Syntax-Guided Automated Program Repair for Hyperproperties

Raven Beutner[1(✉)], Tzu-Han Hsu[2], Borzoo Bonakdarpour[2], and Bernd Finkbeiner[1]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{raven.beutner,finkbeiner}@cispa.de
[2] Michigan State University, East Lansing, MI, USA
{tzuhan,borzoo}@msu.edu

Abstract. We study the problem of automatically repairing infinite-state software programs w.r.t. temporal hyperproperties. As a first step, we present a repair approach for the temporal logic HyperLTL based on symbolic execution, constraint generation, and syntax-guided synthesis of repair expression (SyGuS). To improve the repair quality, we introduce the notation of a *transparent* repair that aims to find a patch that is as close as possible to the original program. As a practical realization, we develop an *iterative* repair approach. Here, we search for a sequence of repairs that are closer and closer to the original program's behavior. We implement our method in a prototype and report on encouraging experimental results using off-the-shelf SyGuS solvers.

1 Introduction

Hyperproperties and *program repair* are two popular topics within the formal methods community. Hyperproperties [14] relate multiple executions of a system and occur, e.g., in information-flow control [56], robustness [12], and concurrent data structures [10]. Traditionally, automated program repair (APR) [25,28] attempts to repair the *functional* behavior of a program. In this paper, we, for the first time, tackle the challenging combination of APR and hyperproperties: given an (infinite-state) software program $\mathbb{P}$ and a violated hyperproperty φ, repair $\mathbb{P}$ such that φ is satisfied.

As a motivating example, consider the data leak in the EDAS conference manager [1] (simplified in Fig. 1). The function `display` is given the current phase of the review process (`phase`), paper title (`title`), session (`session`), and acceptance decision (`decision`), and computes a string (`print`) that will be displayed to the author(s). As usual in a conference management system, the displayed string should not leak information other than the `title`, unless the review process has been concluded. We can specify this *non-interference* policy as a hyperproperty in HyperLTL [13] as follows:

$$\forall \pi_1.\, \forall \pi_2.\, \big(\texttt{phase}_{\pi_1} \neq \texttt{"Done"} \wedge \texttt{phase}_{\pi_2} \neq \texttt{"Done"} \wedge \texttt{title}_{\pi_1} = \texttt{title}_{\pi_2}\big) \rightarrow \bigcirc\big(\texttt{print}_{\pi_1} = \texttt{print}_{\pi_2}\big). \qquad (\varphi_{\text{edas}})$$

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14683, pp. 3–26, 2024.
https://doi.org/10.1007/978-3-031-65633-0_1

That is, for any *two* execution traces π_1, π_2 of `display` that, initially (i.e., at the first `observe` statement in line 3), have not reached the `"Done"` phase (i.e., `phase` $\neq$ `"Done"`) and agree on the title, should, at the second `observe` in line 10, agree on the value of `print`. It is straightforward to observe that function `display` violates φ_{edas}. The code *implicitly* leaks the acceptance decision by printing the session iff the paper is accepted. A natural question to ask is whether it is possible to automatically *repair* the `display` function such that φ_{edas} is satisfied.

```
1  display(string phase, string title,
2   string session, string decision) {
3     observe
4     decision = decision
5     if (decision == "Accept") {
6        print = title + session
7     } else {
8        print = title
9     }
10    observe
11 }
```

Fig. 1. Information leak in EDAS conference management system.

Constraint-Based Repair for Hyperproperties. As a first contribution, we propose a constraint-based APR approach for HyperLTL. Similar to existing constraint-based APR methods for functional properties [46], we rely on fault localization to identify potential repair locations (e.g., line 4 of our example in Fig. 1). We then replace the repair locations with a fresh function symbol; use symbolic execution to explore symbolic paths of the program; and generate repair constraints on the inserted function symbols. We show that we can use the *syntax-guided synthesis* (SyGuS) framework [2] to express (and solve) the repair constraints for HyperLTL properties with an *arbitrary* quantifier prefix.

Many Solutions. The main challenge in APR for hyperproperties lies in the large number of possible repair patches; a problem that already exists when repairing against functional properties [50] but is even more amplified when targeting hyperproperties. Different from functional specification, hyperproperties do not reason about the concrete functional (trace-level) behavior of a program, and rather express abstract relations between multiple computation traces. For example, information-flow policies such as observational determinism [56] can be checked and applied to arbitrary programs, regardless of their functional behavior. In contrast to functional trace properties, we thus cannot partition the set of all program executions into "correct" executions (i.e., executions that already satisfy the trace property and should be preserved in the repair) and "incorrect" executions. Instead, we need to alter the *set* of all program executions such that the executions together satisfy the hyperproperty, leading to an even larger space of potential repairs. Moreover, within this large space, many repairs trivially satisfy the hyperproperty by severely changing the functional behavior of the program, which is usually not desirable.

In our concrete example, the φ_{edas} property implicitly reasons about the (in)dependence between `phase`, `title`, and `print` but does not impose *how* the (in)dependence is realized functionally. If we apply our basic SyGuS-based repair approach, i.e., search for *some* repair of line 4 that satisfies φ_{edas}, it will immediately return a trivial repair patch: `decision ="Reject"`. This repair

```
if (phase == "Done"){
   decision = decision
} else {
   decision = "Reject"
}
```

(a)

```
if ((phase == "Done") or (decision != "Accept")){
   decision = decision
} else {
   decision = "Reject"
}
```

(b)

Fig. 2. Repair candidates discovered by our iterative repair.

simply sets the `decision` to some string not equal to `"Accept"` (we use `"Reject"` here for easier presentation). While this certainly satisfies our information-flow requirement, it does not yield a desirable implementation of `display` because the session is never displayed.

Transparent Repair. To tackle this issue, we strengthen our repair constraints using the concept of *transparency* (borrowed from the runtime enforcement literature [45]). Intuitively, we search for a repair that not only satisfies the hyperproperty but preserves as much functional behavior of the original program as possible. We show that we can integrate this within our SyGuS-based repair constraints. In the extreme, *full transparency* states that a repair is only allowed to deviate from the original program's behavior if absolutely necessary, i.e., only when the original behavior is part of a violation of the hyperproperty.

