS, and they did not prove any results regarding internal stability.

To motivate our guideline extensions, consider the following proof that even without external update messages, an AS may fail to converge internally.

Example (Internal Divergence)**.** *Without our import guideline, there exists an AS that is not guaranteed to eventually become stable, even assuming that* all *incoming connections to the AS will eventually become stable.*

Proof. Consider the example from Fig. 3.7. The AS x consists of two routers r_1 and r_2. To show that x is not guaranteed to eventually become stable, we construct a trace in which x's routers keep changing their Loc-RIBs forever.

Let the network topology surrounding e_1 and e_2, be such that we can construct a partial trace where e_1 and e_2 will eventually be stable, and that this trace leads to a network state where r_1 has received a message with attributes a_1 from e_1, and r_2 has received a message with attributes a_2 from e_2. The AS's incoming connections will thus eventually become stable.

Now consider the case where router r_1 prefers the attributes a_2 over a_1, and router r_2 prefers the attributes a_1 over a_2. This can for example be implemented by increasing the

pref value in the *imp* function for messages received from internal routers (which violates our internal import guideline).

Now consider the following extension of the partial trace. Each router will select the attributes received from the external neighbor, and will export those attributes to the other internal router. This is the situation at time t_n in Fig. 3.7.

Next, consider extending the trace such that the routers receive the attributes just sent over the internal connection. Router r_1 receives a_2. r_1 prefers a_2 over a_1 and selects it. Further, r_1 exports a_2 to the internal neighbor r_2. *exp* drops a_2 as demanded by rule 3 in Fig. 3.5. *na* is not equal to a_1, which was previously stored in the Adj-RIBs-Out, and r_1 thus withdraws the previously sent attributes a_1 from router r_2 by sending *na*. The same happens at r_2 except with different attributes. This is the situation at time t_{n+1}.

Next, consider extending the trace such that the routers receive the withdraws. Now that the internal message has been withdrawn, they select the external message and do the appropriate export. This is the same situation as the one at time t_n. We can thus continue extending the trace indefinitely, in such a way that the AS will never become stable. □

GR overlooked the problem of internal stability, because the original proof by GR uses a simplified semantics of BGP where every router has instant access to all messages received by all other routers of the same AS, and an AS thus instantaneously becomes stable, once all incoming connections have become stable.

The goal henceforth is to show that an AS x implementing the guideline extension will eventually become stable, assuming that there exists a non-empty subset D of x's routers such that: 1) every router r in D will eventually become non-internally stable, and 2) D is dominant (non-internally stable and dominant are defined below).

A router will eventually become *non-internally stable* iff it will eventually be stable for *non-internal* (i.e. either injected or external) incoming neighbors. Formally, there exist *best non-internal attributes* a_r and a *best non-internal incoming neighbor* i_r, such that: 1) r's Adj-RIBs-In for i_r^* will eventually contain a_r, and 2) the imported a_r will eventually be preferred over any other attributes a imported from any other non-internal incoming neighbor i of r. Once non-internally stable, the router's Loc-RIB might still change.

A set of non-internally stable routers D is *dominant*, iff the imported best non-internal attributes of all routers in D are equally preferred, and the imported best non-internal attributes are strictly more preferable than any imported non-internal attributes of any other router in the AS. Note that dominance, just like all other properties, are defined for one particular prefix. This is illustrated in Fig. 3.8. Formally, for any two routers r and r' of the AS x where r is in the set D, the following must hold: 1) If r' is in D, then r's imported best non-internal attributes a_r^* must have the same preference as r''s imported best non-internal attributes $a_{r'}^*$, i.e. $a_r^* \simeq a_{r'}^*$. 2) If r' is not in D, then r's imported best non-internal attributes a_r^* must always be strictly preferred over any imported non-internal attributes a' of r', i.e. $a_r^* > a'$.

The proof that the AS will eventually become stable proceeds in three steps. The *Internal Link Invariant* shows that the attributes of any internally sent messages have equal or worse

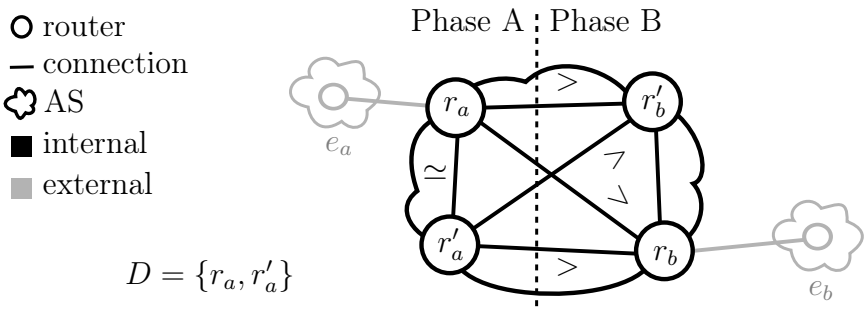

Figure 3.8: Internal Convergence Example. The routers r_a and r'_a in D dominate (their routes are better than) the AS's other routers. The AS will therefore eventually become stable.

preference than the best non-internal attributes. *Claim A* shows that routers in D will eventually become stable. *Claim B* shows that all other routers of x will also eventually become stable.

Lemma (Internal Link Invariant)**.** *Eventually, the attributes of any update message on any internal link will have equal or worse preference than the best non-internal attributes.*

Proof. Any update messages that are already on internal connections will eventually be delivered due to fairness. It thus suffices to only show that new update messages sent via internal connections have equal or worse preference than the best non-internal attributes. We have to consider two cases: 1) A router (e.g. router r_a in Fig. 3.8) exports an update message received from an internal neighbor (e.g. router r'_a). By rule 3 of Fig. 3.5, update messages from internal routers are blocked on export to other internal routers. Thus, the router may only send a withdraw (na), and na's preference is less than or equal to any attributes (see Section 3.3.2). 2) A router exports an update message received from a non-internal neighbor (e.g. router e_a). Because of dominance, the attributes of any update message from a non-internal neighbor are less than or equal to the best non-internal attributes. The same will be the case for the exported message, due to the extension to GR's guidelines (see Section 3.3.1). □

Lemma (Claim A)**.** *Any router r in D (e.g. router r_a) will eventually become stable.*

Proof. By the definition of dominance, we already know that r's non-internal incoming attributes will eventually become stable. What remains to be shown is that eventually, the router will never receive an update message with attributes a from an internal connection i, such that (i, a) is preferable over the router's best non-internal attributes (i^*_r, a^*_r).

We know that the preference of a is equal or lower to the preference of a^*_r by the Internal Link Invariant. Even if a's preference is equal to a^*_r's preference, a^*_r will be preferred because update messages from non-internal neighbors are preferred over update messages from internal neighbors (see Section 3.3.2). □

Lemma (Claim B)**.** *Any router r not in D (e.g. router r_b) will eventually become stable.*

Proof. Every router r' in D (e.g. router r_a) will eventually become stable by Claim A. In the process, r' will export its best attributes to each of its neighbors, including router r. The attributes of the update message that is eventually received by r have equal preference to r''s best attributes by the extension to GR's guidelines.

Eventually, r will have received the best attributes from each dominant router. Router r is guaranteed to prefer one of these over all other attributes, by the definition of dominance, and the preference relation's partial antisymmetry (see Section 3.3.2). Once this route is selected, router r will be stable. □

Theorem (Internal Stability)**.** *An AS will eventually become stable, assuming that a non-empty subset D of the AS's routers will eventually become non-internally stable, and the set of routers D is dominant.*

Proof. By Claim A and Claim B. □

Global Convergence

The proof of global convergence proceeds in three steps. The *Gliding Path Invariant* shows that any update message will first steadily increase in its AS level, and then steadily decrease in its AS level. In contrast to Sobrinho et al. [75], GR assumed this invariant, but neither stated nor proved it explicitly. *Claim 1* shows that *Phase 1 ASes* ((in)direct providers of the injecting AS x_0) will eventually become stable. *Claim 2* shows that *Phase 2 ASes* (all other ASes) will also eventually become stable. Note that the phases are defined for one particular prefix.

Lemma (Gliding Path Invariant)**.** *Any in-flight update message on a connection between AS x and x' is in one of three stages (as illustrated by Fig. 3.9):* Rising*: the message's AS level has so far strictly increased with every forwarding. This means that x is an (in)direct provider of x_0, and x' is a provider of x.* Gliding*: after the message's AS level strictly increased with every forwarding, it is now sent to a peer. This means that x is an (in)direct provider of x_0, and x' is a peer of x.* Falling*: the message's AS level has previously increased, potentially glided, and is now decreasing. This means that x is a provider of x'.*

Proof. This invariant is a consequence of GR's export guideline. A rising message (i.e. a message from a customer) can transition to the rising, gliding, or falling stage (i.e. be forwarded to a provider, peer, or customer); a gliding message (i.e. a message received from a peer) can only transition to the falling stage (i.e. be forwarded to a customer); and a falling message (i.e. a message received from a provider) can only transition to the falling stage (i.e. be forwarded to a customer). □

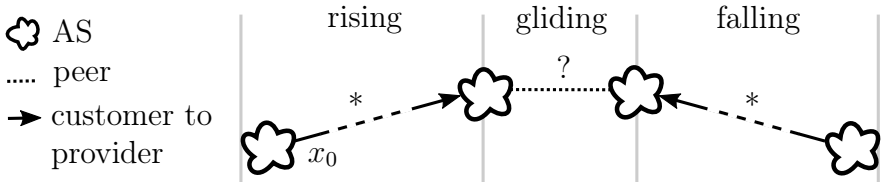

Figure 3.9: Gliding Path Invariant. Assuming the GR guidelines are followed, any update message first traverses any number of customer-to-provider connections, zero or one peer connections, and finally any number of provider-to-customer connections.

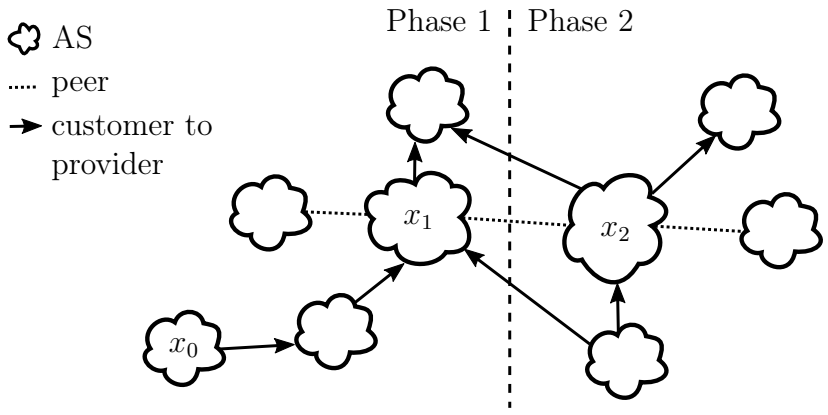

Figure 3.10: Global Stability Example. Illustrates a BGP network where AS x_1 is connected via all possible relationships (customer-to-provider, provider-to-customer, and peer) to various neighboring ASes from all Phases (Phase 1 and Phase 2). The same is true of x_2. Customer-to-provider relationships from Phase 1 to Phase 2 are impossible.

Lemma (Claim 1)**.** *Every Phase 1 AS (an (in)direct provider of the injecting AS x_0) will eventually become stable.*

Proof. Using well founded induction on the customer-to-provider relationship, we have to show that any Phase 1 AS x will eventually become stable, assuming that every Phase 1 customer AS of x will eventually become stable. We have to consider two cases:

If x is x_0, then r_0 is non-internally stable, and the singleton set containing only the router r_0 is dominant, because a_0 is better than any attributes any router could receive from any non-internal source, by the Injection Guideline. x thus eventually becomes stable by the Internal Stability theorem.

If x is not x_0 (e.g. x is the AS x_1 in Fig. 3.10), then we first show that the connection from every customer to x will eventually become stable. We have to consider two cases: 1) If the customer is a Phase 1 AS, the connection will eventually be stable by the induction hypothesis. 2) If the customer is a Phase 2 AS, the connection will always be empty (and thus stable) by the Gliding Path Invariant — Phase 2 ASes are not (in)direct providers of x_0, any message sent by a Phase 2 AS must thus be in the falling stage, and any connections to non-customers are thus empty. Note that Phase 2 ASes never send any messages to Phase 1 ASes.

Knowing that the connection from every customer of x will eventually become stable, pick the set of customer connections C that have sent the best attributes. These attributes are preferred over attributes from peers or providers, because of the import guidelines. Each router connected to any customers in C is thus non-internally stable, and the set of routers connected to any customers in C is dominant. x thus eventually becomes stable by the Internal Stability theorem. □

Lemma (Claim 2)**.** *Every Phase 2 AS (not a Phase 1 AS) will eventually become stable.*

Proof. Using well founded induction on the provider-to-customer relationship, we have to show that any Phase 2 AS x (e.g. x_2 from Fig. 3.10) will eventually become stable, assuming that every Phase 2 provider AS of x will eventually become stable.

