Optimizing Horn Solvers for Network Repair

Hossein Hojjat*, Philipp Rümmer†, Jedidiah McClurg‡, Pavol Černý‡, and Nate Foster*
* Cornell University, USA † Uppsala University, Sweden ‡ CU Boulder, USA
{hojjat,jnfoster}@cs.cornell.edu philipp.ruemmer@it.uu.se {jedidiah.mcclurg,pavol.cerny}@colorado.edu

***Abstract*—Automatic program repair modifies a faulty program to make it correct with respect to a specification. Previous approaches have typically been restricted to specific programming languages and a fixed set of syntactical mutation techniques—e.g., changing the conditions of *if* statements. We present a more general technique based on repairing sets of unsolvable Horn clauses. Working with Horn clauses enables repairing programs from many different source languages, but also introduces challenges, such as navigating the large space of possible repairs. We propose a conservative semantic repair technique that only removes incorrect behaviors and does not introduce new behaviors. Our proposed framework allows the user to request the *best* repairs—it constructs an optimization lattice representing the space of possible repairs, and uses a novel local search technique that exploits heuristics to avoid searching through sub-lattices with no feasible repairs. To illustrate the applicability of our approach, we apply it to problems in software-defined networking (SDN), and illustrate how it is able to help network operators fix buggy configurations by properly filtering undesired traffic. We show that interval and Boolean lattices are effective choices of optimization lattices in this domain, and we enable optimization objectives such as modifying the minimal number of switches. We have implemented a prototype repair tool, and present preliminary experimental results on several benchmarks using real topologies and realistic repair scenarios in data centers and congested networks.**

I. Introduction

Program repair is a promising approach to software development that synthesizes a modification to a faulty system to make verification succeed. A number of approaches have been explored in the literature including deductive program repair [1] and automatic patch generation [2], but these often have several limitations.

1) They target *specific types of programs*—e.g., repairing functional Scala programs, or patching PHP programs to make them pass a test suite.
2) They search for *specific types of repairs*—e.g., finding syntactically similar programs by swapping arguments to functions, or modifying the conditions on *if* statements by conjoining (or disjoining) additional conditions.
3) In general, they are not able to find repairs that are optimal with respect to a given objective function.

This paper develops a general approach to the program repair problem. Rather than developing tools customized for specific languages, we utilize a general modeling framework that can be used to encode a wide variety of software artifacts. Additionally, rather than examining specific types of repairs, we explore the space of all possible repairs, and develop techniques for doing this efficiently. Importantly, our tool also has the ability to search for optimal repairs, specified using a domain-specific objective function.

Our approach is based *Horn clauses*—a general framework that is able to model a wide variety of systems and has scalable algorithms and verification tools [3], [4], [5]. In order to use the framework in the context of program repair, we formulate the *Horn clause repair problem*: given a set of Horn clauses that violates a safety invariant, our goal is to produce a repaired set of clauses where the repair is *optimal* with respect to a domain-specific objective function. To find the optimal repair, we must search through a large (in fact, potentially infinite) space of Horn clause repairs. To do this, we construct a finite lattice that abstracts the space of possible repairs—e.g., using *Boolean* and *interval* lattices. Our algorithm for solving Horn-clause optimization problems over finite lattices combines ideas from local search with conflict-driven learning (inspired by SAT and SMT solvers) to prune parts of the optimization lattice that are guaranteed to not contain solutions.

To evaluate our approach, we show how it can be used to solve a variety of real-world problems in the domain of software-defined networking (SDN). To apply our techniques in a given domain, we need a user-defined mapping from the source language to Horn clauses (and vice-versa), and an objective function that specifies in what sense a repair is optimal. In SDN, a *configuration* consists of tables of packet-forwarding rules of the individual switches in the network. Network configurations often contain bugs—e.g., due to loops, black-holes, or access-control violations [6]. We model network configurations using Horn clauses and use an objective function that minimizes the number of switches whose configuration is modified by the repair. We show that our repair framework is able to produce optimal repairs of realistic network configurations efficiently.

II. Motivating Example

The network shown in Fig. 1(a) corresponds to a topology commonly used in large data centers—switches are grouped into three layers: core, aggregation, and ToR (top-of-rack) switches. During normal operation, packets are forwarded from a host upward through aggregation and core switches, and then back downward to the destination host. Although there are physical loops in this network a packet should take only a finite number of hops in any configuration.

In this example, the data center configuration provides service to multiple tenants: hosts H_1, H_2, and H_3 belong to one customer, and H_4 belongs to another customer. Host H_1 sends traffic to H_2 and H_3, but this traffic should not reach H_4, as

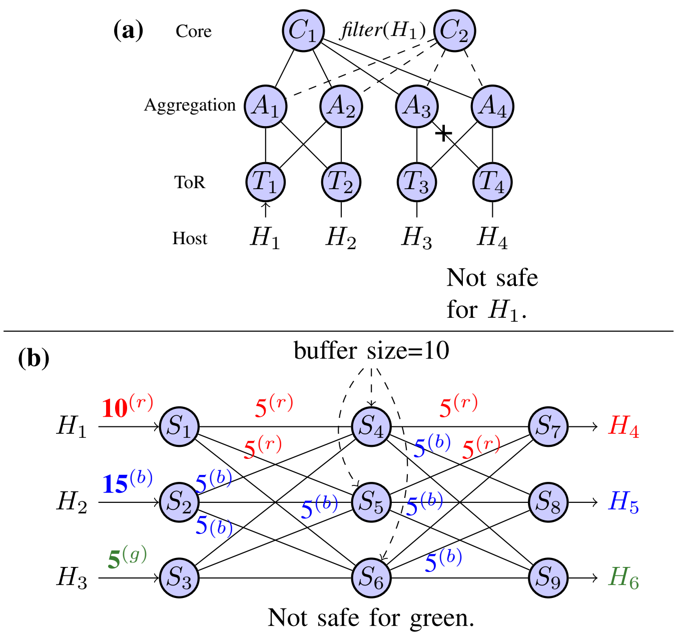

Fig. 1: a) Repair in data center, b) Repair w.r.t. bandwidth and queue sizes.