Iterative Repair. In the setting of hyperproperties, full transparency is often not particularly useful. It strictly dictates what traces can be changed by a repair, potentially resulting in the absence of a repair (within a given search space). In other instances (including the EDAS example), many paths (in the EDAS example, *all* paths) take part in some violation of the hyperproperty, allowing the repair to intervene arbitrarily. We introduce a more practical repair methodology that follows the same objective as (full) transparency (i.e., preserve as much original program behavior as possible). Our method, which we call *iterative repair*, approximates the global search for an optimal repair by a step-wise search for repairs of increasing quality. Concretely, starting from some initial repair, we iteratively try to find repair patches that preserve *more* original program behavior than our previous repair candidate. We show that we can effectively encode this into SyGuS constraints, and existing off-the-shelf SyGuS solvers can handle the resulting queries in many challenging instances. Notably, while some APR approaches (for functional properties) also try to find repairs that are close to the original program, they often do so heuristically. In contrast, our iterative repair constraints *guarantee* that the repair candidates strictly improve in each iteration. See Sect. 7 for more discussion.

Iterative Repair in Action. Coming back to our initial EDAS example, we can use iterative repair to improve upon the naïve repair `decision = "Reject"`. When using our iterative encoding, we find the improved repair solution in Fig. 2a that (probably) best mirrors the intuition of a programmer (cf. [47]):

This repair patch only overwrites the decision in cases where the phase does not equal "Done". In particular, note how our iterative repair finds the *explicit* dependence of decision on phase (in the form of a conditional) even though this is only specified *implicitly* in φ_{edas}. In a third iteration, we can find an even closer repair, displayed in Fig. 2b: This repair only changes the decision if the review process is not completed *and* the decision equals "Accept".

Implementation. We implement our repair approach in a prototype named HyRep and evaluate HyRep on a set of repair instances, including k-safety properties from the literature and challenging information-flow requirements.

Structure. Section 2 presents basic preliminaries, including our simple programming language and the formal specification language for hyperproperties targeted by our repair. Section 3 introduces our basic SyGuS-based repair approach, and we discuss our transparent and iterative extensions in Sects. 4 and 5, respectively. We present our experimental evaluation in Sect. 6 and discuss related work in Sect. 7.

2 Preliminaries

Given a set Y, we write Y^* for the set of finite sequences over Y, Y^ω for the set of infinite sequences, and $Y^\star := Y^* \cup Y^\omega$ for the set of finite and infinite sequences. For $t \in Y^\star$, we define $|t| \in \mathbb{N} \cup \{\infty\}$ as the length of t.

Programs. Let X be a fixed set of program variables. We write $\mathcal{E}_\mathbb{Z}$ and $\mathcal{E}_\mathbb{B}$ for the set of all arithmetic (integer-valued) and Boolean expressions over X, respectively. We consider a simple (integer-valued) programming language

$$\mathbb{P}, \mathbb{Q} := \texttt{skip} \mid x = e \mid \texttt{if}(b, \mathbb{P}, \mathbb{Q}) \mid \texttt{while}(b, \mathbb{P}) \mid \mathbb{P} \fatsemi \mathbb{Q} \mid \texttt{observe}$$

where $x \in X$, $e \in \mathcal{E}_\mathbb{Z}$, and $b \in \mathcal{E}_\mathbb{B}$. Most statements behave as expected. Notably, our language includes a dedicated observe statement, which we will use to express *asynchronous* (hyper)properties [5,11,29]. Intuitively, each observe statement causes an observation in our temporal formula, and we skip over unobserved (intermediate) computation steps (see also [7]).

Semantics. Programs manipulate (integer-valued) stores $\sigma : X \to \mathbb{Z}$, and we define $\mathit{Stores} := \{\sigma \mid \sigma : X \to \mathbb{Z}\}$ as the set of all stores. Our (small-step) semantics operates on configurations $C = \langle \mathbb{P}, \sigma \rangle$, where $\mathbb{P}$ is a program and $\sigma \in \mathit{Stores}$. Reduction steps have the form $C \xrightarrow{\mu} C'$, where $\mu \in \mathit{Stores} \cup \{\epsilon\}$. Most program steps have the form $C \xrightarrow{\epsilon} C'$ and model a transition without observation. Every execution of an observe statement induces a transition $C \xrightarrow{\sigma} C'$, modeling a transition in which we observe the current store σ. Figure 3 depicts a selection of reduction rules. For a program $\mathbb{P}$ and store σ, there exists a unique *maximal* execution $\langle \mathbb{P}, \sigma \rangle \xrightarrow{\mu_1} \langle \mathbb{P}_1, \sigma_1 \rangle \xrightarrow{\mu_2} \langle \mathbb{P}_2, \sigma_2 \rangle \xrightarrow{\mu_3} \cdots$, where

$$\frac{}{\langle x = e, \sigma\rangle \xrightarrow{\epsilon} \langle \texttt{skip}, \sigma[x \mapsto [\![e]\!]_\sigma]\rangle} \qquad \frac{}{\langle \texttt{observe}, \sigma\rangle \xrightarrow{\sigma} \langle \texttt{skip}, \sigma\rangle}$$

$$\frac{[\![b]\!]_\sigma = \mathit{true}}{\langle \texttt{if}(b, \mathbb{P}, \mathbb{Q}), \sigma\rangle \xrightarrow{\epsilon} \langle \mathbb{P}, \sigma\rangle} \qquad \frac{}{\langle \texttt{skip} \fatsemi \mathbb{P}, \sigma\rangle \xrightarrow{\epsilon} \langle \mathbb{P}, \sigma\rangle} \qquad \frac{\langle \mathbb{P}, \sigma\rangle \xrightarrow{\mu} \langle \mathbb{P}', \sigma'\rangle}{\langle \mathbb{P} \fatsemi \mathbb{Q}, \sigma\rangle \xrightarrow{\mu} \langle \mathbb{P}' \fatsemi \mathbb{Q}, \sigma'\rangle}$$

Fig. 3. Selection of small-step reduction rules. We write $[\![e]\!]_\sigma \in \mathbb{Z}$ and $[\![b]\!]_\sigma \in \mathbb{B}$ for the value of expression e and b in store σ, respectively.