It suffices to show that every router of x eventually becomes non-internally stable. If this is the case, the set of routers with the most preferred best non-internal attributes are dominant, and x thus eventually becomes stable by the Internal Stability theorem.

A router will eventually become non-internally stable, if every external incoming connection will eventually become stable.

Each external incoming connection is either from Phase 1 or Phase 2. Every connection from Phase 1 will eventually become stable by Claim 1. What remains are connections from Phase 2. There are two cases. 1) Connections from Phase 2 providers will eventually become stable by the induction hypothesis. 2) Connections from Phase 2 customers and peers will always be empty (and thus stable) by the Gliding Path Invariant — Phase 2 ASes are not (in)direct providers of x_0, any message sent by a Phase 2 AS must thus be in the falling stage, and any connections to non-customers are thus empty. □

Theorem (Global Convergence)**.** *All guideline-conforming BGP networks converge.*

Proof. All ASes will eventually become stable for any given trace and prefix p by Claim 1 and Claim 2. As the set of prefixes is finite, this implies that the BGP network will eventually become stable for any trace. This implies that for all traces, the BGP network will eventually become stable, and thus that the BGP network converges. □

We have formally stated and verified the above theorem, along with all its assumptions and lemmas, in Coq.

3.3.4 Comparison to Gao & Rexford's Original Proof

The pen-and-paper proof by GR makes various simplifying assumptions about the BGP protocol. For example, routers have access to all the routes received by other routers within the same AS, routes are not sent over a network but are instantly accessible whenever a router is "activated", and route announcements cannot be withdrawn.

Our proof uses our semantics of RFC 4271, which eliminates the aforementioned simplifying assumptions. Because our proof is expressed formally in the Coq proof assistant, it requires more rigorous and detailed reasoning. Furthermore, because the proof involves a more accurate model that introduces new complications, the proof requires additional insights.

Because our semantics models both intra-domain and inter-domain routing, we have to prove local stability for each AS (Section 3.3.3), which also requires an extension to GR's original configuration guidelines (Section 3.3.1).

Our formal proof extends GR's original proof with novel arguments about edge-cases (e.g. about the injection, availability, and withdraw of update messages).

Our formal proof also states novel invariants about BGP. For example, if a property P eventually hold for the update message in a router r's Adj-RIBs-Out to some neighbor r', then P will eventually hold for every message on the connection between r and r', and P will eventually hold for the update message in r''s Adj-RIBs-In for r.

Our proof totals 5123 lines of Coq code. 432 lines formalize the proof's assumptions (Section 3.3.2), 218 lines prove global convergence (Section 3.3.3), 1410 lines prove internal stability (Section 3.3.3), and 3063 lines prove invariants about BGP (including the Gliding Path Invariant).

We have ensured that our guidelines are not vacuous by proving that some small AS topologies and configurations satisfy them. We have not yet proven that our guidelines are satisfied by realistic configurations from ISPs like Internet2 and BelWü. Note though, that we did instantiate our BGP semantics with these configurations in our evaluation of Bagpipe.

3.3.5 Discussion

Our main theorem is, "there exists some finite number of steps after which no more messages are sent and the control plane is quiescent." We structured our proof in a classical, and monolithic way: the proof simply shows that there exists a number of steps that is correct. If

we were to redo our proof, we would provide a constructive proof that decompose the above theorem into two clearly separated parts. One part that computes the number, and another part which proves that the number is correct. Such a structure would enable the statement and proof of additional theorems, like an upper bound on the number of steps it takes for a network to converge.

3.4 Bagpipe

The Bagpipe tool statically checks for router misconfigurations. Bagpipe defines a declarative domain-specific language to express control-plane specifications. The language is rich enough to express specifications inferred from real AS configurations, specifications found in the literature (such as the GR guidelines [25] and prefix-based filtering [56]), and specifications for 10 configuration scenarios from the Juniper TechLibrary [40, 9]. Bagpipe automatically verifies that an AS's router configurations satisfy the given specification.

Bagpipe was initially developed without any formal semantics of BGP. As a result, Bagpipe was not sound. It produced warnings that were useful in practice, but there was no guarantee that a bug in Bagpipe did not lead to false alarms or missed alarms. We formally verified the Bagpipe checker, so that it provides rigorous guarantees about the properties that it checks. Formally verifying Bagpipe using our semantics had two primary benefits. 1) It deepened our understanding of the domain, e.g. it led to the discovery of the *initial network reduction*, which justifies Bagpipe's verification algorithm. 2) It revealed 2 bugs in Bagpipe and ensured that the fixed version of Bagpipe correctly handles all the details in our semantics of RFC 4271. As a third benefit, the verification demonstrates the usefulness and richness of our BGP semantics.

3.4.1 Initial Network Reduction

Given a specification and router configurations for a single AS, Bagpipe either guarantees that the specification holds, or provides a counterexample to the specification's assertion. For example, a specification may assert that "*the AS's routers will never select routes for invalid IP prefixes*". If Bagpipe states that this specification holds, it provides the guarantee that:

$\forall\, t\ r\ p\ n,\ \texttt{let}\ (\Gamma, \Sigma) := t[n]\ \texttt{in}\ \mathit{invalid}(p) \rightarrow \texttt{locRIB}(\Sigma(r), p) = na$

This means that for all traces t, routers r, and invalid prefixes p, the router state Σ after n transition of the trace $t[n]$, does not have attributes selected for prefix p (Bagpipe currently assumes that traces are free of UPD and INJ transitions for routers of the AS under consideration).

Bagpipe is a search-based tool. Because the set of all traces is infinite, Bagpipe cannot verify such a specification by searching the set of all traces for a counterexample. Instead, Bagpipe only searches for counterexamples in a finite set of short trace prefixes from the initial state. This is sound because if any counterexample exists, one exists in the finite set.

Before the verification process, the Bagpipe implementers had only a vague intuitive understanding of why. During the verification process, we developed the initial network reduction, which justifies searching only this finite space.

The initial network reduction formalizes the observation that if a BGP router will ever select or forward a particular update message, it would also do so immediately after initialization, i.e. before it has received any other update messages. This is because, in the initial network, the message does not have to compete with any other messages during the selection phase. A BGP network thus exhibits "maximal behavior" with respect to a given announcement at the beginning of its execution trace. We formally verified the initial network reduction in Coq.

3.4.2 Bagpipe Correctly Handles RFC 4271

Our verification of Bagpipe identified two bugs. That version of Bagpipe did not verify policy specifications for *na* attributes in the adjRIBsIn. It also created incorrect paths in which a router r appeared twice if an update messaged entered and exited the AS at r without being forwarded within the AS. These bugs did not seem to lead to user-visible failures, such as missed alarms or false alarms when running Bagpipe over real BGP configurations. The authors of Bagpipe acknowledged and fixed these bugs. We then formally verified Bagpipe's soundness. The proof is 2960 lines of Coq code.

Our verified version of Bagpipe uses naive brute force search to check the correctness of policy specifications, instead of the fast solver-aided checking performed by Bagpipe. Our verified version of Bagpipe is thus not yet ready to replace the original version of Bagpipe at run-time.

3.5 C-BGP

The easiest and most cost-effective way to find bugs is testing, and real systems tend to be tested before verification is attempted. Because our BGP semantics is executable, it aids in testing other tools and it can be tested itself.

We performed differential testing [20] of our semantics against C-BGP [64], a popular open-source BGP simulator. In differential testing, test cases are run in multiple implementations of a system and their output is compared; a difference indicates a possible bug in one of the implementations. Passing tests provide evidence for the correctness and completeness of both implementations. Our testing revealed two bugs in C-BGP and no bugs in our BGP semantics.

C-BGP enables network engineers to test configurations without running them on real hardware. It simulates a group of routers from any number of ASes running the BGP protocol. Users configure C-BGP by describing a network's topology and by inputting router configurations in a format similar to the one accepted by Cisco routers. C-BGP is used by network administrators to determine the effects of changes to their configuration or topology; it is therefore important that it faithfully reflects the BGP specification.

We wrote a testing tool in Coq that determines whether a sequences of BGP events (outputted by C-BGP) are permitted by our semantics. We used Coq's extraction facility to extract it, and our BGP semantics, to Haskell. We wrote a parser for C-BGP's trace format in order to convert C-BGP's text traces to checkable sequences of events. We then used the QuickCheck random testing tool [14] to write generators for random network topologies and BGP configurations, and we converted them to both C-BGP's input format and to the corresponding functions in our BGP semantics (e.g., *imp*). This enables us to check C-BGP traces on randomly-generated inputs to determine whether they agree with our semantics.

We ran this differential testing tool over 100,000 times, on randomly generated topologies of up to 8 ASes, each of which has up to 8 border routers. Each AS is full-mesh; the connections between other routers are random. Each router's configuration is randomly generated, and consists of up to 8 rules per neighbor. Each rule can test the communities on a route (multiple tests are combined using standard boolean operators), can add or remove communities, and can accept or reject a route.

The randomly generated topologies and configurations cover a number of corner cases such as adding and then removing the same community, creating the same route via two different code paths, and sending messages back to the originating router. We have not formally measured the coverage of the test cases.

The process revealed no errors in our semantics. The cases where C-BGP disagreed with our semantics revealed two issues in C-BGP.

If a router receives two messages two messages m_{i1} and m_{i2}, and its export filters yield the same message m_o for both, the router will send m_o twice in a row to the same neighbor, in violation of Section 9.2 of the BGP specification RFC 4271 [65]. Sending redundant messages can have serious consequences, e.g. it may impact BGP convergence times by triggering route flap dampening. We reported this bug to the C-BGP maintainers, who acknowledged it as a bug.

Another issue with C-BGP is that rejected update messages are not placed in a router's `adjRIBsIn`. Thus, a router can receive a legitimate withdraw message when there is no corresponding update message in its `adjRIBsIn` — a situation the developers thought was impossible, according to a comment in the C-BGP source code. This bug does not lead to incorrect observable behavior (since it does not affect the trace of delivered messages), but it is still a violation of the BGP specification.

Testing is an easy, effective, standard software engineering practice, and we encourage other researchers to take advantages of synergies between tests and proofs as we have done. The ability to test is a motivation for an executable semantics. Whereas testing provides no guarantee, it can find bugs like those in C-BGP, and successful testing increases confidence in a system. For instance, it validates our not-available semantics, which corresponds to multiple different concepts in C-BGP, like in the BGP specification.

3.6 Related Work

In this section, we address related work on BGP formalisms, checkers, simulators, and software-defined networking.

BGP Formalisms The Stable Paths Problem (SPP) [29] is a simplified model of BGP for which many theoretical results have been proven, including that solving SPPs is PSPACE hard [12]. In contrast to our semantics, SPP does not model all required features of RFC 4271. For example, SPP does not model routers within an AS, multiple connections between ASes, Routing Information Bases, update messages and all their attributes, the full route selection algorithm, update message withdrawals, and multiple ASes injecting a route for the same prefix; all these features are frequently used in practice. Extensions exist to SPP [30, 28, 77] that mitigate some (but not all) of these limitations.

Gao & Rexford's proof of Internet-wide route convergence [25] is based on a simplified model of BGP which, in contrast to our semantics, does not model all required features of RFC 4271. The proof has also been adapted to SPP [24].

Andreas Voellmy used Isabelle/HOL to formalize a simplified model of BGP's operation at the AS level [89]. Similar to SPP, it does not model the behavior of individual routers or communication within an AS. This model was used to verify one policy for one textbook example configuration.

The Formally Verifiable Routing (FVR) project [91, 90, 92] provides a formal algebra for reasoning about BGP properties (e.g. convergence). FVR's formalism is based on the SPP semantics and thus has the same limitations.

BGP Checkers and Compilers Propane [7] provides a high-level language that BGP administrators can use to specify how packets should be routed though the network. Propane then compiles specifications written in this language to a collection of BGP router configurations. This compilation step is currently not verified. We believe that our semantics could be used to verify the compilation step of Propane, and similar tools like Nettle [88], to guarantee that their generated BGP configurations are correct by construction.

Batfish [22] is a Datalog-based BGP configuration checker, which translates router configurations and a topology description into Datalog facts. Given these facts, Batfish employs a set of Datalog rules to populate each router's routing tables. Batfish's model of the BGP data-plane is quite accurate, and can be used to test properties (e.g. network convergence) given a particular set of router configurations and received BGP announcements. However, unlike our semantics, it cannot be used to verify properties for all configurations and all announcements in the control plane.

rcc [21] is a BGP configuration checker that is notable for its adoption by AS administrators and for finding a large number of router misconfigurations. rcc infers inter-AS relationships from router configurations to find violations of route validity and path visibility. The tool is not based on a formal model of BGP, and reports both false positives and false negatives.