it is not owned by the customer. To implement this policy, the operator might install a forwarding rule at C_1 to filter packets from H_1 going towards A_4 and also disable the link A_3–T_4 for good measure (in the figure, this disabled link is indicated by a "**+**" symbol.) Now assume that the network operator has brought the core switch C_2 down for maintenance—i.e., the dashed links cannot be traversed by any packet. After the maintenance task has been completed, the network operator decides to bring up C_2 to help balance load within the data center. Unfortunately, this causes the safety requirement to be violated, since there is a new path that forwards H_1 traffic to H_4. Our repair framework interactively helps the network operator bring the network back to safety. The operator can provide as input (i) a description of the network as a high-level transition system, and (ii) a set of required safety properties. Our tool then synthesizes a set of possible repairs which returns the system to safety. As a first solution, the repair engine might suggest that we either disconnect the links A_1–C_2 and A_2–C_2, or take C_2 offline and return the network to its initial state. The network operator could reject this "trivial" repair by stipulating that any repair must not disconnect links. The repair engine might also suggest solutions that rewrite the traffic from H_1 to another type of traffic by modifying packet headers, and the network operator could reject such solutions by stipulating that the repair engine must not modify headers. After providing such restrictions, our tool returns a solution in which filters for H_1 traffic have been added on a number of links: $\{A_1$–C_2, A_2–$C_2\}$, $\{C_2$–A_4, A_4–$T_4\}$, $\{A_4$–$T_4\}$, etc. Our framework uses objective (ranking) functions to guide the repair engine to the "best" answers. For example, the network operator might be interested in solutions that modify the configurations on the smallest number of switches. By providing a suitable objective function, our tool can find an optimal correct solutions—e.g., adding a single filter on the link C_2–A_4 or on the link A_4–T_4.

Another important class of network configuration repairs is related to quantitative measures of bandwidth and traffic. As an example, consider Fig. 1(b) and suppose that each intermediate node can buffer at most 10 units of traffic. The hosts H$_1$, H$_2$, H$_3$ on the left receive 10 units of "red" traffic, 15 units of "blue" traffic, and 5 units of "green" traffic respectively. Red traffic should be sent to H_4, blue traffic to H_5, and green traffic to H_6. In addition, the green traffic must not traverse the intermediate node S_6. Initially, the network operator decides to send 5 units of red traffic to each of S_4 and S_5. She also decides to send 5 units of blue traffic to each of S_4, S_5, and S_6. Unfortunately, this configuration does not allow the green traffic to reach its destination since it cannot flow through S_6, and the buffers of S_4 and S_5 are already full. A correct repair might shift some of red or blue traffic (or both) to S_6 to make room for the green traffic to pass through S_4 or S_5. Our repair engine might generate a solution that sends all green traffic to S_4, and allows the red and blue traffic to be arbitrary divided between S_4, S_5, and S_6, provided the total amount of traffic does not exceed the buffer capacity.

III. Basic Definitions

a) Constraint languages: Throughout this paper, we assume that a first-order vocabulary of *interpreted symbols* has been fixed, consisting of a set Σ_f of fixed-arity function symbols, and a set Σ_p of fixed-arity predicate symbols. The interpretation of Σ_f and Σ_p is determined by a fixed structure (U, I), consisting of a non-empty universe U, and a mapping I that assigns to each function in Σ_f a set-theoretic function over U, and to each predicate in Σ_p a set-theoretic relation over U. As a convention, we assume the presence of an equality symbol "$=$" in Σ_p, with the usual interpretation. Given a set X of variables, a *constraint language* is a set *Constr* of first-order formulae over Σ_f, Σ_p, X. For example, the language of quantifier-free Presburger arithmetic (mainly used in this paper) has $\Sigma_f = \{+, -, 0, 1, 2, \ldots\}$ and $\Sigma_p = \{=, \leq, |\}$, with the usual semantics.

b) Horn Clauses: We consider a set R of uninterpreted fixed-arity relation symbols. The arity of a symbol $p \in R$ is denoted by $\alpha(p)$. A *Horn clause* is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$, where C is a constraint over Σ_f, Σ_p, X; each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in R$ to first-order terms over Σ_f, X; and H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in R$ to first-order terms, or *false*.

H is called the *head* of the clause, and $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. In case $C = true$, we usually omit C and just write $H \leftarrow B_1 \wedge \cdots \wedge B_n$. First-order variables in a clause are implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in S$. Notions like (un)satisfiability and entailment generalize to formulae with relation symbols.

Definition 3.1: Let HC be a set of Horn clauses over relation symbols R. HC is called *(semantically) solvable* (in the structure (U, I)) if there is an interpretation σ of the relation

Algorithm 1: Generalize Procedure

```
Input: Unsolvable Horn clauses $HC$
Result: Solvable Horn clauses $HC$
$ok := \mathit{false}$;
while $\neg ok$ do
    $(ok, CEX) := \textsc{Solve}(HC)$;
    pick $CEX' \subseteq CEX$;  $HC := (HC \setminus CEX')$;
    if $\neg ok$ then
        for $h := (H \leftarrow C \wedge \bigwedge_j B_j) \in CEX'$ do
            $m := \mathit{fresh\_symbol}$;
            $HC := HC \cup (H \leftarrow C \wedge m \wedge \bigwedge_j B_j)$;
```

symbols R as set-theoretic relations such that the universal closure $Cl_\forall(h)$ of every clause $h \in HC$ holds in (U, I), denoted by $\sigma \models HC$; in other words, if the structure (U, I) can be extended to a model of the clauses HC.

We can practically check solvability of sets of Horn clauses by means of *predicate abstraction* [7], [8], using tools like Z3 [9], HSF [7], or Eldarica [5].

IV. Horn-Clause Repair

This section defines the Horn clause repair problem and presents our conservative approach to solving it.

Definition 4.1 (Repair): Let HC be a set of Horn clauses and ϕ a safety invariant, encoded as a Horn clause h_ϕ. Now assume that HC violates the safety invariant—i.e., $HC \cup \{h_\phi\}$ is unsolvable. The set HC' is a repair of HC if (i) $HC' \cup \{h_\phi\}$ is solvable, and (ii) the models I of the first-order variables in HC are a superset of the models I' for HC'.