$\mu_1, \mu_2, \mu_3, \ldots \in \mathit{Stores} \cup \{\epsilon\}$. Note that this execution can be finite or infinite. We define $\mathit{obs}(\mathbb{P}, \sigma) := \mu_1\mu_2\mu_3 \cdots \in \mathit{Stores}^\star$ as the (finite or infinite) *observation sequence* along this execution (obtained by removing all ϵs). We write $\mathit{Traces}(\mathbb{P}) := \{\mathit{obs}(\mathbb{P}, \sigma) \mid \sigma \in \mathit{Stores}\} \subseteq \mathit{Stores}^\star$ for the set of all traces generated by $\mathbb{P}$. We say a program $\mathbb{P}$ is *terminating*, if all its executions are finite.

Syntax-Guided Synthesis. A Syntax-Guided Synthesis (SyGuS) problem is a triple $\Xi = (\{\tilde{f}_1, \ldots, \tilde{f}_n\}, \varrho, \{G_1, \ldots, G_n\})$, where $\tilde{f}_1, \ldots, \tilde{f}_n$ are function symbols, ϱ is an SMT constraint over the function symbols $\tilde{f}_1, \ldots, \tilde{f}_n$, and $G_1, \ldots, G_n$ are grammars [2]. A solution for Ξ is a vector of terms $\boldsymbol{e} = (e_1, \ldots, e_n)$ such that each e_i is generated by grammar G_i, and $\varrho[\tilde{f}_1/e_1, \ldots, \tilde{f}_n/e_n]$ holds (i.e., we replace each function symbol $\tilde{f}_i$ with expression e_i).

Example 1. Consider the SyGuS problem $\Xi = (\{\tilde{f}\}, \varrho, \{G\})$, where

$$\varrho := \forall x, y.\, \tilde{f}(x, y) \geq x \wedge \tilde{f}(x, y) \geq y \wedge (\tilde{f}(x, y) = x \vee \tilde{f}(x, y) = y)$$

$$G := \begin{cases} I \to x \mid y \mid 0 \mid 1 \mid I + I \mid I - I \mid \mathit{ite}(B, I, I) \\ B \to B \wedge B \mid B \vee B \mid \neg B \mid I = I \mid I \leq I \mid I \geq I. \end{cases}$$

This SyGuS problem constrains $\tilde{f}$ to be the function that returns the maximum of its arguments, and the grammar admits arbitrary piece-wise linear functions. A possible solution to Ξ would be $\tilde{f}(x, y) := \mathit{ite}(x \leq y, y, x)$. △

HyperLTL. As the basic specification language for hyperproperties, we use HyperLTL, an extension of LTL with explicit quantification over execution traces [13]. Let $\mathcal{V} = \{\pi_1, \ldots, \pi_n\}$ be a set of *trace variables*. For a trace variable $\pi_j \in \mathcal{V}$, we define $X_{\pi_j} := \{x_{\pi_j} \mid x \in X\}$ as a set of indexed program variables and $\boldsymbol{X} := X_{\pi_1} \cup \cdots \cup X_{\pi_n}$. We include predicates from an arbitrary first-order theory $\mathfrak{T}$ to reason about the infinite variable domains in programs (cf. [7]), and denote satisfaction in $\mathfrak{T}$ with $\models^{\mathfrak{T}}$. We write $\mathcal{F}_{\boldsymbol{X}}$ for the set of first-order predicates over variables $\boldsymbol{X}$. HyperLTL formulas are generated by the following grammar:

$$\varphi := \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi$$
$$\psi := \theta \mid \psi \wedge \psi \mid \psi \vee \psi \mid \bigcirc \psi \mid \psi \,\mathcal{W}\, \psi$$

where $\pi \in \mathcal{V}$, $\theta \in \mathcal{F}_{\boldsymbol{X}}$, and $\bigcirc$ and $\mathcal{W}$ are the *next* and *weak-until* operator, respectively. W.l.o.g., we assume that all variables in $\mathcal{V}$ occur in the prefix exactly once. We use the usual derived constants and connectives *true*, *false*, $\rightarrow$, and $\leftrightarrow$.

Remark 1. We only allow negation within the atomic predicates, effectively ensuring that the LTL-like body denotes a *safety property* [38]. The reason for this is simple: In our program semantics, we specifically allow for both infinite and *finite* executions. Our repair approach is thus applicable to reactive systems but also handles (classical) programs that terminate. By requiring that the body denotes a safety property, we can easily handle arbitrary combinations of finite and infinite executions. Note that our logic supports *arbitrary* quantifier alternations, so we can still express hyperliveness properties such as GNI. △

Let $\mathcal{T} \subseteq \mathit{Stores}^\star$ be a set of traces. For $t \in \mathcal{T}$ and $i < |t|$, we write $t(i)$ for the ith store in t. A trace assignment is a partial mapping $\Pi : \mathcal{V} \rightharpoonup \mathcal{T}$ from trace variables to traces. We write $\Pi_{(i)}$ for the assignment $\boldsymbol{X} \to \mathbb{Z}$ given by $\Pi_{(i)}(x_\pi) := \Pi(\pi)(i)(x)$, i.e., the value of x_π, is the value of x in the ith step on the trace bound to π. We define the semantics inductively as:

$$
\begin{array}{lll}
\Pi, i \models_{\mathcal{T}} \psi & \text{if} & \exists \pi \in \mathcal{V}.\, |\Pi(\pi)| \leq i \\
\Pi, i \models_{\mathcal{T}} \theta & \text{if} & \Pi_{(i)} \models^{\mathfrak{T}} \theta \\
\Pi, i \models_{\mathcal{T}} \psi_1 \wedge \psi_2 & \text{if} & \Pi, i \models_{\mathcal{T}} \psi_1 \text{ and } \Pi, i \models_{\mathcal{T}} \psi_2 \\
\Pi, i \models_{\mathcal{T}} \psi_1 \vee \psi_2 & \text{if} & \Pi, i \models_{\mathcal{T}} \psi_1 \text{ or } \Pi, i \models_{\mathcal{T}} \psi_2 \\
\Pi, i \models_{\mathcal{T}} \bigcirc \psi & \text{if} & \Pi, i+1 \models_{\mathcal{T}} \psi \\
\Pi, i \models_{\mathcal{T}} \psi_1 \,\mathcal{W}\, \psi_2 & \text{if} & (\exists j \geq i.\, \Pi, j \models_{\mathcal{T}} \psi_2 \text{ and } \forall i \leq k < j.\, \Pi, k \models_{\mathcal{T}} \psi_1) \text{ or} \\
 & & \quad (\forall j \geq i.\, \Pi, j \models_{\mathcal{T}} \psi_1) \\
\Pi, i \models_{\mathcal{T}} \exists \pi.\, \varphi & \text{if} & \exists t \in \mathcal{T}.\, \Pi[\pi \mapsto t], i \models_{\mathcal{T}} \varphi \\
\Pi, i \models_{\mathcal{T}} \forall \pi.\, \varphi & \text{if} & \forall t \in \mathcal{T}.\, \Pi[\pi \mapsto t], i \models_{\mathcal{T}} \varphi
\end{array}
$$

As we deal with safety formulas (cf. Remark 1), we let Π, i satisfy any formula ψ as soon as we have moved past the length of the shortest trace in Π (i.e., $\exists \pi \in \mathcal{V}.\, |\Pi(\pi)| \leq i$). A program $\mathbb{P}$ satisfies φ, written $\mathbb{P} \models \varphi$, if $\emptyset, 0 \models_{\mathit{Traces}(\mathbb{P})} \varphi$, where $\emptyset$ denotes the trace assignment with an empty domain.

NSA. A *nondeterministic safety automaton* (NSA) over alphabet Σ is a tuple $\mathcal{A} = (Q, Q_0, \delta)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, and $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation. A *run* of $\mathcal{A}$ on a word $u \in \Sigma^\star$ is a sequence $q_0 q_1 \cdots \in Q^\star$ such that $q_0 \in Q_0$ and for every $i < |u|$, $(q_i, u(i), q_{i+1}) \in \delta$. We write $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\star$ for the set of words on which $\mathcal{A}$ has *some* run.

3 Program Repair by Symbolic Execution

In our repair setting, we are given a pair $(\mathbb{P}, \varphi)$ such that $\mathbb{P} \not\models \varphi$, and try to construct a repaired program $\mathbb{Q}$ with $\mathbb{Q} \models \varphi$. In particular, we repair w.r.t. a *formal specification* instead of a set of input-output examples. The reason for this lies within the nature of the properties we want to repair against: When repairing against trace properties (i.e., functional specifications), it is often intuitive to write input-output examples that test a program's functional behavior.

$$\frac{}{\langle x = e, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \texttt{skip}, \nu[x \mapsto [\![e]\!]_\nu], \alpha, \beta\rangle} \qquad \frac{}{\langle \texttt{observe}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \texttt{skip}, \nu, \alpha, \beta \cdot \nu\rangle}$$

$$\frac{}{\langle \texttt{skip} \fatsemi \mathbb{P}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}, \nu, \alpha, \beta\rangle} \qquad \frac{}{\langle \texttt{if}(b, \mathbb{P}, \mathbb{Q}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}, \nu, \alpha \wedge [\![b]\!]_\nu, \beta\rangle}$$

$$\frac{\langle \mathbb{P}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}', \nu', \alpha', \beta'\rangle}{\langle \mathbb{P} \fatsemi \mathbb{Q}, \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P}' \fatsemi \mathbb{Q}, \nu', \alpha', \beta'\rangle} \qquad \frac{}{\langle \texttt{if}(b, \mathbb{P}, \mathbb{Q}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{Q}, \nu, \alpha \wedge \neg[\![b]\!]_\nu, \beta\rangle}$$

$$\frac{}{\langle \texttt{while}(b, \mathbb{P}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \mathbb{P} \fatsemi \texttt{while}(b, \mathbb{P}), \nu, \alpha \wedge [\![b]\!]_\nu, \beta\rangle}$$

$$\frac{}{\langle \texttt{while}(b, \mathbb{P}), \nu, \alpha, \beta\rangle \xrightarrow{sym} \langle \texttt{skip}, \nu, \alpha \wedge \neg[\![b]\!]_\nu, \beta\rangle}$$

Fig. 4. Small-step reduction rules for symbolic execution.

In contrast, hyperproperties do not directly reason about concrete functional behavior but rather about the abstract relation between multiple computations. For example, information-flow properties such as non-interference can be applied to arbitrary programs; independent of the program's functional behavior. Perhaps counter-intuitively, in our hyper-setting, formal specifications are thus often easier to construct than input-output examples.