BGP networks are often symmetric, which allows BGP checkers to exploit techniques recently developed by Plotkin et al. [63]. Our semantics could be used to gain confidence in the correctness of such checkers.

BGP Simulators There exist many BGP simulators [64, 66, 59, 55] that, given a topology and a set of configurations, determine how traffic will be routed. Network administrators can use simulators both for debugging existing problems and for testing potential new configurations. Our semantics can be used to test simulators, as well as actual implementations of BGP routers, which have been bug-prone in the past [52].

SDN Software defined networking (SDN) is a new paradigm for intra-domain routing within an AS. With SDN, routing is controlled by a single program running on a master router, instead of the interplay between a multitude of individually configured routers. Much work has been devoted to verifying the behavior of SDNs, especially on the data plane, including language support [57], model-checking [3, 18], and full formal verification [31, 1]. By providing a semantics for BGP, we hope to enable similar achievements for inter-domain routing (routing between ASes) and the control plane.

3.7 Conclusion

This chapter presented the first mechanized formal semantics of the BGP specification RFC 4271. The semantics is implemented in Coq. Our semantics models all required features of the BGP specification modulo low-level details such as bit representation of update messages and TCP.

Three case studies showed how to use our semantics to develop reliable proofs, checkers, and simulators. 1) We formalized the seminal pen-and-paper proof by Gao & Rexford on the convergence of BGP. 2) We verified the soundness of the Bagpipe tool which automatically checks that BGP configurations adhere to given specifications, finding 2 bugs in a prototype implementation. 3) We tested the popular BGP simulator C-BGP against our semantics, revealing 2 bugs in C-BGP.

Our work's contributions can be viewed in several ways.

Our work contributes a verification case study that has produced valuable artifacts in the domain of BGP: a formal proof of Gao & Rexford's configuration guidelines correctness; identification of flaws in a configuration-checking tool and a proof of correctness for the corrected tool; a differential tester and bugs in the C-BGP simulator.

Our work contributes the first mechanized semantics for BGP — a solid formal foundation that will enable the creation of future tools and proofs with similar impact. The semantics refactors the messy, informal English specification into a simpler, cleaner form, yielding a pluggable architecture in which users can make their own decisions about what details to retain and which to abstract away.

Our work contributes new proof methodology and techniques. We built a library that eases modeling infinite executions as co-inductive sequences of states, which is mathematically natural but heretofore has been ill-supported by Coq. We abstracted out and parameterized over required properties; while the notion of abstraction is not novel, other researchers can mimic the successful way that we did it. An underappreciated benefit of an executable model is the ability to run tests; we confirmed the benefits of testing in addition to proof, and we encourage other researchers to do the same.

Chapter 4

SPACE SEARCH

4.1 Introduction

Solver-aided tools are used in a variety of domains including data-race detection [48, 67, 44], memory-model checking [83], and compiler optimization validation [50, 43]. Such tools reduce complex properties in their application domain to simpler queries that can be checked by a high performance automated solver. In practice, the correctness of these reductions is rarely formally verified, decreasing confidence in the soundness of the tool.

Past work has helped mitigate this problem by providing solver-aided host languages, such as Smten [86] and Rosette [81, 82]. These languages modify an existing language runtime to provide a higher-level interface to the underlying solver, which reduces the effort required to build solver-aided tools by orders of magnitude and, because the implementation is simpler, improves confidence in the tool's correctness.

However, solver-aided host languages are not designed to support formal reasoning about the meaning of solver calls in terms of the tool's application domain. Such reasoning is often necessary, since reductions to satisfiability typically depend on sophisticated domain knowledge, making them difficult to get right [96, 83, 43, 71]. For example, PEC is a solver-aided tool for verifying compiler loop optimizations [43]; it decomposes the proof of equivalence between the original and optimized code (its *application domain property*) by splitting the code at its branching points, and using an SMT solver to establish the equivalence of the resulting straight-line code fragments. Ensuring that the equivalence of these straight-line code fragments can be "stitched together" to prove the optimization correct requires reasoning in a higher-order logic [79].

Even after a problem has been reduced to a solver query, various optimizations are typically required to achieve good performance, including selecting the right solver data types [73, 72, 61], query incrementalization [62], and query parallelization [38]. Without the ability to formally reason about solver results, it is difficult to ensure that such optimizations maintain the soundness of the tool.

This chapter presents SpaceSearch, a library that provides a higher-level interface for building and formally verifying solver-aided tools within a proof assistant, i.e. SpaceSearch is a solver-aided host language for proof assistants. Using the expressive logic of the proof assistant, programmers can formally verify that the results of SpaceSearch's operations are sufficient to establish the desired application domain property. Once a solver-aided tool is implemented against this interface, it can be extracted (translated) to a solver-aided host language (e.g. Rosette) where the SpaceSearch interface is instantiated with calls to an SMT

solver. This combines the strong correctness guarantees of developing a tool in a proof assistant with the high performance of modern SMT solvers. To enable construction of efficient tools, SpaceSearch employs a modular design that factors its interface into multiple abstract data types (ADTs). Thanks to this design, SpaceSearch can be easily extended with new backends and optimizations, including incrementalization and parallelization.

We evaluate SpaceSearch on two solver-aided tools. First, we built and verified SaltShaker, a solver-aided tool that checks, for all possible machine states, that an x86 instruction executed by RockSalt's Coq x86 semantics [58] behaves according to its instruction specifications extracted from STOKE [69]. SaltShaker verified the RockSalt semantics of over 15,000 instruction instantiations in under 2h, found 7 bugs in RockSalt, and found 1 bug in STOKE. We reported these bugs, and they were subsequently fixed by the respective developers.

Second, we used SpaceSearch to reimplement and verify a prototype of Bagpipe, a solver-aided tool written in Rosette that checks Border Gateway Protocol (BGP) configurations. Bagpipe's main algorithm relies on a sophisticated reduction from its domain-specific BGP problem to a set of SMT queries. Our verified reimplementation, dubbed BGProof, runs on industrial configurations with over 240 KLOC, finds 19 inconsistencies, and provides the same performance as the unverified Bagpipe prototype. In the process of verifying BGProof, we found 2 bugs in the Bagpipe prototype.

This chapter presents:

- The SpaceSearch library, which exposes an interface for constructing solver-aided tools in proof assistants, as well as formal denotational semantics to reason about this interface (Section 4.3).

- Various backends for SpaceSearch's high-level interface that enable solving search problems using brute force, parallel, incremental, and SAT/SMT search (Section 4.4), as well as a discussion of when each variant is most appropriate.

- An evaluation of SpaceSearch via the construction of two solver aided tools: SaltShaker checks the correctness of x86 instruction semantics in RockSalt (Section 4.5), and BGProof checks the correctness of Border Gateway Protocol (BGP) configurations (Section 4.6).

SpaceSearch presents, to the best of our knowledge, the first general approach to formally verifying solver-aided tools. This required several technical contributions including:

- Formalizing the Smten interface to support reasoning about solver aided tools in proof assistants (Section 4.2.1). Our denotational semantics enables both ease of reasoning and extensibility as shown by our case studies and libraries (Sections 4.3, 4.5 and 4.6).

$$
\begin{array}{ll}
Space : Type \to Type & [\![empty]\!] = \emptyset \\
[\![\ \_\ ]\!] : Space(A) \to \mathcal{P}(A) & [\![single(x)]\!] = \{x\} \\
 & [\![union(s,t)]\!] = [\![s]\!] \cup [\![t]\!] \\
empty_A : Space(A) & \\
single_A : A \to Space(A) & [\![bind(s,f)]\!] = \bigcup_{a \in [\![s]\!]} [\![f(a)]\!] \\
union_A : Space(A) \to Space(A) \to Space(A) & \\
bind_{A,B} : Space(A) \to (A \to Space(B)) \to Space(B) & search_A : Space(A) \to option(A) \\
 & search(s) = None \implies [\![s]\!] = \emptyset \\
 & search(s) = Some(a) \implies a \in [\![s]\!]
\end{array}
$$

Figure 4.1: SpaceSearch Basic ADT.

- Extensions to the Smten interface to support parallelization and incrementalization (Sections 4.3 and 4.4). These extensions could be back-ported to existing solver-aided frameworks and are not proof-assistant specific. To the best of our knowledge, these general mechanisms are completely novel and were essential for building effective tools in our case studies.

- A translation from Smten's interface to Rosette's symbolic execution API (Section 4.4.2), thus enabling any Smten-based tool to potentially be ported to Rosette.

- An approach to exploiting proof assistant extraction mechanisms to support solver-aided tool development (Section 4.4.2) without modifying the host language interpreter (as done in Haskell for Smten, and Racket for Rosette).

- Designing a set of libraries users can employ to build and reason about solver-aided tools. The design tradeoffs for such libraries were not obvious to us, and we believe the discussions in Section 4.3 will be valuable for users applying and extending SpaceSearch.

4.2 Overview

SpaceSearch provides a high-level interface to solver operations and their semantics in a proof assistant, enabling development and verification of solver-aided tools. Instead of exposing a low-level solver interface (e.g., SMTLib [4] data types and commands), SpaceSearch provides a high-level interface inspired by Smten [86]. This interface exposes solver functionality as operations to construct and to automatically solve *search problems*, leading to both compact high-level encodings and high-performance solver-aided tools [86].

This section presents an overview of SpaceSearch operations for constructing and solving search problems, a denotational semantics to reason about these operations, and an explanation

of how they are implemented upon extraction. We show how to use SpaceSearch to build and verify a toy solver-aided tool for solving n-Queens problems.

4.2.1 SpaceSearch Interface

The SpaceSearch interface (Figure 4.1) provides the search problem type, operations to construct search problems, and an operation to solve these problems. The SpaceSearch interface can be implemented either naively within the proof assistant (e.g. via the use of a finite set library), or efficiently by extraction to a solver-aided host language (e.g. Rosette).

Search Space Type SpaceSearch uses the type $Space(A)$ to represent search problems, which we call *search spaces*, for solutions of some type A. SpaceSearch assigns meaning to a search problem s by providing the function $[\![s]\!]$, which denotes s to a subset of the inhabitants of A (the powerset $\mathcal{P}(A)$). For example, the search problem of finding a "prime number greater than 1000" can be thought of as the problem of finding a value in the subset of numbers containing only "prime numbers greater than 1000".

Constructing Search Spaces SpaceSearch provides four operations to construct search spaces, the semantics of these operations is as follows. The $empty_A$ operation constructs a search problem with no solutions, $single_A(x)$ with exactly one solution x, and $union_A(s, t)$ with solutions of type A that are either in s or t. The $bind_{A,B}(s, f)$ operation creates a search problem by first applying f to every solution of type A in s, and then combining the solutions of type B from the resulting search problems into one. Search space subtraction is a subtle operation, discussed in Section 4.3.1. These operations are subscripted by the type of solutions in the search problem, but we omit subscripts that can be easily inferred from the context.

Note that the operations for constructing search spaces do not require the corresponding SpaceSearch implementation to actually enumerate the entire space. Because ADTs hide their implementation details, SpaceSearch is free to choose which ever internal representation is most efficient for conducting searches over a particular ADT's space.

Solving Search Spaces SpaceSearch also provides the *search* operation, which takes a search space s, and either returns *None*, which means that the search space s is empty; or $Some(a)$, which means that a is a solution to s. In the case of multiple solutions to s, only one arbitrary solution is returned. To return another solution, remove a from s and rerun *search*.

4.2.2 n-Queens Example

To illustrate SpaceSearch, we apply it to build and verify a simple solver-aided tool for solving n-queens problems. This case study, as well as the more complex case studies in Sections 4.5 and 4.6, follow the following methodology: 1) We formally express the tool's specification, using a proof assistant. 2) We implement a solver-aided tool, using the operations

```
solveNQueens(n : Integer) : option(list(Integer × Integer))
  search(bind(placements(n, n), (λq.
    if noAttack(q) then single(q) else empty))).

noAttack(q : list(Integer × Integer)) : bool :=
  distinct(map(fst, q)) ∧ distinct(map(snd, q)) ∧
  distinct(map(plus, q)) ∧ distinct(map(minus, q))

placements(n, 0) := single([ ])
placements(n, S(x)) :=
  bind(range(0, n), (λy : Integer.
  bind(placements(n, x), (λq : list(Integer × Integer)
    single((x, y) :: q)))))
```

Figure 4.2: N-Queens in SpaceSearch.

of SpaceSearch. 3) We prove that the solver-aided tool meets its specification, using the denotational semantics of SpaceSearch.