Given a set of unsolvable Horn clauses, there can be many different strategies for repairing them—i.e., to make the clauses solvable—but it is important that we be able to map repairs back into the problem domain. As an example, in our case studies, we will interested in converting suggested repairs from Horn clauses back to network configurations. The relation symbols in the Horn clause representation will have a specific meaning in the problem domain (e.g., position of the packets or the distribution of traffic in network), and the clauses will have a specific meaning (e.g., forwarding across links). Our repair procedure is conservative in the sense that it does not add clauses, remove clauses, or change the structure of the relation symbols. This makes the translation of repairs back to the problem domain easy—we merely add constraints to the bodies of the clauses to make the clauses more constrained with the goal of removing bad behaviors. We show that this kind of repair corresponds to adding filters or packet-processing rules to switches, and we argue in Section VII that this strategy is not restrictive in the networking domain.

The generalization procedure in Algorithm 1 removes counterexamples to a set of Horn clauses by adding fresh relation symbols to the bodies of a subset (CEX') of the clauses that constitute the counterexample (CEX). The arguments to the fresh relation symbol m are either determined by the problem domain, or use all of the arguments from the existing relation symbols in the head and body of the clause. Algorithm 1 removes every counterexample so that the **while** loop eventually terminates. In the worst case, it conjoins fresh relation symbols to the bodies of all clauses. The fresh relation symbol added to the body of each clause are trivially satisfiable, since the symbols can be set to *false*. However, our Horn optimization problem attempts to synthesize more interesting solutions.

V. Horn-Clause Repair Optimization

We now develop a general framework for formulating and solving optimization problems subject to Horn constraints. The framework is a good match for a range of analysis and synthesis tasks, and in particular, for the purpose of repairing networks. In this setting, side conditions in the form of Horn clauses are used to represent the network, its desired correctness properties, and the space of possible network repairs, while the optimization objective captures preferences about the generated repair—e.g., the smallest number of switches should be updated. Since multiple incomparable solutions may exist in general, we arrange the search space as a lattice.

Definition 5.1 (Optimization lattice): Suppose again that R is a set of uninterpreted fixed-arity relation symbols, and that

$$S_R = \{\sigma : R \to \mathcal{P}(U^*) \mid \sigma(p) \subseteq U^{\alpha(p)}\}$$

is the space of possible interpretations of the R symbols as set-theoretic relations over the universe U. An *optimization lattice* is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a mapping $\mu : L \to \mathcal{P}(S_R)$ from elements of $\langle L, \sqsubseteq_L \rangle$ to sets of interpretations of the R symbols, such that:

1) the bottom element is mapped to $\mu(\bot) = S_R$, the set of all interpretations; and
2) μ is anti-monotonic, i.e., $a \sqsubseteq_L b$ implies $\mu(a) \supseteq \mu(b)$.

The lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ is Horn-definable if there is a function π mapping elements $l \in L$ to finite sets $\pi(l)$ of Horn clauses over relation symbols $R \cup R'$, such that $\mu(l) = \{\sigma|_R \mid \sigma \models \pi(l)\}$ for every $l \in L$.

Given a set HC of Horn clauses, we call a lattice element $l \in L$ *feasible* if there is an interpretation $\sigma \in \mu(l)$ with $\sigma \models HC$; in other words, if the clauses are satisfied by some interpretation associated with l. Since μ is anti-monotonic, feasibility is an anti-monotonic predicate on optimization lattices as well: if a node is infeasible, all of its successors are also infeasible. An element $l \in L$ is *maximal feasible* if l is feasible, but all of its successors are infeasible.

Definition 5.2: A *Horn optimization problem* is defined by a set HC of Horn clauses over relation symbols R, an optimization lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ over R, and a monotonic function $obj : L \to D$ to a totally ordered domain D. A solution is a lattice element $l_{max} \in L$ such that

1) l_{max} is maximal feasible for HC; and
2) $obj(l_{max}) = \max\{obj(l) \mid l \in L \text{ is feasible for } HC\}$.

Example 5.1: Consider the topology shown in Fig. 2 and suppose we want to implement IP multicast from H to I_1 and I_2 with TTL scoping. As background, the TTL (time-to-live) field is initialized to a default value (e.g., 64) and is

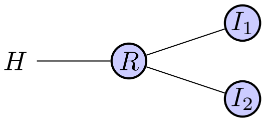

Fig. 2: Multicast router with TTL scoping.

decremented at every hop. Packets with TTL 0 are dropped, which prevents forwarding loops. In TTL scoping, the operator assigns a TTL threshold to each output port on all multicast routers. The routers only forward packets whose TTL value is greater than or equal to the configured threshold. In this example, we will consider a stronger version of TTL scoping with upper and lower bounds. To represent multicasting of a packet to the hosts I_1 and I_2 using Horn clauses, we assign relation symbols R to the router, and I_1, I_2 to the destination hosts (Section VI shows how to encode networks as Horn clauses). Now suppose the network operator wants to disable multicasting by allowing only traffic to I_1 or I_2 (but not both) by adding a filter on TTL values for traffic coming from H. Representing the newly added filter using the relation symbol f, we obtain the following Horn clauses:

$$\begin{aligned} R(t) &\leftarrow f(t) \\ I_1(t') &\leftarrow R(t) \wedge (t' = t-1) \wedge (t' \geq 3) \\ I_2(t') &\leftarrow R(t) \wedge (t' = t-1) \wedge (1 \leq t' \leq 2) \end{aligned}$$

The safety specification is $\mathit{false} \leftarrow I_1(t) \wedge I_2(t')$.

A. Optimization in Boolean Lattices

We discuss two lattices that are frequently useful for defining Horn optimization problems: Boolean lattices, defined as the powerset lattice of some finite set, and interval lattices, which can capture value or address ranges to be enabled or blocked in network repair problems (Sect. V-B). One can construct more complicated optimization lattices—e.g., by taking the Cartesian product of lattices.

We first consider powerset lattices $\langle \mathcal{P}(B), \subseteq \rangle$ of some finite base set B. The bottom element of such a lattice is the empty set $\emptyset$, while the top element is the full set B. This kind of lattice is useful for modeling optimization problems of discrete character, and also covers (weighted) first-order Max Horn SAT problems—i.e., the problem of satisfying a maximum subset of some set of Horn constraints [10].