3.1 Symbolic Execution

The first step in our repair pipeline is the computation of a mathematical summary of (parts of) the program's executions using symbolic execution (SE) [37]. In SE, we execute the program using symbolic placeholders instead of concrete values for variables, and explore all symbolic paths of a program (recording conditions that a concrete store needs to satisfy to take any given branch). A *symbolic store* is a function $\nu : X \to \mathcal{E}_\mathbb{Z}$ that maps each variable to an expression, and we write $\mathit{SymStores} := \{\nu \mid \nu : X \to \mathcal{E}_\mathbb{Z}\}$ for the set of all symbolic stores. A *symbolic configuration* is then a tuple $\langle \mathbb{P}, \nu, \alpha, \beta\rangle$, where $\mathbb{P}$ is a program, $\nu \in \mathit{SymStores}$ is a symbolic store, $\alpha \in \mathcal{F}_X$ is a first-order formula over X that records which conditions the current path should satisfy (called the *path condition*), and $\beta \in \mathit{SymStores}^*$ is a sequence of symbolic stores recording the observations. For $e \in \mathcal{E}_\mathbb{Z}$ and $\nu \in \mathit{SymStores}$, we write $[\![e]\!]_\nu$ for the expression obtained by replacing each variable x in e with $\nu(x)$. For example, if $\nu = [x \mapsto x - 1, y \mapsto z * y]$, we have $[\![x + y]\!]_\nu = (x - 1) + (z * y)$. We give the symbolic execution relation $\xrightarrow{sym}$ in Fig. 4. We start the symbolic execution in symbolic store $\nu_0 := \big[x \mapsto x\big]_{x \in X}$ that maps each variable to itself, path condition $\alpha_0 := \mathit{true}$, and an empty observation sequence $\beta_0 := \epsilon$. Given a program $\mathbb{P}$, a symbolic execution is a *finite* sequence of symbolic configurations

$$\rho = \langle \mathbb{P}, \nu_0, \alpha_0, \beta_0\rangle \xrightarrow{sym} \langle \mathbb{P}_1, \nu_1, \alpha_1, \beta_1\rangle \xrightarrow{sym} \cdots \xrightarrow{sym} \langle \mathbb{P}_m, \nu_m, \alpha_m, \beta_m\rangle \qquad (1)$$

We say execution ρ is *maximal* if $\mathbb{P}_m = \texttt{skip}$, i.e., we cannot perform any more execution steps. Given a symbolic execution ρ, we are interested in the

acc^{ψ}_{Δ}

For $\Delta : \{\pi_1, \dots, \pi_n\} \to \mathit{SymStores}^*$, define $acc^{\psi}_{\Delta} := \bigvee_{q \in Q_{0,\psi}} acc^{q,0}_{\Delta}$, where $acc^{q,i}_{\Delta} := \mathit{true}$ iff $\exists \pi. |\Delta(\pi)| \leq i$, and otherwise

$$acc^{q,i}_{\Delta} := \bigvee_{(q,\iota,q') \in \delta_\psi} \Big(acc^{q',i+1}_{\Delta} \wedge \bigwedge_{\theta \in F \mid \theta \in \iota} \theta\Big[x_{\pi_j}/(\Delta(\pi_j)(i)(x)[y/y_{\pi_j}])\Big] \wedge \bigwedge_{\theta \in F \mid \theta \notin \iota} \neg\theta\Big[x_{\pi_j}/(\Delta(\pi_j)(i)(x)[y/y_{\pi_j}])\Big] \Big)$$

Fig. 5. Encoding for acceptance of ψ.

path condition α_m (to ensure that we follow an actual program path), and the observation sequence β_m (to evaluate the HyperLTL property). We define a *symbolic path* as a pair in $\mathcal{F}_X \times \mathit{SymStores}^*$, recording the path condition and symbolic observation sequence. Each execution ρ of the form in (1), yields a symbolic path (α_m, β_m). We call the symbolic path (α_m, β_m) *maximal* if ρ is maximal, and *satisfiable* if α_m is satisfiable (i.e., some actual program execution can take a path summarized by ρ). We write $\mathit{SymPaths}(\mathbb{P}) \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ for the set of all satisfiable symbolic paths of $\mathbb{P}$ and $\mathit{SymPaths}_{max}(\mathbb{P}) \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ for the set of all satisfiable maximal symbolic paths.

Remark 2. An interesting class of programs are those that are terminating and where $\mathit{SymPaths}_{max}(\mathbb{P})$ is *finite*. This is either the case when the program is loop-free or has some upper bound on the number of loop executions (and thus control paths). Crucially, if $\mathit{SymPaths}_{max}(\mathbb{P})$ is finite, it provides a precise and complete mathematical summary of the program's executions. △

3.2 Symbolic Paths and Safety Automata

We can use symbolic paths to approximate the HyperLTL semantics by explicitly considering path combinations. Let $\varphi = \mathbb{Q}_1\pi_1 \dots \mathbb{Q}_n\pi_n.\, \psi$ be a fixed HyperLTL formula, where $\mathbb{Q}_1, \dots, \mathbb{Q}_n \in \{\forall, \exists\}$ are quantifiers, and ψ is the LTL body of φ. Further, let $F \subseteq \mathcal{F}_{\boldsymbol{X}}$ be the *finite* set of predicates used in ψ. Due to our syntactic safety restriction on LTL formulas, we can construct an NSA $\mathcal{A}_\psi = (Q_\psi, Q_{0,\psi}, \delta_\psi)$ over alphabet 2^F accepting exactly the words that satisfy ψ [38].

Assume $\Delta : \{\pi_1, \dots, \pi_n\} \to \mathit{SymStores}^*$ is a function that assigns each path variable $\pi_1, \dots, \pi_n$ a symbolic observation sequence. We design a formula acc^{ψ}_{Δ}, which encodes that the symbolic observation sequences in Δ have an accepting prefix in $\mathcal{A}_\psi$, given in Fig. 5. The intermediate formula $acc^{q,i}_{\Delta}$ encodes that the observations in Δ have some run from state q in the ith step. For all steps i, longer than the shortest trace in Δ, we accept (i.e., $acc^{q,i}_{\Delta} := \mathit{true}$, similar to our HyperLTL semantics). Otherwise, we require some transition $(q, \iota, q') \in \delta_\psi$ such