1) n-Queens Specification A solution to an n-queens problem places n queens on an $n \times n$ chessboard so that no two distinct queens attack each other, defined as two queens sharing the same column, row, or diagonal. Formally, a set of integer pairs Q is a solution to the n-queens problem iff $|Q| = n$, all integers are in the range $[0..n)$, and:

$$\forall (x, y), (x', y') \in Q.(x, y) \neq (x', y') \implies x \neq x' \wedge y \neq y' \wedge |x - x'| \neq |y - y'|$$

The following section describes a solver-aided tool that decides the n-queens problem for any given n, i.e. for any n, it either returns a solution to the n-queens problem, or reports that no such solution exists.

2) n-Queens Solver-Aided Tool Figure 4.2 shows a SpaceSearch implementation of the n-queens decision procedure taken from a Smten tutorial [85]. The implementation consists of three functions:

solveNQueens takes the problem size n and uses the *search* operation to find a solution in the space of non-attacking queens. This space is constructed by binding over a space

of queen placements $placements(n, n)$, and only keeping those placements that are non-attacking.

$placements(n, m)$ is a space containing placements for m queens on an $n \times n$ chessboard. The space contains the placements that position the x^{th} queen (out of m) in the $x - 1^{\text{th}}$ column (x-value) and any row (y-value) contained in the space $range(0, n)$ of integers $[0..n)$.

$noAttack(q)$ checks whether a placement of queens q is non-attacking. This is implemented by checking that the column (x-values, accessed using *fst*) of all queens is distinct, that the row (y-values, accessed using *snd*) of all queens is distinct, that the column plus row ($x + y$) of all queens is distinct (*plus* sums the components of a pair), and that the column minus row ($x - y$) of all queens is distinct (*minus* subtracts the components of a pair).

3) n-Queens Correctness The Smten algorithm for deciding n-queens problems does not obviously meet its specification. This is mainly due to two optimizations, which we will now prove correct using the SpaceSearch semantics and Coq.

The first optimization reduces the space of all queen placements to the space containing only the placements $placements(n, n)$ that put each queen in a different column (x-value). We can prove this optimization correct in Coq, as any placement of two queens on the same column leads to an attack, and is thus not a solution.

The second optimization improves the performance of the *noAttack* check. Instead of checking that no two queens share the same diagonal (distance between the two queens' x-values equals distance between the two queens' y-values), it checks that the sums and differences of all queen placements are distinct. We prove this optimization correct by formalizing the intuitive argument given in the Smten tutorial [85].

The tutorial uses the following tables to explain why the optimization is correct for a 4×4 chessboard:

y \ x	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

sum $(x + y)$

y \ x	0	1	2	3
0	0	1	2	3
1	−1	0	1	2
2	−2	−1	0	1
3	−3	−2	−1	0

difference $(x - y)$

The first table labels the cell at position x, y with the sum $x + y$, while the second table labels the cell x, y with the difference $x - y$. Observe that any two queens with a different sum of x, y are also on a different diagonal going from bottom-left to top-right. Similarly, any two queens with a different difference of x, y are also on a different diagonal going from top-left to bottom-right. The optimized *noAttack* check therefore enforces the rules of the puzzle.

$$
\begin{array}{lcl}
\mathcal{P}(A) & := & A \to \mathit{Prop} \\
\emptyset & := & \lambda a.\, \bot \\
\{x\} & := & \lambda a.\, a = x \\
s \cup t & := & \lambda a.\, s(a) \vee t(a) \\
\bigcup_{a \in s} f(a) & := & \lambda b.\, \exists a.\, s(a) \wedge (f(a))(b)
\end{array}
$$

Figure 4.3: Ensembles.

While we omit both proofs for brevity , we note that SpaceSearch enables us to verify the solver-aided tool that will eventually be executed, not just a model of an n-queens algorithm.

4.3 The SpaceSearch Interface

The SpaceSearch interface exposes operations to construct and solve search problems in proof assistants. SpaceSearch bundles its operations by functionality into Abstract Data Types (ADTs) [49]. In general, ADTs provide (1) operations for introducing values of some abstract type and (2) operations for eliminating values of that type. This section presents the SpaceSearch ADTs for constructing and solving search problems.

4.3.1 Constructing Search Problems

Basic ADT SpaceSearch denotes (Figure 4.1) a search problem for solutions of type A to a subset of A. In the theorem prover, this subset is represented by an *ensemble*—a function that maps every value of type A to a proposition *Prop* (i.e. a logical claim). An ensemble s contains the value a if and only if the proposition $s(a)$ is true (i.e. if $s(a)$ is a provable logical claim). This is summarized in Fig. 4.3.

Using ensembles, the empty set $\emptyset$ is the function that maps every element a in A to the false proposition $\bot$, the singleton $\{x\}$ is the function that maps every element a to the proposition that is only true if a is equal to x, the binary union $s \cup t$ is the function that maps every element a to the proposition that is only true if a is either contained in s or in t, and the infinitary union $\bigcup_{a \in s} f(a)$ is the function that maps every element b to the proposition that is only true if there exists a value a in s, such that b is in the ensemble returned by $f(a)$.

Specialized Search Problems While the Basic ADT can be used to construct search problems for any of Coq's native types, these problems cannot always be solved efficiently. For example, we initially tried to build SaltShaker using Coq's native implementation of bit vectors. But we found that even simple space constructions, like the space of all 32-bit vectors equal to 5, cannot be searched efficiently (i.e., within a day). SpaceSearch therefore

$Integer : Type$
$[\![\_]\!] : Integer \to \mathbb{Z}$
$intPlus : Integer \to Integer \to Integer$
$[\![intPlus(n, m)]\!] = [\![n]\!] + [\![m]\!]$
$intFull : Space(Integer)$
$[\![intFull]\!] = \lambda n. \top$
...

$bv : \mathbb{N} \to Type$
$[\![\_]\!]_n : bv(n) \to \{m : \mathbb{N} \mid m < 2^n\}$
$bvZero_n : bv(n)$
$[\![bvZero_n]\!] = 0$
...

Figure 4.4: SpaceSearch Integer and BitVector ADTs.

also exposes ADTs to construct *specialized* search spaces that certain solvers can search more efficiently.

Figure 4.4 describes SpaceSearch's *BitVector ADT*, which provides the $bv(n)$ type for bit vectors of size n, as well as constants and operations on bit vectors (not shown). Elements of type $bv(n)$ are denoted to the natural numbers up to 2^n. Using the Rosette Backend, these bit vectors are extracted to the bit vector theory provided by the underlying SMT solver. The result is that the aforementioned space construction (of all 32-bit vectors equal to 5) can be searched in fractions of a second.

Infinite Search Problems The Basic ADT is sufficient to construct full spaces of finite types, e.g. the full space of the *bool* type is: $union(single(true), single(false))$. But these basic operations cannot be used to construct infinite spaces, like the space of all integers. SpaceSearch thus provides additional ADTs to construct *infinite* search spaces.

Figure 4.4 describes SpaceSearch's *Integer ADT*, which provides the *Integer* type. Elements n of type *Integer* are denoted to the mathematical integers $\mathbb{Z}$ with $[\![n]\!]$ (this function has the same syntax as, but is different from, the function provided by the Basic ADT). The ADT also provides constants and operations on integers, such as *intPlus*, which are denoted to the corresponding constants and operations on the mathematical integers.

The most interesting value provided by the Integer ADT is the *intFull* space. This space contains every one of the infinitely many integers, and is thus denoted as the ensemble that returns the true proposition $\top$ for every integer. As we will see in Section 4.3.2, providing this space has potential implications on the solvability of search problems.

Abstract Data Type (ADT) vs Domain Specific Language (DSL) During the design of the SpaceSearch library, we were confronted with the choice of either providing the interface in the form of a Domain Specific Language (DSL), or in the form of ADTs (which is shown in the previous sections).

$$
\begin{aligned}
&\mathit{Callable} : \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\
&\mathit{call}_{A,B} : \mathit{Callable}(A, B) \to A \to B \\
&\mathit{callableBind}_{A,B} : \mathit{Space}(A) \to \mathit{Callable}(A, \mathit{Space}(B)) \to \mathit{Space}(B) \\
&\mathit{callableBind}(s, r) = \mathit{bind}(s, \mathit{call}(r))
\end{aligned}
$$

Figure 4.5: SpaceSearch Callable ADT.

A library that provides its interface in the form of a DSL defines a language (usually as an Abstract Syntax Tree (AST)), and provides an interpreter for this language. For SpaceSearch, the DSL would be a language to construct search problems, whose AST $\mathit{SpaceAST}(A)$ would have the constructors:

$$
\begin{aligned}
&\mathit{emptyAST}_A : \mathit{SpaceAST}(A) \\
&\mathit{singleAST}_A : A \to \mathit{SpaceAST}(A) \\
&\mathit{unionAST}_A : \mathit{SpaceAST}(A) \to \mathit{SpaceAST}(A) \to \mathit{SpaceAST}(A) \\
&\ldots
\end{aligned}
$$

What differentiates the ADT from the DSL approach is that a user of the DSL interface knows exactly which operations can be used to construct search problems (because the user can pattern match on an AST), whereas a user of the ADT only knows that there are some operations to construct search problems (but there may be more).

To support extensibility with various optimizations like SMT theories, parallelization, and incrementalization, we opted to provide the SpaceSearch interface in the form of ADTs, where adding new operations is guaranteed not to break user code.

Callable Search Problems A weakness of the SpaceSearch interface is that it cannot be efficiently implemented directly in Coq. To see why, consider the expression $\mathit{bind}(s, f)$. No matter how the space s is implemented, an implementation of *search* in Coq will have to invoke f for every element in s (which is very slow or impossible), because in Coq, the only way to learn anything about a function is via function invocation.

SpaceSearch gets around this problem with the use of extraction. Once extracted, the SpaceSearch interface can be efficiently implemented because the target language's interpreter can inspect a function's source code, and can thus provide an efficient implementation of *bind*. For example, if the interpreter recognizes that the function f's source code is equivalent to *single*, it can just replace $\mathit{bind}(s, f)$ with s.

To also allow the efficient implementation of *bind* in Coq, SpaceSearch provides *callableBind* (Fig. 4.5), a version of *bind* whose second argument is a value that can be called, but depending

$$
\begin{aligned}
&\mathit{minus}_A : \mathit{Space}(A) \to \mathit{Space}(A) \to \mathit{Space}(A) \\
&[\![\mathit{minus}(s,t)]\!] = [\![s]\!] \setminus [\![t]\!] \\
&\mathit{incSearch}(s,t,f) := \mathit{search}(\mathit{bind}(\mathit{minus}(s,t),f))
\end{aligned}
$$

Figure 4.6: SpaceSearch Minus ADT and Incremental Search.

$$
\begin{array}{lcl}
\mathit{preciseSearch}_A : \mathit{Space}(A) \to \mathit{option}(A) & & \\
\mathit{preciseSearch}(s) = \mathit{None} & \Longrightarrow & [\![s]\!] = \emptyset \\
\mathit{preciseSearch}(s) = \mathit{Some}(a) & \Longrightarrow & a \in [\![s]\!]
\end{array}
\qquad
\begin{array}{lcl}
\mathit{heuristicSearch}_A : \mathit{Space}(A) \to \mathit{option}(\mathit{option}(A)) & & \\
\mathit{heuristicSearch}(s) = \mathit{None} & \Longrightarrow & \top \\
\mathit{heuristicSearch}(s) = \mathit{Some}(\mathit{None}) & \Longrightarrow & [\![s]\!] = \emptyset \\
\mathit{heuristicSearch}(s) = \mathit{Some}(\mathit{Some}(a)) & \Longrightarrow & a \in [\![s]\!]
\end{array}
$$

Figure 4.7: Search ADTs.

on the *Callable* type, may also support other operations, such as looking the value's abstract syntax tree. The *callableBind* operation takes as input a search space s and a callable r, calls r on every solution in s, and returns the search problem containing all the solutions produced by calling r. A callable r of type $\mathit{Callable}(A, B)$ can be called using the *call* operation, which converts r to a function from A to B. The $\mathit{callableBind}(s, r)$ operation thus has the semantics of $\mathit{bind}(s, \mathit{call}(r))$.

Space Minus and Incremental Search All of the operations in the aforementioned ADTs are monotonic: whenever the input spaces to these operations grow in size, the output size grows or stays the same. However, in some applications, it is useful to be able to reduce the size of a space. In particular, having a notion of subtracting spaces allows for performance gains by incrementalizing searches.

Figure 4.6 describes the Minus ADT and the incremental search function $\mathit{incSearch}(s, t, f)$. This function returns all the solutions of $\mathit{bind}(s, f)$ assuming that $\mathit{bind}(t, f)$ has already been searched and has no solutions. As a result, $\mathit{incSearch}(s, t, f)$ avoids having to perform the bind on a portion of the space that is already known not to return a solution.

The $\mathit{incSearch}(s, t, f)$ function is useful for applications that apply computationally expensive functions f to spaces that change often, but only in relatively few, easily isolated ways. For example, SaltShaker applies a computationally expensive verification function to every element in a frequently changing set of instructions.