To convert $\langle \mathcal{P}(B), \subseteq \rangle$ into an optimization lattice, a mapping π_B from $\mathcal{P}(B)$ to sets of Horn clauses can be defined as a homomorphism $\pi_B(A) = \bigcup_{x \in A} \pi_B(x)$, given a π_B that maps every element of B to a (finite) set of Horn clauses. In other words, every element $x \in B$ is responsible for enabling some Horn constraints. The mapping π_B induces an anti-monotonic mapping $\mu_B(A) = \{\sigma \mid \sigma \models \pi_B(A)\}$ to sets of interpretations, and an optimization lattice $(\langle \mathcal{P}(B), \subseteq \rangle, \mu_B)$.

Example 5.2: Recall Example 5.1. We will show how to convert this system into a Horn optimization problem. To start, we choose a base set of clauses

$$B = \begin{Bmatrix} f(t) \leftarrow t < 2, \ f(t) \leftarrow t = 2, \ f(t) \leftarrow t = 3, \\ f(t) \leftarrow t = 4, \ f(t) \leftarrow t > 4 \end{Bmatrix}$$

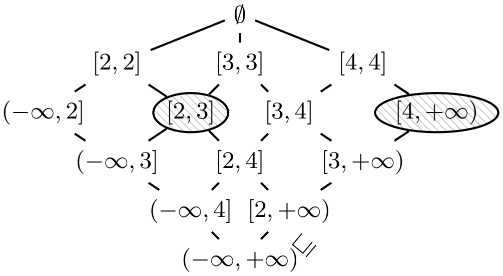

Fig. 3: Example interval lattice $\langle I_2^4, \sqsubseteq_2^4 \rangle$.

and generate a 32-element lattice $\langle \mathcal{P}(B), \subseteq \rangle$. Since each lattice element is identified with a set of Horn clauses, the mapping π_B can be defined as the identity function. Each element of B describes constraints on considered solutions of f, and maximal feasible elements correspond to solutions where f accepts as many TTL values t as possible. The maximal feasible elements are:

$$\begin{aligned} m_1 &= \{f(t) \leftarrow t < 2, \ f(t) \leftarrow t = 2, \ f(t) \leftarrow t = 3\}, \text{ and} \\ m_2 &= \{f(t) \leftarrow t < 2, \ f(t) \leftarrow t = 4, \ f(t) \leftarrow t > 4\}, \end{aligned}$$

i.e., f must filter either values $t \geq 4$, or values $t \in [2, 3]$.

B. Optimization in Interval Lattices

Boolean lattices tend to grow rapidly in practice (as in the previous example). As a more compact (though more coarse-grained) representation, lattices of *intervals* are more useful. Given integers $a, b \in \mathbb{Z}$ ($a \leq b$), we define the lattice $\langle I_a^b, \sqsubseteq_a^b \rangle$:

$$\begin{aligned} I_a^b &= \{\emptyset\} \cup \{(-\infty, \infty)\} \cup \\ &\quad \{[x, y] \mid x, y \in \mathbb{Z}, a \leq x \leq y \leq b\} \cup \\ &\quad \{(-\infty, x], [x, \infty) \mid x \in \mathbb{Z}, a \leq x \leq b\} \\ \sqsubseteq_a^b &= \{(I, J) \in I_a^b \times I_a^b \mid I \supseteq J\} \end{aligned}$$

where $[x, y], (-\infty, x]$, etc., denote non-empty intervals of integers. The bottom element of the lattice is the full interval $(-\infty, \infty) = \mathbb{Z}$, and the top element is the empty set $\emptyset$. As an example, the 14-element lattice $\langle I_2^4, \sqsubseteq_2^4 \rangle$ is given in Fig. 3.

A lattice $\langle I_a^b, \sqsubseteq_a^b \rangle$ can naturally be used to express network repairs that consist of blocking certain ranges (of packet types, addresses, ports, etc.). For instance, given a unary Horn predicate p, a mapping π_p from interval lattice elements to Horn clauses can be defined by

$$\pi_p(I) = \{p(z) \leftarrow z \notin I\} \qquad (\text{for } I \in I_a^b) \ .$$

The clause $\pi_p(I)$ implies that p holds for all values outside of the interval I, while p can be false for values within the interval.[1] As before, π_p induces an anti-monotonic mapping $\mu_p(I) = \{\sigma \mid \sigma \models \pi_p(I)\}$, and therefore gives rise to an optimization lattice $(\langle I_a^b, \sqsubseteq_a^b \rangle, \mu_p)$. Preference of some intervals over others (e.g., minimizing the lower bound of solution intervals) can be captured by adding a suitable monotonic objective function obj.

[1] For the opposite situation, constraining p to be true for all values *within* some interval, a dual lattice can be constructed in which the empty set $\emptyset$ forms the bottom element, and the full interval $(-\infty, \infty)$ is top.

Algorithm 2: Optimization Procedure

```
Input: Horn clauses $HC$, optimization lattice
       $(\langle L, \sqsubseteq_L \rangle, \mu)$, objective function $obj : L \to D$
Result: Set $Sol$ of all solutions of optimization problem
$Sol := \emptyset$;   $SubOpt := \emptyset$;   $B := -\infty$;
while there is a feasible $l \in L$ that is incomparable with $Sol \cup SubOpt$ do
    $m$ or $so$ :=
      $boundedMaximize(HC, (\langle L, \sqsubseteq_L \rangle, \mu), obj, l, B)$;
    if $m$ was returned, and $obj(m) > B$ then
        $SubOpt := SubOpt \cup Sol$;
        $Sol := \{m\}$; $B := obj(m)$;
    else
        $Sol := Sol \cup \{m\}$ or $SubOpt := SubOpt \cup \{so\}$;
return $Sol$;
```

Algorithm 3: $boundedMaximize$ $(HC, (\langle L, \sqsubseteq_L \rangle, \mu), obj, l, B)$

```
Input: Horn clauses $HC$, feasible lattice element $l \in L$,
       optimization bound $B$
Result: $m \in L$  s.t.  $l \sqsubseteq_L m$, $m$ is maximal feasible,
                          and $obj(m) \geq B$   or
        $so \in L$  s.t.  $l \sqsubseteq_L so$, $obj(so) < B$, and all
                          successors of $so$ are infeasible.
1  $upperBound := \top$;
2  for all immediate successors $s$ of $l$ do
3      if $s \sqsubseteq_L upperBound$ then
4          if $s$ is feasible then
5              $l := s$;   Restart loop at line 2;
6          else if $\exists b.\ feasibilityBound(l, s, b)$ then
7              $upperBound := upperBound \sqcap b$;
8              if $obj(upperBound) < B$ then
9                  return $so := upperBound$;
10             if $upperBound$ is feasible then
11                 return $m := upperBound$;
12 if $obj(l) < B$ then  return $so := l$ ;
13 return $m := l$;
```

Example 5.3: We again use the system from Example 5.1, and the lattice $\langle I_2^4, \sqsubseteq_2^4 \rangle$ in Fig. 3 as illustration. With the mapping π_f defined as in (V-B), and the Horn constraints from Example 5.1, the maximal feasible elements are $[2, 3]$ and $[4, \infty)$, which are marked in Fig. 3. Note that those solutions correspond to the ones identified in Example 5.2, but that the interval lattice is more compact than the Boolean lattice.