$enc^{\varphi}_{\mathcal{P}}$

For $\mathcal{P} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$, define $enc^{\varphi}_{\mathcal{P}} := enc^{\varphi}_{\mathcal{P},\emptyset}$, where

$$
\begin{aligned}
enc^{\psi}_{\mathcal{P},\Delta} &:= acc^{\psi}_{\Delta} \\
enc^{\exists \pi_j . \varphi'}_{\mathcal{P},\Delta} &:= \exists_{x_{\pi_j} \in X_{\pi_j}} x_{\pi_j} . \bigvee_{(\alpha,\beta)\in\mathcal{P}} \Big(\alpha[x/x_{\pi_j}] \wedge enc^{\varphi'}_{\mathcal{P},\Delta[\pi_j \mapsto \beta]} \Big) \\
enc^{\forall \pi_j . \varphi'}_{\mathcal{P},\Delta} &:= \forall_{x_{\pi_j} \in X_{\pi_j}} x_{\pi_j} . \bigwedge_{(\alpha,\beta)\in\mathcal{P}} \Big(\alpha[x/x_{\pi_j}] \rightarrow enc^{\varphi'}_{\mathcal{P},\Delta[\pi_j \mapsto \beta]} \Big)
\end{aligned}
$$

Fig. 6. Encoding of the HyperLTL semantics on symbolic paths $\mathcal{P}$.

that $acc^{q',i+1}_{\Delta}$ holds, and the label $\iota \in 2^F$ holds in step i. To encode the latter, we use the symbolic observation sequences in Δ: For every predicate $\theta \in F$, we require that $\theta \in \iota$ iff $\theta\big[x_{\pi_j}/\big(\Delta(\pi_j)(i)(x)[y/y_{\pi_j}]\big)\big]$. That is, we replace variable x_{π_j} with the expression $\Delta(\pi_j)(i)(x)[y/y_{\pi_j}]$, i.e., we look up the expression bound to variable x in the ith step on $\Delta(\pi_j)$, and – within this expression – index all variables with π_j (i.e., replace each variable $y \in X$ with $y_{\pi_j} \in X_{\pi_j}$).

3.3 Encoding for HyperLTL

Let $\mathcal{P} \subseteq \mathcal{F}_X \times \mathit{SymStores}^*$ be a finite set of symbolic paths and consider the formula $enc^{\varphi}_{\mathcal{P}}$ in Fig. 6. Intuitively, the formula encodes the satisfaction of φ on the symbolic paths in $\mathcal{P}$. For this, we maintain a *partial* mapping $\Delta : \{\pi_1, \ldots, \pi_n\} \rightharpoonup \mathit{SymStores}^*$, and for each subformula φ' we define an intermediate formula $enc^{\varphi'}_{\mathcal{P},\Delta}$. If we reach the LTL body ψ, we define $enc^{\psi}_{\mathcal{P},\Delta} := acc^{\psi}_{\Delta}$, stating that the symbolic observation sequences in Δ satisfy ψ (cf. Fig. 5). Each trace quantifier is then resolved on the symbolic paths in $\mathcal{P}$. Concretely, for a subformula $\exists \pi_j . \varphi'$, we existentially quantify over variables X_{π_j} and *disjunctively* pick a symbolic path $(\alpha, \beta) \in \mathcal{P}$. We require that path condition α holds (after replacing each variable x with x_{π_j}), and that the remaining formula φ' is satisfied if we bind observation sequence β to π_j (i.e., $enc^{\varphi'}_{\mathcal{P},\Delta[\pi_j \mapsto \beta]}$).

Proposition 1. *If $\mathbb{Q}$ is a terminating program and $\mathit{SymPaths}_{max}(\mathbb{Q})$ is finite, then $\mathbb{Q} \models \varphi$ if and only if $enc^{\varphi}_{\mathit{SymPaths}_{max}(\mathbb{Q})}$.*

The above proposition essentially states that we can use SE to verify a program (with finitely-many symbolic paths) against HyperLTL formulas with *arbitrary* quantifier alternations. This is in sharp contrast to existing SE-based approaches, which only apply to k-safety properties (i.e., $\forall^*$ HyperLTL formulas) [17,18,22,51,52]. To the best of our knowledge, ours is the first approach that can check properties containing arbitrary alternations on fragments of infinite-state software programs. Previous methods either focus on finite-state systems [6,8,16,24,31,32] or only consider restricted quantifier structures [7,23,34,49,53].

Alternation-Free Formulas. In many situations, we cannot explore *all* symbolic paths of a program $\mathbb{Q}$ (i.e., $SymPaths_{max}(\mathbb{Q})$ is infinite). However, even by just exploring a subset of paths, our encoding still allows us to draw conclusions about the full program as long as the formula is *alternation-free*.

Proposition 2. *Assume φ is a $\forall^*$ formula and $\mathcal{P} \subseteq SymPaths_{max}(\mathbb{Q})$ is a finite set of maximal symbolic paths. If $enc^{\varphi}_{\mathcal{P}}$, then $\mathbb{Q} \models \varphi$.*

Proposition 3. *Assume φ is a $\forall^*$ formula and $\mathcal{P} \subseteq SymPaths(\mathbb{Q})$ is a finite set of (not necessarily maximal) symbolic paths. If $\neg enc^{\varphi}_{\mathcal{P}}$, then $\mathbb{Q} \not\models \varphi$.*

In particular, we can use Proposition 3 for our repair approach for $\forall^*$ properties (which captures many properties of interest, such as non-interference, cf. φ_{edas}). If we symbolically execute a program to some fixed depth (and thus capture a subset of the symbolic paths), any possible repair must satisfy the bounded property described in $enc^{\varphi}_{\mathcal{P}}$ (cf. Sect. 3.4). Note that this does not ensure that the repair patch that fulfills $enc^{\varphi}_{\mathcal{P}}$ is correct on the entire program; $enc^{\varphi}_{\mathcal{P}}$ merely describes a *necessary* condition any possible repair needs to satisfy. In our experiments (cf. Sect. 6), we (empirically) found that the repair for the bounded version also serves as a repair for the full program in many instances.