4.3.2 Solving Search Problems

This section explains how to find solutions to search problems constructed using SpaceSearch ADTs. SpaceSearch's interface comes with ADTs to perform both precise and heuristic based search. These ADTs are formalized in Fig. 4.7.

Precise Search The *preciseSearch* operation takes as input a search space s and returns either *None*, which means that the search space s is empty, or $Some(a)$, which means that a is a solution to s. In the case of multiple solutions to s, the interface only specifies that one of them has to be returned, without specifying which one.

Unlike Smten, *preciseSearch* does not wrap search results in the IO monad. This enables the use of SpaceSearch in pure functional code, and simplifies reasoning in Coq. It is also safe because all SpaceSearch implementations (including Rosette) are pure (always returning the same solution for the same search problem), and support nested search queries. To also support impure and non-nesting implementations, SpaceSearch's interface could easily be extended with another ADT that wraps search results in the IO monad.

Heuristic Search The *heuristicSearch* operation takes as input a search space s and returns either $Some(None)$, which means that the search space s is empty; $Some(Some(a))$, which means that a is a solution to s; or *None*, which means that the *heuristicSearch* operation could not determine whether s contains a solution or not.

The *heuristicSearch* operation is important because some search problems are undecidable and thus lack a *preciseSearch* operation that always returns a result. For example, the *intFull* operation from the Integer ADT can be used to construct undecidable search problems (search problems constructed using only Basic ADT are, however, always decidable). One of the challenges of this library is to statically prevent the use of *preciseSearch* on undecidable search problems. Section 4.4 shows how the separation of the SpaceSearch interface into multiple ADTs achieves this goal.

The *heuristicSearch* operation is not only useful for undecidable problems, but it can also be used to perform QuickCheck [14] style testing. Such an implementation of $heuristicSearch_A(s)$ generates multiple elements of type A, and tests whether one of these elements is a solution to the search space s. If a is a solution, the operation returns $Some(Some(a))$. If none of the elements is a solution, it returns *None*.

4.4 The SpaceSearch Backends

SpaceSearch provides three backends that implement the various ADTs contained in the SpaceSearch interface. The Native Backend (Section 4.4.1) provides an inefficient, but provably correct, implementation of SpaceSearch directly in Coq (thus ensuring the consistency of the library interface); the Rosette Backend (Section 4.4.2) implements SpaceSearch interfaces using Rosette primitives on extraction to Racket; and the Places Backend (Section 4.4.3)

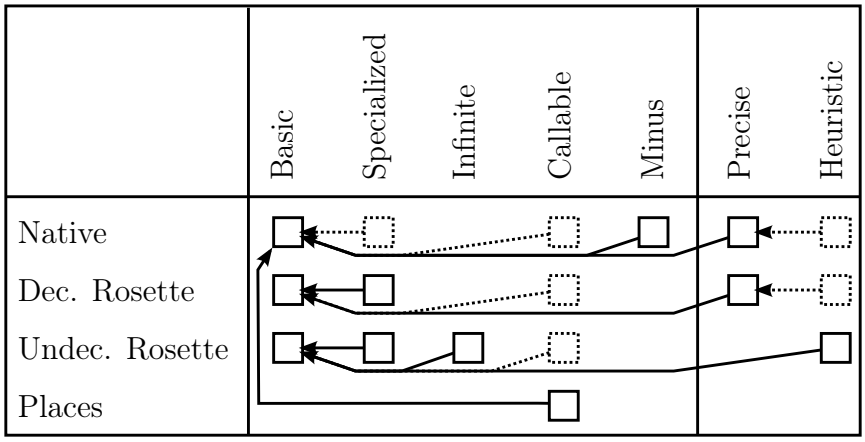

Figure 4.8: SpaceSearch Backend ADT Instances. A box means that the row's backend provides an instance of the column's ADT. An arrow $i \leftarrow j$ means that j depends on i. A dotted instance is automatically derived from the instance it depends on (indicated by a dotted arrow).

$$
\begin{array}{lcl}
\mathit{Space}(A) & := & \mathit{list}(A) \\
[\![s]\!](a) & := & \mathit{in}(a, s) \\
\mathit{empty} & := & [\,] \\
\mathit{single}(x) & := & [\,x\,] \\
\mathit{union}(s, t) & := & \mathit{append}(s, t) \\
\mathit{bind}(s, f) & := & \mathit{flatten}(\mathit{map}(f, s))
\end{array}
\qquad
\begin{array}{lcl}
\mathit{preciseSearch}([\,]) & := & \mathit{None} \\
\mathit{preciseSearch}(a :: \_) & := & \mathit{Some}(a) \\
\mathit{minus}(s, t) & := & \mathit{listMinus}(s, t) \\
\mathit{heuristicSearch}(s) & := & \mathit{Some}(\mathit{preciseSearch}(s)) \\
\mathit{Callable}(A, B) & := & A \to B \\
\mathit{call}(r) & := & r \\
\mathit{callableBind}(s, r) & := & \mathit{bind}(s, r)
\end{array}
$$

Figure 4.9: SpaceSearch Native Backend.

implements SpaceSearch's Callable ADT using Distributed Places [80] on extraction to Racket . Figure 4.8 provides an overview of the ADT instances provided by each backend, as well as the dependencies between these ADTs instances.

4.4.1 Native Backend

The Native Backend (Fig. 4.9) instantiates the Basic ADT's *Space* type with the type of lists, and denotes a list to an ensemble using the *in* predicate. The $\mathit{in}(a, s)$ predicate is true iff a is an element of the list s. *empty* is then just the empty list, $\mathit{single}(x)$ is the singleton list of x, $\mathit{union}(s, t)$ concatenates the two lists s and t, and $\mathit{bind}(s, f)$ first maps f over every element in s, and then flattens the resulting list. Infinite ADTs are not supported; for example, the

```
empty      ≜ (lambda () (assert false))
single(x)  ≜ (lambda () x)
union(s,t) ≜ (lambda ()
               (if (symbolic-bool) (s) (t)))
bind(s,f)  ≜ (lambda () ((f (s))))

preciseSearch(s)    ≜ (solve (s))
heuristicSearch(s)  ≜ (solve (s))
```

Figure 4.10: SpaceSearch Decidable and Undecidable Rosette Backend.

Native Backend does not provide an instance for the Integer ADT , because *intFull*—the list of all integers—does not exist.

The Native Backend instantiates the Precise ADT's *preciseSearch* operation by simply returning the first element in the list, if there is one. This implementation therefore depends on details of the Native Backend's Basic ADT implementation, namely that Spaces are lists. Space subtraction also depends on the fact that Spaces are lists in the Native Backend. The Native Backend's implementation of *minus* calls the $listMinus(l, l')$ function, which simply removes all elements in the list l' from the list l.

The Heuristic and Callable ADTs are implemented using existing operators from the Precise ADT and Basic ADT respectively. In the Heuristic ADT, *heuristicSearch* just performs a *preciseSearch*, and will thus never return an imprecise result. In the Callable ADT, $Callable(A, B)$ is implemented as an ordinary function $A \to B$, and *call* is thus just the identity function; using this definition of callable, *callableBind* is implemented as a direct invocation of *bind*. The Heuristic and Callable ADT instances do not depend on any implementation details of the Native Backend, and can thus be automatically derived from any Precise ADT and Basic ADT instance respectively.

We found the Native Backend useful for efficient search of small search problems, for testing a solver aided tool directly within Coq, and for verifying that SpaceSearch's interface is not vacuous, i.e. it can be implemented and proven correct.

4.4.2 Rosette Backend

The Rosette Backends provide an efficient implementation of SpaceSearch's ADTs using the Rosette language [81, 82], which extends Racket with primitives for constructing solver aided tools. A program using the Rosette Backends can be proven correct against the SpaceSearch interface in Coq, but can only be executed after extraction to Racket.

Rosette Background Rosette [81, 82] extends Racket with the following solver-aided primitives: *symbolic values*, which are created using functions such as (`symbolic-bool`) and (`symbolic-integer`); *assertions*, which are created using the `assert` construct; and *solver queries*, which are made via the `solve` construct. The `solve`(e) query takes an expression e and tries to find a concrete assignment to any symbolic values in e such that no assertion in

e is violated. If `solve` finds a valid concrete assignment c, it returns the expression e, where symbolic values have been replaced with concrete values from the assignment c.

For example, the following Rosette program checks De Morgan's law $\forall\, x\, y.\, x \wedge y \iff \neg(\neg x \vee \neg y)$ by checking that its negation is unsatisfiable:

```
(solve
  (let ((x (symbolic-bool)) (y (symbolic-bool)))
  (if (eq? (and x y) (not (or (not x) (not y))))
    (assert false) (cons x y))))
```

The `solve` query tries to find an assignment to x and y such that the if-condition evaluates to false to avoid the assertion failure. If `solve` found such an assignment (which it does not), it would return the tuple (`cons x y`) concretized with the values of the assignment (i.e. a counterexample). With this encoding, the verified property holds iff every execution of the program fails.

The `solve` query works by translating the input expression into an SMT-LIB [5] formula and solving it with an off-the-shelf SMT solver.

Extraction to Rosette SpaceSearch provides two Rosette Backends. The Decidable Rosette Backend implements those SpaceSearch ADTs that only allow the construction of decidable search problems (i.e. it does not implement the Integer ADT), and can therefore implement the Precise ADT. The Undecidable Rosette Backend also implements SpaceSearch ADTs that allow the construction of undecidable search problems, but can therefore only implement the Heuristic ADT.

The Rosette Backends implement SpaceSearch ADTs using *parameters*, which are extracted to the Racket terms described in Fig. 4.10. Coq's built-in extraction mechanism compiles Coq expressions to a target language, in our case Racket. Coq's extraction mechanism also supports the definition of extraction parameters—uninterpreted expressions that, at extraction time, are instantiated with an expression in the target language. In Fig. 4.10., $p \triangleq e$ indicates that the parameter p is extracted to the target language expression e.

To a first approximation, both Rosette Backends extract a search space to a symbolic value such that the valid instantiations of the symbolic value are equal to the solutions of the search space. In particular, $\mathit{single}(x)$ evaluates to the concrete value x, i.e. a symbolic value with exactly one instantiation; $\mathit{union}(s, t)$ evaluates to an symbolic value that, depending on the value of a symbolic boolean, is either the symbolic value s or the symbolic value t; *empty* evaluates to (`assert false`), i.e. a symbolic value with no instantiations; and $\mathit{bind}(s, f)$ evaluates to a call of the function f with the symbolic value s.

The $\mathit{preciseSearch}(s)$ and $\mathit{heuristicSearch}(s)$ operations both call the `solve` query, which returns an instantiation of the symbolic value s (if there is one), and therefore a solution to the search problem that s represents. Rosette's `solve` query is deterministic (in that it always returns the same result when invoked with the same arguments) when used with a

determinstic SAT/SMT solver like Z3. The only difference between *preciseSearch*(s) and *heuristicSearch*(s) is that for *preciseSearch*(s), we know that it can never be called with an undecidable search problem, and thus that it will always either return a solution or indicate that s is empty.

Extraction to Rosette in this simplified fashion leads to problems with the evaluation order of symbolic values. For example, (`if (symbolic-bool) 42 (assert false)`) has the solution 42, while the following term has no solution, because the assertion is executed before the if statement:

```
(let ((x (assert false))) (if (symbolic-bool) 42 x))
```

The Rosette Backends overcome this problem by wrapping all symbolic values in functions that take no arguments (thunks); and evaluating these thunks in the appropriate order.

Specialized ADT Extraction The implementation of the BitVector and Integer ADTs is straightforward. The ADTs' constants and operations are parameterized and extracted to the appropriate Rosette constants and operations.

Using SpaceSearch's specialized ADTs is one way to build efficient solver-aided tools. Another way is to develop solver-aided tools using native Coq data types, and on extraction use another feature of Coq's extraction mechanism: the feature of literally replacing Coq definitions (not parameters) with arbitrary target language expressions. While this approach is arguably less principled, it also has two advantages. First, it enables the efficient use of existing frameworks with SpaceSearch (we used this feature in SaltShaker). Second, it simplifies reasoning because native Coq types can provide judgemental equalities, whereas ADTs can only provide propositional equalities. For example, the native integer $0 + n$ is judgementally equal to n, whereas the ADT Integer *intPlus*(*intZero*, n) is only propositionally equal to n. SpaceSearch supports both approaches, as each has its strengths and weaknesses.

4.4.3 Distributed Places Backend

The Places Backend provides a parallel, distributed implement of SpaceSearch's Callable ADT using the Distributed Places [80] Racket library.

Distributed Places Background The Distributed Places library provides the `dynamic-place` operation that takes the name of a function, and runs the function identified by that name on some thread of some node of a cluster. The `dynamic-place` operation also allows communication between the thread and its caller, by creating a channel that is both passed to the function running on the cluster and returned to the caller of the operation. This communication channel can be used to send and receive messages using the `put` and `get` operations respectively.