Since there are multiple maximal feasible elements, we can use a monotonic objective function obj to disambiguate—e.g., such a function could return the negated upper endpoint, which would express a preference for $[2, 3]$ over $[4, \infty)$:

$$obj(I) = \begin{cases} -y & \text{if } I = [x, y] \text{ or } I = (-\infty, y] \\ -\infty & \text{if } I = [x, \infty) \\ \infty & \text{if } I = \emptyset \end{cases}$$

C. Effective Optimization for Finite Lattices

We now present our algorithm for solving Horn optimization problems over finite lattices. The algorithm combines ideas from local search (e.g., [11]) with conflict-driven learning (inspired by SAT and SMT solvers) to prune parts of the optimization lattice that are guaranteed to not contain solutions. The algorithm is partly derived from an earlier search procedure for optimal Craig interpolants [12].

Example 5.4: We first illustrate the procedure using Example 5.1, and the interval lattice in Example 5.3. The two maximal feasible elements in the lattice (Fig. 3) are $[2, 3]$ and $[4, \infty)$. Interval $[2, 3]$ has cost $obj([2, 3]) = -3$, and is the optimal solution (obj from Example 5.3).

Our algorithm starts by choosing an arbitrary feasible lattice element, and then walks upward in the lattice until a maximal feasible element is reached. In the example, we can choose the bottom element $(-\infty, +\infty)$, since if any lattice element is feasible, then so is bottom; suppose that maximizing this element (walking upward as long as feasible successors exist) yields $[2, 3]$, which also happens to be the global optimum.

After identifying $[2, 3]$ as a possible solution, optimality must be verified. For this, we make the observation that every further solution has to be *incomparable* to $[2, 3]$, since elements above $[2, 3]$ are infeasible, and elements below are not maximal. Our procedure therefore picks an arbitrary feasible incomparable element, and then again walks upward towards a maximal feasible element. To find feasible incomparable elements, we enumerate all *minimal* incomparable elements, and check whether any of them is feasible (otherwise, no feasible incomparable element can exist). Here, the two minimal elements incomparable to $[2, 3]$ are $(-\infty, 2]$ and $[3, \infty)$, and we suppose that the latter (the feasible one) is picked.

To walk upward, we check whether $[3, \infty)$ has a feasible successor. Suppose we first consider $[3, 4]$, which turns out to be infeasible. Our algorithm utilizes this information to derive a *feasibility bound:* since $[3, \infty)$ is feasible and $[3, 4]$ infeasible, it follows that every feasible element above $[3, \infty)$ has to be below or equal to $[4, \infty)$, i.e., further search can be bounded by $[4, \infty)$. Since $obj([4, \infty)) = -\infty < obj([2, 3])$, we can conclude that no solution can possibly exist above $[3, \infty)$, and the search must backtrack. Note that it is not relevant whether $[4, \infty)$ itself is feasible.

At this point, the feasibility bound $[4, \infty)$ can be used to prune further search, since no solutions can exist above or below $[4, \infty)$. We search for further feasible elements that are incomparable to both $[2, 3]$ and $[4, \infty)$. The minimal incomparable elements are now $(-\infty, 2]$ and $[3, 4]$, both of which are infeasible. It follows that no further feasible incomparable elements exist, and that $[2, 3]$ is the (unique) solution. ■

The pseudo-code of the optimization procedure is shown in Alg. 2 and 3. The main loop in Alg. 2 maintains a set Sol of solutions, a set $SubOpt$ of blocking elements, and cost B of the best solution so far. In each iteration, Alg. 2 computes a feasible element l that is incomparable to all elements in $Sol \cup SubOpt$ (i.e., neither above nor below any element in

$\mathit{Sol} \cup \mathit{SubOpt}$, line 2), and then searches for a maximal feasible element above l using *boundedMaximize* (line 3).

To update the variable *upperBound* (line 7 in Alg. 3), the algorithm exploits the fact that a feasible lattice element l with an infeasible successor s has been found. Given a pair $l \sqsubseteq_L s$ such that l is feasible and s is infeasible, we define what it means for an element $b \in L$ to be a *feasibility bound*:

$$\mathit{feasibilityBound}(l, s, b) \;\equiv\; \begin{cases} l = s \sqcap b, \text{ and} \\ l \sqsubseteq x \text{ implies } x \sqsubseteq b \text{ for every feasible element } x \in L. \end{cases}$$

Given a feasible element l with infeasible successor s of l, the predicate *feasibilityBound* provides an upper bound b for every feasible successor of l. This allows the subsequent maximization to ignore parts of lattice that are not underneath b. Derivation of feasibility bounds is discussed in Sect. V-D.

Feasibility bounds often enable our procedure to prune away large parts of the search space. As the experiments in Sect. VIII show, the algorithm can in practice handle optimization lattices with more than 10^{30} elements, only needing to inspect a tiny fraction of the lattice to find all solutions. The procedure is furthermore an "anytime procedure," which can at any point provide (possibly sub-optimal) solutions, should time run out. The procedure is also complete:

Theorem 5.1: When applied to a finite optimization lattice, Alg. 2 terminates and returns the set of all solutions.