3.4 Program Repair Using SyGuS

Using SE and our encoding, we can now outline our basic SyGuS-based repair approach. Assume $\mathbb{P} \not\models \varphi$ is the program that should be repaired. As in other semantic-analysis-based repair frameworks [44,46], we begin our repair by predicting fault locations [54] within the program, i.e., locations that are likely to be responsible for the violation of φ. In our later experiments, we assume that these locations are provided by the user. After we have identified a set of n repair locations, we instrument $\mathbb{P}$ by replacing the expressions in all repair locations with fresh *function symbols*. That is, if we want to repair statement $x = e$, $\texttt{if}(b, \mathbb{P}_1, \mathbb{P}_2)$, or $\texttt{while}(b, \mathbb{P})$, we replace the statement with $x = \tilde{f}(x_1, \dots, x_m)$, $\texttt{if}(\tilde{f}(x_1, \dots, x_m), \mathbb{P}_1, \mathbb{P}_2)$, or $\texttt{while}(\tilde{f}(x_1, \dots, x_m), \mathbb{P})$, respectively, for some fresh function symbol $\tilde{f}$ and program variables $x_1, \dots, x_m \in X$ (inferred using a lightweight dependency analysis). Let $\mathbb{Q}$ be the resulting program, which contains function symbols, $\tilde{f}_1, \dots, \tilde{f}_n$. We symbolically execute $\mathbb{Q}$, leading to a set of symbolic paths $\mathcal{P}$ containing $\tilde{f}_1, \dots, \tilde{f}_n$, and define the SyGuS problem $\Xi_{\mathcal{P}} := (\{\tilde{f}_1, \dots, \tilde{f}_n\}, enc^{\varphi}_{\mathcal{P}}, \{G_1, \dots, G_n\})$. Here, we fix a grammar G_i for each function symbol $\tilde{f}_i$, based on the type and context of each repair location. Note that $enc^{\varphi}_{\mathcal{P}}$ now constitutes an SMT constraint over $\tilde{f}_1, \dots, \tilde{f}_n$. Any solution for $\Xi_{\mathcal{P}}$ thus defines concrete expressions for $\tilde{f}_1, \dots, \tilde{f}_n$ such that the symbolic paths in $\mathcal{P}$ satisfy φ. Concretely, let $\boldsymbol{e} = (e_1, \dots, e_n)$ be a solution to $\Xi_{\mathcal{P}}$. Define $\mathbb{Q}[\boldsymbol{e}] := \mathbb{Q}[\tilde{f}_1/e_1, \dots, \tilde{f}_n/e_n]$, i.e., we replace each function symbol $\tilde{f}_i$ by expression e_i. As $\boldsymbol{e}$ is a solution to $\Xi_{\mathcal{P}}$, we directly obtain that $\mathbb{Q}[\boldsymbol{e}]$ satisfies φ; at least restricted to the executions captured by the symbolic paths in $\mathcal{P}$. Afterward, we can *verify* that $\mathbb{Q}[\boldsymbol{e}]$ indeed satisfies φ (even on paths not explored in $\mathcal{P}$), using existing hyperproperty verification techniques [7,23,34,49,53].

Example 2. Consider the EDAS program $\mathbb{P}$ in Fig. 1, and let $\mathbb{Q}$ be the modified program where the assignment in line 4 is replaced with a fresh function symbol $\tilde{f}$. Define $X := \{\texttt{phase}, \texttt{title}, \texttt{session}, \texttt{decision}\}$. If we perform SE on $\mathbb{Q}$, we get two symbolic paths $\mathcal{P}_{\mathbb{Q}} = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2)\}$, where $\alpha_1 = (\tilde{f}(X) = \texttt{"Accept"})$, $\alpha_2 = (\tilde{f}(X) \neq \texttt{"Accept"})$, $\beta_1 = [[\ldots], [\texttt{print} \mapsto \texttt{title + session}, \texttt{decision} \mapsto \tilde{f}(X), \ldots]]$, and $\beta_2 = [[\ldots], [\texttt{print} \mapsto \texttt{title}, \texttt{decision} \mapsto \tilde{f}(X), \ldots]]$. For illustration, we consider the simple trace property $\varphi_{trace} = \forall\pi.\, \bigcirc(\texttt{print}_\pi = \texttt{title}_\pi)$. If we construct $enc_{\mathcal{P}_{\mathbb{Q}}}^{\varphi_{trace}}$, we get

$$\bigwedge_{x_\pi \in X_\pi} x_\pi.\, \Big(\tilde{f}(X_\pi) = \texttt{"Accept"} \rightarrow \texttt{title}_\pi \texttt{ + } \texttt{session}_\pi = \texttt{title}_\pi\Big) \wedge \Big(\tilde{f}(X_\pi) \neq \texttt{"Accept"} \rightarrow \texttt{title}_\pi = \texttt{title}_\pi\Big),$$

allowing the simple SyGuS solution $\tilde{f}(X_\pi) := \texttt{"Reject"}$. △

4 Transparent Repair

As argued in Sect. 1, searching for *any* repair (as in Sect. 3) often returns a patch that severely changes the functional behavior of the program. In this paper, we study a principled constraint-based approach on how to guide the search towards a useful repair without requiring extensive additional specifications. Our method is based on the simple idea that the repair should be somewhat *close* to the original program. Crucially, we define "closeness" via *rigorous* systems of (SyGuS) constraints, guiding our constraint-based repair towards minimal patches, with *guaranteed* quality. In this section, we introduce the concept of a (fully) transparent repair. In Sect. 5, we adapt this idea and present a more practical adaption in the form of iterative repair.

4.1 Transparency

Our transparent repair approach is motivated by ideas from the *enforcement* literature [45]. In enforcement, we do not repair the program (i.e., we do not manipulate its source code) but rather let an enforcer run alongside the program and intervene on unsafe behavior (by, e.g., overwriting the output). The obvious enforcement strategy would thus always intervene, effectively overwriting all program behaviors with some dummy (but safe) behavior. To avoid such trivial enforcement, researchers have developed the notion of *transparency* (also called *precision* [45]). Transparency states that the enforcer should not intervene unless an intervention is *absolutely necessary* to satisfy the safety specification, i.e., a safe prefix of the program execution should never trigger the enforcer.