$Callable(A, B)$:= $Worker(A, B)$
$call(r)$:= $runWorker(r)$
$callableBind(l, w)$ $\triangleq$ `(bind (map (spawn w) l) get)`

```
spawn := (lambda (w a)
  (let ((ch (dynamic-place (quote worker-place))))
    (put ch w) (put ch a) ch)))

worker-place := (lambda (ch)
  (put ch (runWorker (get ch) (get ch))))
```

Figure 4.11: SpaceSearch Distributed Places Backend.

Extraction to Places The Places Backend implements the Callable ADT as described in Fig. 4.11. The Callable ADT does not implement the Basic ADT, but instead reuses the existing Native Basic ADT implementation. Every $Space(A)$ is thus implemented as a $list(A)$.

The $callableBind(l, w)$ implementation first `spawn`s a new thread with the worker w for every element (solution) a of the list (space) l, then `get`s the list of results generated by each thread, and finally flattens these result lists into one final result list. The `spawn`(w, a) operation runs the worker w on task a and returns the result. This is implemented by first spawning a thread with `dynamic-place`, where we use `quote` to pass the name of the `worker-place` function, and then sending the worker w and task a to that thread. Upon receipt, the `worker-place` function calls the `runWorker` function to run the worker w on the task a, and then sends the result of this call back to via the communication channel.

The Places Backend must also provide instances for the *Callable* type and *call* operation of the Callable ADT. Ideally, the Places Backend would have the same instantiation as the Native Backend: a *Callable* is a function. This would give users of the Backend maximal flexibility by running arbitrary functions with arbitrary inputs and outputs. However, such an implementation is not possible because each callable, its input, and its output are sent over a network, which means that these values have to be *serializable* (i.e. it must be possible to convert them to a list of bits), and functions are *not* serializable, both in Coq and Racket.

It is, however, possible to serialize *statically defined functions*. These are functions that have a globally visible name at compile time; e.g. `worker-place` is a statically defined function, but `(lambda (x) x)` is not. The `dynamic-place` operation takes the name of a statically defined function, sends that name over the network, and runs the function with that name on the thread that it spawns. This means that users of `dynamic-place` can send arbitrary functions, as long as they are defined statically.

The Places Backend exposes this capability of the Places library (sending statically defined functions) in Coq as follows. The Places Backend instantiates *Callable* and *call* with two parameters, *Worker* and *runWorker* (which is a statically defined function) respectively, but the Places Backend does not specify how to extract them. The extraction is specified by the user of the backend. Specifically, a user can define and extract parameters representing elements of the *Worker* type, and then extract *runWorker* to an arbitrary function. For example, a user can specify

$$
\begin{aligned}
&\mathit{succ} : \mathit{Worker}(\mathit{natural}, \mathit{list}(\mathit{natural})) \triangleq \texttt{(quote succ)}\\
&\mathit{sqrt} : \mathit{Worker}(\mathit{integer}, \mathit{list}(\mathit{double})) \triangleq \texttt{(quote sqrt)}\\
&\mathit{runWorker} \triangleq \texttt{(lambda (w a) (cond}\\
&\quad \texttt{((eq? w (quote succ)) (list (+ 1 a)))}\\
&\quad \texttt{((eq? w (quote sqrt)) (list (sqrt a) (- (sqrt a))))))},
\end{aligned}
$$

where $\mathit{callableBind}([1, 4, 9], \mathit{sqrt})$ evaluates to the list $[-1, 1, -2, 2, -3, 3]$ of the square roots of $[1, 4, 9]$, and $\mathit{callableBind}(\ [1, 2, 3], \mathit{succ})$ evaluates to the list $[2, 3, 4]$ of the successors of $[1, 2, 3]$.

A user of the Places Backend must ensure serializability. The *Worker* type in the Callable ADT makes this easy—serializability is ensured when for all A and B: all values of $\mathit{Worker}(A, B)$ are serializable, and $\mathit{Worker}(A, B)$ is only inhabited if all values of A and B are serializable.

The $\mathit{callableBind}(s, r)$ operation can be more efficient than a normal *bind* whenever s has a medium-sized number of solutions that are easily enumerable, and the callable r performs an expensive computation.

4.5 SaltShaker: Verifying x86 Semantics

4.5.1 SaltShaker Overview

RockSalt [58] is a formal checker for the safety of Native Client [99] code, a sandbox developed by Google. Part of this checker is a specification of a subset of the x86 instruction set that powers most desktops and servers today. Since Rocksalt is developed in Coq, it has a relatively small trusted code base, but (among other things) it relies on the correctness of its x86 semantics. We developed SaltShaker, which checks that the RockSalt semantics, for a given set of instructions, is sound with respect to another x86 specification. This case study follows the same methodology as the overview: 1) We formally express the tool's specification, using a proof assistant. 2) We implement a solver-aided tool, using the operations of SpaceSearch. 3) We prove that the solver-aided tool meets its specification, using the denotational semantics of SpaceSearch.

1) SaltShaker Specification SaltShaker checks that the RockSalt semantics of a given instruction i is sound, where soundness is defined as follows: any property P provable in

$rocksalt : (i : instr) \to bv(rocksaltOracleBits(i)) \to state \to state$
$rocksaltOracleBits : instr \to \mathbb{N}$
$spec : (i : instr) \to bv(specOracleBits(i)) \to state \to state$
$specOracleBits : instr \to \mathbb{N}$

$saltShaker(i : instr) : option(state \times specOracleBits(i)) :=$
  $preciseSearch($
    $bind(stateFull, (\lambda s : state.$
    $bind(bvFull(specOracleBits(i)), (\lambda o_s.$
      $\texttt{if}\ \ exists(\lambda o_r : bv(5).\ spec(i, o_s, s) =?$
                                  $rocksalt(i, extend(o_r), s))$
      $\texttt{then}\ \ empty\ \ \texttt{else}\ \ single(s, o))))))$

$state := \{$
  $eax : bv(32);\ ecx : bv(32);\ edx : bv(32);\ ebx : bv(32);$
  $esp : bv(32);\ ebp : bv(32);\ esi : bv(32);\ edi : bv(32);$
  $cf : bv(1);\ pf : bv(1);\ zf : bv(1);\ sf : bv(1);\ of : bv(1)$
$\}$

$stateFull : Space(state) :=$
  $bind(bvFull(32), (\lambda eax'.\ bind(bvFull(32), (\lambda ebx'. \ldots$
    $single(\{eax := eax'; ebx := ebx'; ...\}))))$

Figure 4.12: SaltShaker.

RockSalt about the output of i also holds for the output of any x86 specification compliant CPU running i.

The RockSalt semantics is modeled as $\mathit{rocksalt}(i, o, s)$ (see Fig. 4.12) which, for a given instruction i, oracle o, and input machine state s, returns a new state that captures the result of executing the instruction i.

The oracle o provides $\mathit{rocksaltOracleBits}(i)$ many bits that determine i's *non-deterministic behavior*.[1] This non-deterministic behavior is used by the RockSalt semantics to 1) model *undefined* output; that is, some instructions are free to store arbitrary bits for particular locations, and 2) *over-approximate* some outputs of an instruction if providing a precise semantics is difficult or not needed.

To check that the output of i also holds for the output of any x86 specification compliant CPU running i, SaltShaker requires a formalization of the official Intel x86 specification [34] (a 3,800 page document using English and informal pseudo-code). SaltShaker assumes this formalization is provided as a function *spec*, and a function *specOracleBits* with similar meaning as the RockSalt equivalent. While the *spec* models undefined output, but must *not* over-approximate instructions.

Formally, given an instruction i, SaltShaker then may return that for any $\mathit{cpu} : \mathit{instr} \to \mathit{state} \to \mathit{state}$, which is specification compliant $\exists o, \forall s, \mathit{cpu}(i, s) = \mathit{spec}(i, o, s)$, the following proposition holds:

$$\forall s\ P.\ (\forall o.\ P(\mathit{rocksalt}(i, o, s))) \to P(\mathit{cpu}(i, s))$$

.

2) SaltShaker Solver-Aided Tool Since the soundness property quantifies over the proposition P, it is higher-order and thus cannot be directly encoded in the logic of a first-order solver like Z3. Instead, SaltShaker uses SpaceSearch to search for a start state s and specification oracle o_s such that there is no RockSalt oracle o_r for which the specification $\mathit{spec}(i, o, s_s)$ and RockSalt $\mathit{rocksalt}(i, o, s_r)$ are equal. Figure 4.12 provides all definitions and shows the implementation of SaltShaker.

To make this check practical, we make the following (sound) approximation. Instead of existentially quantifying over the space of all RockSalt oracles, SaltShaker searches over oracles that only provide at most five non-deterministic bits, and extend that oracle to the full size using 0s. This choice is useful because most often only the flag registers are undefined (and there are only five flags in our machine state). Furthermore, the space of all oracles of this kind is finite and small (2^5 to be precise), and therefore the existential quantifier can be replaced by a disjunction.

---

[1] In practice, this number is easy to obtain, as the official Intel specification says exactly how many output bits are undefined.

3) SaltShaker Correctness SaltShaker is incomplete but sound: whenever SaltShaker returns *None*, then for any start state and specification oracle, there is a RockSalt oracle such that the specification and RockSalt are equal, which implies the soundness property. We have proven this formally in the Coq proof assistant. Whenever SaltShaker returns $Some((s, o))$, the tuple (s, o) may be a counter example to RockSalt's soundness.

4.5.2 SaltShaker Evaluation

Our evaluation of SpaceSearch seeks to answer the following research questions:

- **Q1**: Can SpaceSearch be used to build and verify solver-aided tools that are both efficient and effective?
- **Q2**: Can SpaceSearch be used with unchanged, existing Coq developments?
- **Q3**: Are specialized SMT data-structures more efficient than native Coq data-structures?
- **Q4**: Does incrementalizing the search improve performance over repeated searches?

We instantiate *spec* with the semantics found in STOKE [69], a stochastic super-optimizer that uses SMT solvers to prove the equivalence of optimizations on x86 programs. The specification used in STOKE was largely automatically learned [33]. The instantiation is implemented by pretty-printing STOKE's semantics of instructions to Rosette functions (using a small custom extension of STOKE), and then extracting *spec* and *specOracleBits* from that.

RockSalt and STOKE support slightly differently subsets of the machine state, and thus in SaltShaker, *state* consists of the common subset. In particular, this includes the eight 32-bit general purpose registers ($eax, ecx, edx, \ldots$), as well as five 1-bit flags: *cf* (carry), *pf* (parity), *zf* (zero), *sf* (sign), and *of* (overflow). STOKE provides a semantics for x86-64, the 64-bit extension of x86, whereas Rocksalt only supports 32 bits. Because x86-64 is largely backwards compatible, it is sufficient to map the parts of the machine state common to both architectures. However, a mapping is more difficult for memory, as addresses are 32 and 64 bits respectively. At the moment, memory is not part of our machine state.

In x86, every opcode (e.g., `add`) gives rise to many different instruction variants, for different operand sizes (8, 16 or 32 bits) and operand types (constant or register), and every instruction variant can be instantiated with different concrete operands (e.g., `add eax, 8` or `add al, bl`). These are called instruction instantiations (or just instructions, for short),

SaltShaker proofs the soundness of Rocksalt's semantics for a given x86 instruction instantiation (e.g. `add eax, 8`), by showing the equivalence between the instruction's Rocksalt semantics and STOKE semantics for all possible machine states. However, because there is a large number of instruction instantiations (e.g. there are 2^{64} variants of `add`'s 64bit immediate alone), it is infeasible to use SaltShaker to prove the soundness of Rocksalt's semantics for all instruction instantiations.

Instead, we ran SaltShaker on all 40 opcodes that are supported by both RockSalt and STOKE. We tested 15,255 different instruction instantiations, using both random operands (random registers or random constants) as well as constants from a fixed set of "interesting" bit patterns (e.g., 0, -1, 2^n for various n, etc.). Because opcodes behave similarly for all their different instantiation, this provides high confidence that the RockSalt semantics is in fact sound for all instruction instantiations. If SaltShaker is run with the aim of gaining a full soundness guarantee for a particular program, is suffices to proof the soundness of Rocksalt's semantics only for those instructions actually used in the program to be verified (e.g. the Chrome browser consists of about 2 million instructions).

Q1: Using a single computer with an Intel i7-4790K CPU and 32 GiB of memory, verifying our 15,255 instructions took 1.8h (0.43s per instruction), and initially 72.7% percent of instructions showed at least 1 bit where the semantics of RockSalt and STOKE differ. We investigated the differences, consulted the manual to determine the correct behavior and reported all issues to the developers of RockSalt and STOKE. After working with the developers, we were able to trace all of the inconsistencies to 7 underlying issues in RockSalt and 1 issue in STOKE[2]. All of these have been fixed since.