D. Feasibility Bounds

The predicate *feasibilityBound* can often be defined generically for a lattice $\langle L, \sqsubseteq_L \rangle$, without taking the actual set HC of clauses into account. For Boolean lattices $\langle \mathcal{P}(B), \subseteq \rangle$, correct *feasibilityBound* statements can be derived using the rule

$$\frac{x \notin A}{\mathit{feasibilityBound}(A, A \cup \{x\}, B \setminus \{x\})} \;.$$

For interval lattices $\langle I_a^b, \sqsubseteq_a^b \rangle$, the predicate can be defined by:

R1. $\mathit{feasibilityBound}([x,x], \emptyset, [x,x])$
R2. $\mathit{feasibilityBound}([x,y], [x+1,y], [x,x])$
R3. $\mathit{feasibilityBound}((-\infty,y], [a,y], (-\infty,a])$
R4. $\mathit{feasibilityBound}([x,y], [x,y-1], [y,y])$
R5. $\mathit{feasibilityBound}([x,\infty), [x,b], [b,\infty])$
R6. $\mathit{feasibilityBound}([x,\infty), [x+1,\infty), [x,x])$
R7. $\mathit{feasibilityBound}((-\infty,\infty), [a,\infty), (-\infty,a])$
R8. $\mathit{feasibilityBound}((-\infty,y], (-\infty,y-1], [y,y])$
R9. $\mathit{feasibilityBound}((-\infty,\infty), (-\infty,b], [b,\infty])$

For instance, **R2** says if $[x,y]$ is feasible and $[x+1,y]$ is infeasible, it can be concluded that every feasible interval I above $[x,y]$ must be below (or equal to) $[x,x]$. Clearly, if $I \sqsupseteq_a^b [x,y]$ is feasible, it must be the case that $I \not\sqsupseteq_a^b [x+1,y]$ (since $[x+1,y]$ is infeasible, and feasibility is anti-monotonic), which implies that I must include the value x; $I \sqsubseteq_a^b [x,x]$.

VI. Software-Defined Networking

To demonstrate the usefulness of our approach in practice, we apply it in the context of software-defined networking. In this paper, we consider a packet to be a bounded natural number $\mathit{pkt} \in \mathbb{N}$ $(0 \leq \mathit{pkt} < 2^b)$ where b is the total required number of bits to represent the header fields. A packet with a value outside the admitted bound (e.g., $\mathit{pkt} = -1$) is an invalid packet, and any switch immediately drops it.

A switch has a forwarding table consisting of a set of rules. Each rule has a pattern which is a predicate on headers. When a packet matches a pattern, the switch forwards it to an output port (with possibly updates to some header fields). If there are multiple matching rules, the switch is free to pick any of them, and if there are no matching rules, it drops the packet.

A. Single-packet Transition System

A *single-packet transition system* is a tuple $S = \langle \mathit{pkt}, \mathit{trc}, Q, Q_i, Q_f, T \rangle$ in which $\mathit{pkt} \in \mathbb{N}$, $\mathit{trc} : [Q]$ (trace of states); Q is a set of states ($Q_i \subseteq Q$ start, $Q_f \subseteq Q$ final); $T \in \; (Q \times \Phi(\mathit{pkt}, \mathit{pkt}') \times Q)$ is the transition relation from state q to q', written as $q \overset{\phi}{\to} q'$. The label $\phi \in \Phi$ is a Presburger formula over pkt (value of pkt in q) and pkt' (value of pkt in q'). Each state $q \in Q$ of a single-packet transition system normally corresponds to a switch in the network. We show the source of a transition with src, destination with dst, and label with ℓ. A transition updates the trc value $\mathit{trc}' = \mathit{trc} \lhd q'$.

a) Drop State: We assume that there is a special state $q_d \in Q_f$ that represents dropping a packet. For any $q \notin Q_f$, there is a transition to the drop state for the invalid packets: $q \xrightarrow{(\mathit{pkt}<0 \;\vee\; \mathit{pkt}\geq 2^b)} q_d$. The condition on this transition is weaker for a switch that drops more packets in the space of admissible packets.

b) Local Progress: We assume that for any packet pkt, there is always a transition out of a non-final state:

$$\forall q{\notin}Q_f . \forall \mathit{pkt}{\in}\mathbb{N}. \exists t{\in}T. \exists \mathit{pkt}'{\in}\mathbb{N}. (\mathit{src}(t){=}q) \wedge \ell(t)(\mathit{pkt}, \mathit{pkt}')$$

Intuitively, this means that a non-final state either forwards a packet to the next or the drop state. The local progress property along with the drop state helps us specify reachability in terms of safety constraints. If there are no forwarding loops in a network, having local progress ensures that a packet is either received at the drop state or a final host.

c) Path: A path of a single-packet transition system $S = \langle \mathit{pkt}, \mathit{trc}, Q, Q_i, Q_f, T \rangle$ is a sequence $\langle \mathit{pkt}_0, \mathit{trc}_0, q_0 \rangle \overset{\phi}{\to} \langle \mathit{pkt}_1, \mathit{trc}_1, q_1 \rangle \overset{\phi'}{\to} \cdots \overset{\phi^{(n-1)}}{\to} \langle \mathit{pkt}_n, \mathit{trc}_n, q_n \rangle$ where $q_0 \in Q_i$ is an initial state, and $q_n \in Q_f$ is a final state.

d) Invariant: A single-packet transition system $S = \langle \mathit{pkt}, \mathit{trc}, Q, Q_i, Q_f, T \rangle$ satisfies an invariant $\psi(\mathit{trc})$ (written as $S \models \psi$) if and only if every path trc satisfies ψ.

e) Horn-Clause Translation: We associate a relation symbol s_q with arity 2 to any $q \in Q$. The following Horn clause represents the transition relation $q \overset{\phi}{\to} q'$:

$$s_{q'}(\mathit{pkt}', \mathit{trc}') \leftarrow s_q(\mathit{pkt}, \mathit{trc}) \wedge \phi(\mathit{pkt}, \mathit{pkt}') \wedge (\mathit{trc}' = \mathit{trc} \lhd q') \;.$$

If in a start state q_i, a packet has an initial value pkt_i, then we add the following clause: $s_{q_i}(\mathit{pkt}, \mathit{trc}) \leftarrow (\mathit{pkt} = \mathit{pkt}_i)$.

We can describe some invariants of interest in the network domain using Horn clauses, such as the following.

Non-dropping—no packet is dropped: $\mathit{false} \leftarrow q_d(\mathit{pkt}, \mathit{trc})$.
Non-Reachability—for a non-dropping network, the traffic from a given source q_a must not reach a certain destination: $\mathit{false} \leftarrow q_f(\mathit{pkt}, \mathit{trc}) \wedge q_a \neq \mathit{trc.head}$.
Way-pointing—a specific switch q_a must be traversed: $\mathit{false} \leftarrow q_f(\mathit{pkt}, \mathit{trc}) \wedge q_a \notin \mathit{trc}$.

B. Bandwidth Constraints

In some repair scenarios, the properties of interest are related to bandwidth capacities of the links, congestion avoidance, or buffer overflows in packet queues. To model traffic sizes, we use a technique based on counter abstraction. The basic idea is to use tokens to represent the sizes of the flows that enter the network. Tokens here are merely used to model bandwidth usage and should not be confused with the actual packets. The token counters get updated whenever a flow of packets travels through a link.
A *bandwidth transition system* is a tuple $S = \langle Q, Q_i, Q_f, M, M_0, T\rangle$ in which Q is a set of states ($Q_i \subseteq Q$ start, $Q_f \subseteq Q$ final); M is the distribution of the traffic tokens in the network at any time. For a state $q \in Q$ and a traffic type $\tau \in \mathbb{N}$, the value of $M(q, \tau)$ is the number of tokens of traffic type τ at state q, M_0 is the initial distribution of tokens in the network, $T \in (Q \times \Phi(M, M') \times Q)$ is the transition relation from state q to q', written as $q \xrightarrow{\phi} q'$. The label ϕ determines how the distribution of the tokens $M(q)$ and $M(q')$ changes during the transition.
a) Invariant: Invariants in bandwidth transition systems are similar to single-packet transition systems, the difference being that the property ψ talks about the distributions of tokens in the network. As an example, if a state q is not safe for traffic type *typ*, then an invariant for the network specifies the number of tokens for *typ* to be 0 at any time.
b) Horn-Clause Translation: Assume that there are n types of traffic in a network, namely $\{\mathit{typ}_1, \cdots, \mathit{typ}_n\}$. For a bandwidth transition system $S = \langle Q, Q_i, Q_f, M, M_0, T\rangle$, we use a single relation symbol s that holds counters to store the number of tokens for each flow at any $Q = \{q_1, \cdots, q_m\}$ position in the network: $s(c_{q_1}^{\mathit{typ}_1}, \cdots, c_{q_1}^{\mathit{typ}_n}, \cdots, c_{q_m}^{\mathit{typ}_1}, \cdots, c_{q_m}^{\mathit{typ}_n})$. Similar to the single-packet case, we add clauses to capture the transitions of T and the updates to M.

VII. Network Repair Problem

A network repair problem $\mathcal{U} = (S, \psi, \rho)$ has the following inputs: S is a single-packet or bandwidth transition system, ψ is an invariant such that $S \not\models \psi$, and objective ρ is a ranking on the space of transition systems. A solution to the repair problem updates the transition relation T in S to obtain S', such that $S' \models \psi$ and if S'' is another transition system that satisfies the above conditions then $\rho(S') \geq \rho(S'')$. Objectives of interest in networking are, e.g., touching a minimal number of switches, filtering fewer traffic paths in the network, etc.

Let HC be the translation of S to Horn clauses. We formulate a Horn optimization problem for single-packet and bandwidth transition systems.

Benchmarks	#Nodes	#Links	#Rels.	#Lattice	#Eld	Time(s)
Gridnet	9	20	–	–	–	–
Cesnet200304	29	33	3	2.22×10^{10}	145	4.98
Arpanet19706	9	10	3	2.22×10^{10}	91	2.98
Oxford	20	26	8	3.89×10^{27}	664	16.70
Garr200902	54	71	6	4.92×10^{20}	3045	107.62
Getnet	7	8	2	7.90×10^{6}	61	1.45
Surfnet	50	73	3	2.22×10^{10}	101	3.49
Itnet	11	10	1	2.81×10^{3}	17	0.18
Garr199904	23	25	1	2.81×10^{3}	19	0.33
Darkstrand	28	31	5	1.75×10^{17}	425	14.81
Carnet	44	43	2	7.90×10^{6}	37	0.49
Atmnet	21	22	1	2.81×10^{3}	15	0.67
HiberniaCanada	13	14	11	8.63×10^{37}	1795	84.56
Evolink	37	45	1	2.81×10^{3}	14	0.20
Dfn	58	87	–	–	–	–
Ernet	30	32	4	6.23×10^{13}	140	4.94
Bren	37	38	6	4.92×10^{20}	974	25.14
Niif	36	41	2	7.90×10^{6}	48	0.92
Renater2001	24	27	3	2.22×10^{10}	101	3.56
Latnet	69	74	2	7.90×10^{6}	47	0.64

Fig. 4: Repairing 20 benchmarks from Topology Zoo [13] on a 1.4 GHz AMD Opteron$^{\mathrm{TM}}$ Processor with 32 Gigabytes of memory (time-out is set to 2 minutes in this experiment).

a) Single-packet Transition System: We use Alg. 1 to add new fresh symbols $m(\mathit{pkt})$ to the bodies of some clauses in HC to get HC' (or $m(\mathit{pkt}, \mathit{pkt}')$ when the source switch can rewrite packets). Assuming the size of the header in a packet is b, to each $m(\mathit{pkt})$ we associate an interval lattice $I_0^{2^b-1}$ (e.g., lattice in Fig. 3) that represents the packets that should be filtered out. The lattice of repair solutions is the product of all the interval lattices for m relations. For an objective function ρ that more highly ranks solutions that filter fewer traffic types, we use an objective function obj in the Horn optimization problem that assigns the lowest rank to the solution that assigns $(-\infty, \infty)$ to every m.
b) Bandwidth Transition System: The formulation of network repair in this case is similar to single-packet transition systems, with the difference being that the lower-bound of the intervals for added fresh symbols is 0, and the upper-bound is the maximum number of tokens for each type.
c) Generality of Repair: We assume that the reason for a violation of the safety property is that the network configuration is under-constrained. In other words, there are forwarding behaviors in the network that should be restricted—e.g., by adding filters on the links. Furthermore, we assume that during the repair procedure, no new switches or links are added to the network. These assumptions are not overly restrictive in practice—if the network operator wants to add new switches to the network, she can connect the new switch to the rest of the network without any constraints: the new switch behaves as a repeater. It is also possible to add links to the network and send all the traffic through the new links. The repair procedure may then restrict forwarding of packets through these links.