Specifically, RockSalt (1) computed wrong result and flags due to using a location that had already been overwritten (several instructions affected); (2) incorrectly computed on 32-bit values for 16-bit versions of `bsf` and `bsr`; (3) used the wrong bits to compute parity flag (of all instructions with a parity flag); (3) computed wrong flags for addition/subtraction with carry/borrow; (4) computed wrong flags for comparison, addition, and subtraction; (5) computed wrong flags for multiplication; and (7) computed wrong flags for `shld` and `shrd`.

Despite these errors, the implementation of NaCL that was verified using the RockSalt semantics [58] is likely correct, because the bugs that we found were mostly in the computation of flags or were introduced in a refactoring after the release of the verified NaCL implementation (issue 1 above).

STOKE computed the incorrect result for the `rcr` instruction due to a bug in STOKE's pretty-printer, which is used by SaltShaker. The bug cannot be triggered when using STOKE directly to reason about programs.

SaltShaker reported a false positive on 57 instruction instantiations that use the RockSalt oracle to non-deterministically set the instruction's 32-bit results (whereas the flag oracle we use only allows for at most 5 non-deterministic choices). SaltShaker also found 113 instruction instantiations (1 opcode) where the STOKE semantics is over-approximating (i.e. leaves an output unspecified even thought the official Intel semantics does specify its behavior).

Q2: In building SaltShaker, we made only slight modifications to RockSalt (20 LOC). Specifically, we replaced the bit vector extraction to OCaml with an extraction to Rosette (5 LOC), extracted a frequently used combination of bit vector operations to a more efficient

[2]We group failures by the conceptually underlying issue in the source code of the semantics. If a single function computes the parity flag for all instructions with a parity flag, then we consider this a single underlying issue.

implementation (6 LOC), and rewrote a function that was inefficient due to Racket's call by value semantics (9 LOC).

This compatibility with existing Coq frameworks is one of the strengths of SpaceSearch over low-level solver interfaces. The *bind* operator enables this compatibility. Once we constructed the space of all machine states s, we were able to call *bind* on s, and then just write a function for checking that RockSalt's x86 interpreter is equivalent to the specification for a *concrete* machine state.

In fact, SaltShaker can even bind over some parts of an instruction (e.g. all possible values of an immediate operand) and can thus verify billions of instruction instantiations in seconds. We did not use this feature in our evaluation, however, because STOKE, a tool that was developed with an SMT solver in mind, only provides semantics for *concrete* instruction instantiations over *symbolic* machine state.

Q3: We initially tried to verify SaltShaker using Coq's native implementation of bit vectors. While the space of all bit vectors is efficiently searchable, even simple space constructions, like the space of all 32-bit vectors that are equal to 5, cannot be searched efficiently. SpaceSearch's support of SMT data-structures is thus crucial for the construction of efficient solver-aided tools.

Q4: SaltShaker cannot feasibly check all x86 instruction instantiations (the space of 32-bit immediates alone is already too large). Instead, we recommend a user of SaltShaker only check those instruction instantiations that are actually used in the program that is being verified against Rocksalt, rerunning SaltShaker whenever the set of used instructions changes.

To improve the performance of this workflow, we provide a version of SaltShaker that takes as input a list (space) of instructions t that have already been checked and a list (space) of instructions s that need to be checked. SaltShaker then incrementally checks the instructions in s with:

$$incSearch(s, t, \lambda i.optionToSpace(saltShaker(i)))$$

Fig. 4.13 shows the results of the following experiment: Given a test set of 15,255 instructions, we split it into 15 partitions of 1,017 instructions each. We compare the time to run a monolithic search over partitions 1 through n, to run an incremental search over partitions 1 through n assuming that 1 through $n-1$ have already been searched, and to run *saltShaker* on each individual instruction in partition n only.

The results of this experiment show that incremental search is much faster than monolithic search. However, incremental search incurs a slight overhead compared to running the verification function on only the new instructions. This overhead is caused by the (currently) quadratic algorithm used to subtract the spaces.

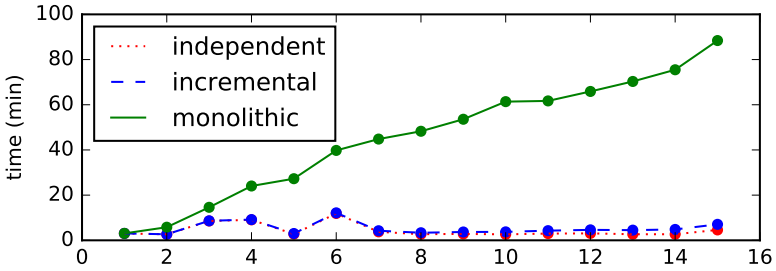

Figure 4.13: Performance gains of incrementalizing SaltShaker.

$$
\begin{aligned}
&\mathit{verifyISP}(c : \mathit{Config}, s : \mathit{Property}) : \mathit{option}(\mathit{Path} \times \mathit{Anno}) := \\
&\quad \mathit{preciseSearch}(\mathit{callableBind}(\\
&\qquad \mathit{bind}(\mathit{fullPath}(c), \lambda p.\mathit{single}((c, s, p))), \mathit{verifyPathWorker}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{verifyPathWorker} : \mathit{Worker}(\mathit{Config} \times \mathit{Property} \times \mathit{Path}, \\
&\quad \mathit{Space}(\mathit{Path} \times \mathit{Anno})) \triangleq (\texttt{quote verifyPathWorker}) \\
&\mathit{runWorker} \triangleq (\texttt{lambda (\_ csp) (verifyPath csp)}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{optionToSpace}(\mathit{Some}(a)) := \mathit{single}(a) \\
&\mathit{optionToSpace}(\mathit{None}) := \mathit{empty}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{verifyPath}(c, s, p) : \mathit{Space}(\mathit{Path} \times \mathit{Anno}) := \\
&\quad \mathit{optionToSpace}(\mathit{preciseSearch}(\\
&\qquad \mathit{bind}(\mathit{fullAnnouncement}, (\lambda a : \mathit{Anno}. \\
&\qquad\quad \texttt{if } \mathit{check}(c, s, p, a) \texttt{ then } \mathit{empty} \texttt{ else } \mathit{single}(t, a))))
\end{aligned}
$$

Figure 4.14: BGProof.

4.6 BGProof: Verifying BGP Configurations

4.6.1 BGProof Overview

Whenever someone wants to watch a video, send an email, or check the news on the Internet, they have to communicate with a server that is potentially on the other side of the world. The Internet itself is a network made up of smaller, interconnected but autonomous networks run by Internet Service Providers (ISPs) like Comcast, MIT, and IBM. For the end-to-end communication over the Internet to work, each ISP must notify all other ISPs of the destinations (like the video/email/news server) that it can communicate with (either directly, or indirectly through another ISP). ISPs do so by sending route announcements via the Border Gateway Protocol (BGP). Once all ISPs have been notified of all destinations, anyone can communicate with anyone else on the Internet.

To ensure reliable and secure communication, ISPs must configure their BGP routers to restrict how route announcements can be used and exchanged. For example, ISPs configure their routers with policies to never use route announcement to bogus destinations like *localhost*. Because BGP gives ISPs freedom to configure their routers, BGP provides very few general guarantees—essentially all desirable properties have to be proven for a particular set of router configurations.

Bagpipe is a solver-aided tool, written in Rosette, that enables ISPs to express desirable properties and automatically check them for a given set of router configurations. This section describes BGProof, a SpaceSearch based, verified reimplementation of a Bagpipe prototype. This case study follows the same methodology as the overview: 1) We formally express the tool's specification, using a proof assistant. 2) We implement a solver-aided tool, using the operations of SpaceSearch. 3) We prove that the solver-aided tool meets its specification, using the denotational semantics of SpaceSearch.

1) Bagpipe Specification BGProof soundly checks that a given set of router configurations c correctly implement a desirable property $s : \mathit{Property}$, i.e. the property s holds for any set of announcements concurrently forwarded along multiple paths through the network defined by c. The correctness property is formally defined in Chapter 2.

2) Bagpipe Solver-Aided Tool Figure 4.14 describes BGProof's checking algorithm, *verifyISP*.[3] Given a desirable property $s : \mathit{Property}$ and a set of router configurations $c : \mathit{Config}$, the *verifyISP* function checks that any announcement $a : \mathit{Anno}$ forwarded along any path $p : \mathit{Path}$ through the network satisfies the desirable property s (where the space of paths *fullPath* is derived from the router configurations c).

BGProof's algorithm uses the Distributed Places Backend to check the desirable property in parallel, for the set of all paths. To do so, *verifyISP* enumerates all paths, and binds each path p (along with the configuration c and desirable property s) to the *verifyPath*

[3]The presentation of this algorithm is simplified. Check Chapter 2 for more details.

function (which is called indirectly via the *Worker/runWorker* mechanism). The *verifyPath* function in turn uses the Rosette Backend to check the desired property symbolically for all announcements a.

3) Bagpipe Correctness BGProof's algorithm is subtle, as it only checks that a single announcement forwarded along a single path satisfies the desired property, but it does not check that multiple announcements forwarded along multiple paths concurrently also satisfy the desired property. However, we have proved BGProof is sound, using a formalization of the pen-and-paper proof discussed in Chapter 2.

4.6.2 BGProof Evaluation

In this evaluation of SpaceSearch, we wanted to answer the following research questions:

- **Q1**: Can SpaceSearch be used to build and verify solver-aided tools that are both efficient and effective?
- **Q2**: Can SpaceSearch's parallelization API improve solver-aided tool performance?
- **Q3**: Can verification with SpaceSearch find bugs in existing solver-aided tools?

Just like in the Chapter 2, we ran an experiment which checked desirable properties for three ISPs: the nation-wide ISP Internet2, the regional ISP BelWü, and the local ISP Selfnet. Their configurations total over 240,000 lines of Cisco and Juniper code. BGProof ran this experiment on Amazon EC2 with 2 instances of type `c3.8xlarge`, each with 32 virtual-cores and 60 GB of memory.

Q1: BGProof ran the experiment in a total of 82h, the cost for which is about $30 using EC2 spot instances. During the verification, BGProof found 19 cases where the configurations did not implement a desirable property, verified 4 desirable properties, and issued no false positives. This is the same as in the original Chapter 2.

Q2: BGProof can check the desired property for each path independently, which means that, apart from a short startup period, BGProof's algorithm is embarrassingly parallel. With over 1,000,000 paths to check, parallelization over paths thus improved performance roughly by the number of CPU cores used during the evaluation.

Q3: Our verification of the Bagpipe prototype identified two bugs. Bagpipe did not verify policy specifications for certain kinds of attributes. It also created incorrect paths in which a router r appeared twice if an update messaged entered and exited the ISP at r without being forwarded within the ISP. These bugs did not seem to lead to user-visible failures, such as missed alarms or false alarms when running Bagpipe over real BGP configurations. The authors of Bagpipe acknowledged and fixed these bugs. We then formally verified Bagpipe's soundness. The proof is 2960 lines of Coq code.

4.7 Related Work

Solver-aided tools Advances in solver technology, including SMT, SAT, and model-finding, have made solver-aided tools a compelling option in many domains; here, we briefly mention a handful of representative examples. Boogie [46] and related tools [45, 44] enable general-purpose verification by compiling verification conditions to SMT queries. Alive [50] and PEC [43] verify compiler optimizations using a solver back end. Batfish [22] verifies data plane properties using a Datalog solver; Vericon [3] verifies policies for software-defined networking controllers using an SMT solver. All of these tools reduce queries in some application domain to queries answerable by an automated solver. But none of these tools come with mechanically-checked proofs that this reduction is sound.

Solver-Aided Domain Specific Languages Solver-aided host languages like Smten [86] and Rosette [81, 82] provide a higher-level interface to underlying solvers, and automatically optimize the orchestration and construction of solver queries. The higher-level interface often leads to less developer effort and order-of-magnitude improvements in code size, while maintaining the performance of hand-crafted solver-aided tools [86].

SpaceSearch itself can be viewed as a solver-aided language, whose interface is inspired by Smten, and which executes solver-aided tools efficiently by extracting them to Rosette. But SpaceSearch extends the state of the art in three ways. First, by exposing its interface in a proof assistant, along with a formal semantics, solver-aided tool developers can verify their optimizations and domain reductions. Second, by providing operations for parallelization and incrementalization, SpaceSearch pushes the boundary on automatically orchestrating solver queries even further. Finally, by splitting its interfaces across various ADTs, SpaceSearch is easily extensible with advanced solver features and provides static guarantees about a solver's timeout behavior.

Integration of Proof Assistants and Solvers Various SAT solver have been built and verified in proof assistants like Coq and Isabelle [47, 60, 53], and they provide an interface against which solver-aided tools can be verified. But verified solvers are currently too slow to be used in most solver-aided tools, and do not provide the additional features of SMT solvers.

Witness checkers [2] alleviate these performance problems by checking the correctness of each individual SMT solver result, instead of verifying the entire solver. However, even a witness checked solver still provides a low-level solver interface, and is thus harder to use than the interface provided by solver-aided languages.

Both verified and witness checked solvers can be used as drop-in replacements for the solver invoked by SpaceSearch.

4.8 Conclusion

This chapter presented SpaceSearch, a library that provides a high-level interface for building and formally verifying solver-aided tools within a proof assistant. In essence, SpaceSearch is a solver-aided host language for proof assistants. SpaceSearch provides a Coq interface against which users build their solver-aided tool and verify that the results of the interface's operations establish the tool's desired high-level properties. Once verified, the tool can be extracted to several backends, including a backend in the Rosette solver-aided language that instantiates the SpaceSearch interface with calls to the Z3 SMT solver. Through its backends, SpaceSearch combines the strong correctness guarantees of a proof assistant with the high performance of modern SMT solvers.

Our evaluation on two solver-aided tools, SaltShaker and BGProof, showed that SpaceSearch can be used to build and verify solver-aided tools that are both efficient and effective. SaltShaker identified 7 bugs in RockSalt and 1 bug in STOKE in under 2h. BGProof scales as well as its unverified hand-crafted predecessor, checking industrial configurations spanning over 240 KLOC and identifying 19 configuration inconsistencies with no false positives. We found that SpaceSearch can be used with almost unchanged existing Coq developments; that SpaceSearch's SMT data-structures are more efficient than native Coq data-structures; and that SpaceSearch's incrementalization and parallelization improve performance. These results show that SpaceSearch is a practical approach to developing efficient, verified solver-aided tools.

BIBLIOGRAPHY

[1] C. J. Anderson et al. "NetKAT: Semantic Foundations for Networks". In: *POPL*. 2014.

[2] M. Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *CPP*. 2011.

[3] T. Ball et al. "VeriCon: Towards Verifying Controller Programs in Software-defined Networks". In: *PLDI*. 2014.

[4] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. `www.SMT-LIB.org`. 2016.

[5] C. Barrett, P. Fontaine, and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.smt-lib.org. 2016.

[6] T. Bates, E. Chen, and R. Chandra. *BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP)*. RFC 4456. 2006.

[7] R. Beckett et al. "Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations". In: *SIGCOMM*. 2016.

[8] *BelWü*. https://www.belwue.de/.

[9] *BGP Feature Guide for the OCX Series*. 2015.

[10] M. Brown. *Pakistan hijacks YouTube*. http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1/. 2008.

[11] C. Cadar, D. Dunbar, and D. Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *OSDI*. 2008.

[12] M. Chiesa et al. "Using routers to build logic circuits: How powerful is BGP?" In: *ICNP*. 2013.

[13] S. Chu et al. "Cosette: An Automated Prover For SQL". In: *CIDR*. 2017.

[14] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *ICFP*. 2000.

[15] E. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *TACAS*. 2004.

[16] R. Coltun et al. *OSPF for IPv6*. RFC 5340. 2008.

[17] J. Cowie. *China's 18-Minute Mystery*. http://research.dyn.com/2010/11/chinas-18-minute-mystery/. 2010.

[18] M. Dobrescu and K. Argyraki. "Software Dataplane Verification". In: *NSDI*. 2014.

[19] J. Dolby, M. Vaziri, and F. Tip. "Finding bugs efficiently with a SAT solver". In: *FSE*. 2007.

[20] R. B. Evans and A. Savoia. "Differential Testing: A New Approach to Change Detection". In: *ESEC-FSE*. 2007.

[21] N. Feamster and H. Balakrishnan. "Detecting BGP Configuration Faults with Static Analysis". In: *NSDI*. 2005.

[22] A. Fogel et al. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015.

[23] N. Foster et al. "Frenetic: A Network Programming Language". In: *ICFP*. 2011.

[24] L. Gao, T. G. Griffin, and J. Rexford. "Inherently safe backup routing with BGP". In: *INFOCOM*. 2001.

[25] L. Gao and J. Rexford. "Stable Internet Routing Without Global Coordination". In: *SIGMETRICS*. 2000.

[26] S. Goldberg. "Why Is It Taking So Long to Secure Internet Routing?" In: *Queue* (2014).

[27] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In: *CAV*. 1997.

[28] T. G. Griffin, F. B. Shepherd, and G. Wilfong. "Policy disputes in path-vector protocols". In: *ICNP*. 1999.

[29] T. G. Griffin, F. B. Shepherd, and G. Wilfong. "The Stable Paths Problem and Interdomain Routing". In: *TON* (2002).

[30] T. G. Griffin and J. L. Sobrinho. "Metarouting". In: *SIGCOMM*. 2005.

[31] A. Guha, M. Reitblatt, and N. Foster. "Machine-verified Network Controllers". In: *PLDI*. 2013.

[32] D. Halperin et al. "Real-time Collaborative Analysis with (Almost) Pure SQL: A Case Study in Biogeochemical Oceanography". In: *SSDBM*. 2013.

[33] S. Heule et al. "Stratified Synthesis: Automatically Learning the x86-64 Instruction Set". In: *PLDI*. 2016.

[34] Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals, Revision 325462-057US*. 2015.

[35] *International Telecommunication Union Statistics*. 2014.

[36] *Internet2 Configurations*. http://vn.grnoc.iu.edu/Internet2/configs/configs.html.

[37] *Internet2 Fees*. http://www.internet2.edu/about-us/membership/.

[38] J. Jeon et al. "Adaptive Concretization for Parallel Program Synthesis". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 2015.

[39] M. Jose and R. Majumdar. "Bug-Assist: assisting fault localization in ANSI-C programs". In: *CAV*. 2011.

[40] *Junos OS: Routing Policies, Firewall Filters, and Traffic Policers Feature Guide for Routing Devices*. 2016.

[41] P. Kazemian, G. Varghese, and N. McKeown. "Header Space Analysis: Static Checking for Networks". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. 2012.

[42] A. S. Koksal et al. "Synthesis of Biological Models from Mutation Experiments". In: *POPL*. 2013.

[43] S. Kundu, Z. Tatlock, and S. Lerner. "Proving Optimizations Correct Using Parameterized Program Equivalence". In: *2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009.

[44] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. "Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers". In: *CAV*. 2009.

[45] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR*. 2010.

[46] K. R. M. Leino. *This is Boogie 2*. Tech. rep. 2008.

[47] S. Lescuyer and S. Conchon. "Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme". In: *FroCoS 2009*. 2009.

[48] G. Li and G. Gopalakrishnan. "Scalable SMT-based Verification of GPU Kernel Functions". In: *FSE*. 2010.

[49] B. Liskov and S. Zilles. "Programming with Abstract Data Types". In: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. 1974.

[50] N. P. Lopes et al. "Provably Correct Peephole Optimizations with Alive". In: *PLDI*. 2015.

[51] D. Madory. *Chinese Routing Errors Redirect Russian Traffic*. http://research.dyn.com/2014/11/chinese-routing-errors-redirect-russian-traffic/. 2014.

[52] P. Mah. *BGP bug found in Juniper router software*. http://www.techrepublic.com/blog/it-news-digest/bgp-bug-found-in-juniper-router-software/. 2007.

[53] F. Marić. "Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL". In: *TCS* 50 (2010).

[54] D. McConnell. *Chinese company 'hijacked' U.S. web traffic*. http://www.cnn.com/2010/US/11/17/websites.chinese.servers/. 2010.

[55] P. McDaniel and K. Butler. "Testing Large Scale BGP Security in Replayable Network Environments". In: *DETER*. 2006.

[56] D. Meyer, J. Schmitz, and C. Alaettinoglu. *Application of Routing Policy Specification Language (RPSL) on the Internet*. 1997.

[57] C. Monsanto et al. "A Compiler and Run-time System for Network Programming Languages". In: *POPL*. 2012.

[58] G. Morrisett et al. "RockSalt: Better, Faster, Stronger SFI for the x86". In: *PLDI*. 2012.

[59] *NetSim: Network Simulator*. http://www.boson.com/netsim-cisco-network-simulator.

[60] D. Oe et al. "versat: A Verified Modern SAT Solver". In: *VMCAI*. 2012.

[61] P. Panchekha and E. Torlak. "Automated Reasoning for Web Page Layout". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016.

[62] P. M. Phothilimthana et al. "Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.

[63] G. D. Plotkin et al. "Scaling Network Verification Using Symmetry and Surgery". In: *POPL*. 2016.

[64] B. Quoitin and S. Uhlig. "Modeling the Routing of an Autonomous System with C-BGP". In: *IEEE Network* (2005).

[65] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. 2006.

[66] G. F. Riley. "Large-scale network simulations with GTNetS". In: *WSC*. 2003.

[67] M. Said et al. "Generating Data Race Witnesses by an SMT-based Analysis". In: *NFM*. 2011.

[68] L. Schaefer. *Deutsche Telekom: 'Internet data made in Germany should stay in Germany'*. http://www.dw.com/en/deutsche-telekom-internet-data-made-in-germany-should-stay-in-germany/a-17165891. 2013.

[69] E. Schkufza, R. Sharma, and A. Aiken. "Stochastic Superoptimization". In: *ASPLOS*. 2013.

[70] *Selfnet*. https://selfnet.de/.

[71] H. Sigurbjarnarson et al. "Push-Button Verification of File Systems via Crash Refinement". In: *OSDI'16*. 2016.

[72] H. Sigurbjarnarson et al. "Push-Button Verification of File Systems via Crash Refinement". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.

[73] R. Singh et al. "Modular Synthesis of Sketches Using Models". In: *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*. 2014.

[74] D. Slane. *2010 Report to Congress of the U.S.–China Economic and Security Review Commission*. 2010.

[75] J. a. L. Sobrinho. "Network Routing with Path Vector Protocols: Theory and Applications". In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. 2003.

[76] A. Solar-Lezama et al. "Combinatorial Sketching for Finite Programs". In: *ASPLOS*. 2006.

[77] M. Suchara, A. Fabrikant, and J. Rexford. "BGP safety with spurious updates". In: *INFOCOM*. 2011.

[78] P. Suter, A. S. Köksal, and V. Kuncak. "Satisfiability modulo recursive programs". In: *SAS*. 2011.

[79] Z. Tatlock and S. Lerner. "Bringing Extensibility to Verified Compilers". In: *PLDI*. 2010.

[80] K. Tew et al. "Distributed Places". In: *TFP*. 2014.

[81] E. Torlak and R. Bodik. "A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages". In: *PLDI*. 2014.

[82] E. Torlak and R. Bodik. "Growing Solver-aided Languages with Rosette". In: *Onward!* 2013.

[83] E. Torlak, M. Vaziri, and J. Dolby. "MemSAT: Checking Axiomatic Specifications of Memory Models". In: *PLDI*. 2010.

[84] D. Turner et al. "California Fault Lines: Understanding the Causes and Impact of Network Failures". In: *SIGCOMM*. 2010.

[85] R. Uhler. *Tutorial 2 - Symbolic Computation*. https://github.com/ruhler/smten/blob/master/tutorials/T2-SymbolicComputation.txt. 2014.

[86] R. Uhler and N. Dave. "Smten with Satisfiability-based Search". In: *OOSPLA*. 2014.

[87] S. Vissicchio, L. Cittadini, and G. Di Battista. "On iBGP Routing Policies". In: *TON* 1 (2015).

[88] A. Voellmy and P. Hudak. "Nettle: A Language for Configuring Routing Networks". In: *DSL*. 2009.

[89] A. R. Voellmy. "Proof of an interdomain policy: a load-balancing multi-homed network". In: *SafeConfig*. 2009.

[90] A. Wang et al. "Analyzing BGP Instances in Maude". In: *FORTE*. 2011.

[91] A. Wang et al. "Formally Verifiable Networking". In: *HotNets*. 2009.

[92] A. Wang et al. "FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing". In: *SIGCOMM*. 2011.

[93] K. Weitz et al. "A Format String Checker for Java". In: *ISSTA*. 2014.

[94] K. Weitz et al. *Bagpipe: Verified BGP Configuration Checking*. Tech. rep. 2016.

[95] K. Weitz et al. *Formal Semantics and Verification for the Border Gateway Protocol*. Tech. rep. 2016.

[96] K. Weitz et al. "Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver". In: *OOPSLA*. 2016.

[97] K. Weitz et al. *SpaceSearch: A Library for Building and Verifying Solver-Aided Tools*. Tech. rep. 2016.

[98] J. R. Wilcox et al. "Verdi: A framework for implementing and formally verifying distributed systems". In: 2015.

[99] B. Yee et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *S&P*. 2009.