VIII. Implementation and Experiments

We have implemented the prototype tool *Marham* (Minimal repair for Horn clause systems) that operates on top of the

Eldarica [5] verifier. To evaluate Marham, we considered two main questions. First, we studied the applicability of our approach to several interesting repair scenarios from the network domain. Second, we benchmarked the performance of our tool against a dataset of real topologies. For the first question, we considered the network properties introduced in Section II using the data center topology shown in Fig. 1(a) with a non-dropping criterion. We used $[0, 7]$ as intervals, with 0 representing SSH traffic. Our tool found the correct repair by suggesting that a filter be added on A_4. We also repaired a way-pointing scenario by removing the path through the waypoint and then repairing. For the Fig. 1(b) example, Marham produces a repair by sending the green traffic to s_4.

For the second question, for topologies from the Internet Topology Zoo set [13], we generated Horn clauses to connect a set of random vertices (Topology Zoo contains data network topologies from around the world). We non-deterministically selected a node and made it unsafe for a certain flow by adding a clause specifying that this type of traffic should not reach that particular point. We considered the objective function that minimizes the number of filtered paths relative to the original configuration. Table in Fig. 4 shows the results of executing Marham for repairing 20 representative topologies from the Topology Zoo. The table reports the number of nodes, links, and synthesized relation symbols, as well as the size of the lattice, number of calls to Eldarica, and total time.

IX. Related Work

Although *optimizing SMT solvers* have been proposed in previous work [14], to the best of our knowledge, our framework is the first to provide such optimization functionality in a Horn clause solver. Our approach also differs from *MaxSAT solvers,* which search for solutions satisfying maximum sets of clauses, in the generic way that optimization lattices and objectives are formulated.

A number of approaches to repair are based on finding **similar expressions**—e.g., by using a game-based approach in which winning strategies correspond to choosing a correct expression [15], adding nondeterministic expressions at problematic locations and using a SAT solver to find a deterministic program that satisfies the specification [16], using a cost function to select a correct expression [17], or using deductive approaches based on *guided* synthesis [1].

Other repair approaches target **specific languages** (e.g., Boolean programs, which are essentially a restricted form of C programs [18]) or **specific types of fixes** (e.g., atomicity violations [19]). Our repair framework is different in that (i) it is not language-specific so it can be used in a variety of settings, (ii) it places no restrictions on the type of repairs that can be made, and (iii) it allows the programmer to repair with respect to a safety property as well as an objective function.

In regards to the problem of synthesizing repairs for network configurations, the closest to our work is [20]. Our work is more general in several aspects. Their specification language is based on regular expressions, and updates are specified as end-to-end paths from the old to new configuration using regular expressions. Our Horn clause specification language gives us the power to consider more general properties such as loop freedom, bandwidth constraints, etc.

X. Conclusion

This paper introduces a framework for repairing a set of Horn clauses, and presents an optimization technique to search the space of repairs efficiently. We have implemented our repair engine in the Marham tool—to investigate its applicability to real world problems, we perform experiments using the Internet Topology Zoo dataset. The generality of Horn clauses in describing problems from various domains makes our proposed approach suitable for repairing various systems.

Acknowledgments

We thank the FMCAD reviewers for helpful and constructive comments. Our work is supported by the National Science Foundation under grants CNS-1111698, CNS-1413972, CCF-1421752, CCF-1422046, CCF-1253165, and CCF-1535952; the Office of Naval Research under grant N00014-15-1-2177; and gifts from Cisco, Facebook, Fujitsu, Google, and Intel.

References

[1] E. Kneuss, M. Koukoutos, and V. Kuncak, “Deductive program repair,” in *CAV*, pp. 217–233, 2015.

[2] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *POPL*, pp. 298–312, ACM, 2016.

[3] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, “Horn clause solvers for program verification,” in *Fields of Logic and Computation II*, pp. 24–51, Springer, 2015.

[4] N. Bjørner, K. McMillan, and A. Rybalchenko, “On solving universally quantified horn clauses,” in *SAS*, pp. 105–125, Springer, 2013.

[5] H. Hojjat, F. Konečný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, “A verification toolkit for numerical transition systems (tool paper),” in *FM*, 2012.

[6] J. McClurg, H. Hojjat, P. Cerný, and N. Foster, “Efficient synthesis of network updates,” in *PLDI*, pp. 196–207, 2015.

[7] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, “Synthesizing software verifiers from proof rules,” in *PLDI*, 2012.

[8] P. Rümmer, H. Hojjat, and V. Kuncak, “Disjunctive interpolants for Horn-clause verification,” in *CAV*, pp. 347–363, 2013.

[9] K. Hoder and N. Bjørner, “Generalized property directed reachability,” in *SAT*, 2012.

[10] B. Jaumard and B. Simeone, “On the complexity of the maximum satisfiability problem for Horn formulas,” *Information Processing Letters*, vol. 26, no. 1, pp. 1 – 4, 1987.

[11] H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., 2004.

[12] J. Leroux, P. Rümmer, and P. Subotic, “Guiding Craig interpolation with domain-specific abstractions,” *Acta Informatica*, 2015.

[13] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, “The internet topology zoo,” *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 9, pp. 1765–1775, 2011.

[14] N. Bjørner, A. Phan, and L. Fleckenstein, “νz - an optimizing SMT solver,” in *TACAS*, pp. 194–199, 2015.

[15] B. Jobstmann, A. Griesmayer, and R. Bloem, “Program repair as a game,” in *CAV*, pp. 226–238, Springer, 2005.

[16] D. Gopinath, M. Z. Malik, and S. Khurshid, “Specification-based program repair using SAT,” in *TACAS*, pp. 173–188, Springer, 2011.

[17] R. Samanta, O. Olivo, and E. A. Emerson, “Cost-aware automatic program repair,” in *SAS*, pp. 268–284, Springer, 2014.

[18] A. Griesmayer, R. Bloem, and B. Cook, “Repair of boolean programs with an application to C,” in *CAV*, pp. 358–371, Springer, 2006.

[19] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, “Automated atomicity-violation fixing,” *PLDI*, vol. 46, no. 6, pp. 389–400, 2011.

[20] S. Saha, S. Prabhu, and P. Madhusudan, “Netgen: Synthesizing data-plane configurations for network policies,” in *SOSR ’15*, 2015.