Transparent Repair. The original transparency definition is specific to program enforcement and refers to the *time step* in which the enforcer intervenes. We propose an adoption to the repair setting based on the idea of preserving as

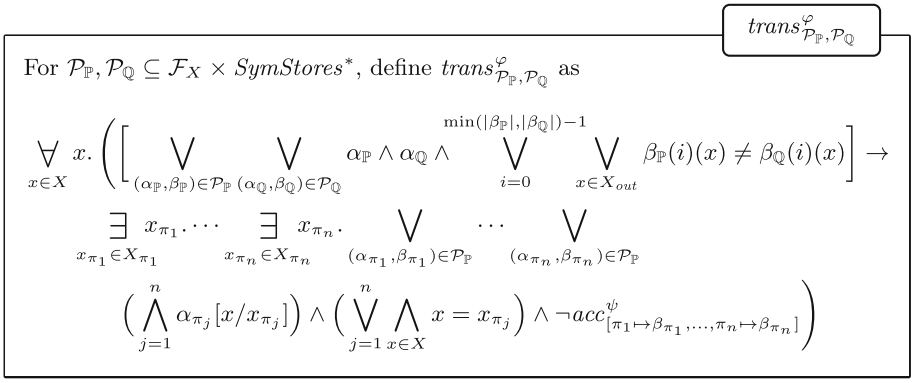

Fig. 7. Encoding for (fully) transparent repair.

much input-output behavior of the original program as possible. Let $X_{out} \subseteq X$ be a set of program variables defining the output. For two stores $\sigma, \sigma' \in Stores$, we write $\sigma \neq_{X_{out}} \sigma'$ if $\sigma(x) \neq \sigma'(x)$ for some $x \in X_{out}$, and extend $\neq_{X_{out}}$ position-wise to sequences of stores.

Definition 1 (Fully Transparent Repair). *Assume $\varphi = \forall\pi_1 \ldots \forall\pi_n. \psi$ is a $\forall^*$ HyperLTL formula and $\mathbb{P}, \mathbb{Q}$ are programs. We say $\mathbb{Q}$ is a fully transparent repair of $(\mathbb{P}, \varphi)$, if (1) $\mathbb{Q} \models \varphi$, and (2) for every store $\sigma \in Stores$ where $obs(\mathbb{P}, \sigma) \neq_{X_{out}} obs(\mathbb{Q}, \sigma)$, there exist stores $\sigma_1, \ldots, \sigma_n \in Stores$ such that $\big[\pi_j \mapsto obs(\mathbb{P}, \sigma_j)\big]_{j=1}^{n}, 0 \not\models \psi$, and $\sigma = \sigma_j$ for some $1 \leq j \leq n$.*

Our definition reasons about inputs σ on which the output behavior of $\mathbb{Q}$ differs from the original program $\mathbb{P}$. Any such input σ must take part in a violation of φ on the original program $\mathbb{P}$. Phrased differently, the repair may only change $\mathbb{P}$'s behavior on executions that take part in a combination of n traces that violate φ. Note that, similar to enforcement approaches [15,45], our transparency definition only applies to $\forall^*$ formulas. As soon as the property includes existential quantification, we can no longer formalize when some execution is "part of a violation of φ". We will extend the central idea underpinning transparency to arbitrary HyperLTL formulas in Sect. 5.

4.2 Encoding for Transparent Repair

Given two finite sets of symbolic paths $\mathcal{P}_\mathbb{P}, \mathcal{P}_\mathbb{Q} \subseteq \mathcal{F}_X \times SymStores^*$, we define formula $trans^{\varphi}_{\mathcal{P}_\mathbb{P},\mathcal{P}_\mathbb{Q}}$ in Fig. 7. The premise states that X defines some input on which $\mathbb{P}$ and $\mathbb{Q}$ differ in their output. That is, for some symbolic paths $(\alpha_\mathbb{P}, \beta_\mathbb{P}) \in \mathcal{P}_\mathbb{P}$ and $(\alpha_\mathbb{Q}, \beta_\mathbb{Q}) \in \mathcal{P}_\mathbb{Q}$, the path conditions $\alpha_\mathbb{P}$ and $\alpha_\mathbb{Q}$ hold, but the symbolic observation sequences yield some different values for some $x \in X_{out}$. In this case, we require that there exist n symbolic paths $(\alpha_{\pi_1}, \beta_{\pi_1}), \ldots, (\alpha_{\pi_n}, \beta_{\pi_n}) \in \mathcal{P}_\mathbb{P}$ and concrete inputs $X_{\pi_1}, \ldots, X_{\pi_n}$, such that (1) the path conditions $\alpha_{\pi_1}, \ldots, \alpha_{\pi_n}$

hold; (2) the assignment to some X_{π_j} equals X; and (3) the symbolic observation sequences $\beta_{\pi_1}, \ldots, \beta_{\pi_n}$ violate ψ (cf. Fig. 5).

Proposition 4. *If* $\mathbb{P}$, $\mathbb{Q}$ *are terminating and* $SymPaths_{max}(\mathbb{P})$, $SymPaths_{max}(\mathbb{Q})$ *are finite, then* $\mathbb{Q}$ *is a fully transparent repair of* $(\mathbb{P}, \varphi)$ *if and only if*

$$enc^{\varphi}_{SymPaths_{max}(\mathbb{Q})} \wedge trans^{\varphi}_{SymPaths_{max}(\mathbb{P}),SymPaths_{max}(\mathbb{Q})}.$$

Example 3. We illustrate transparent repairs using Example 2. If we set $X_{out} := \{\texttt{decision}\}$, and compute $trans^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$, we get

$$\bigvee_{x \in X} x.\ (\texttt{decision} \neq \tilde{f}(X)) \rightarrow \Big((\texttt{decision} = \texttt{"Accept"} \wedge \texttt{title + session} \neq \texttt{title}) \vee (\texttt{decision} \neq \texttt{"Accept"} \wedge \texttt{title} \neq \texttt{title})\Big).$$

For simplicity, we directly resolved the existentially quantified variables X_π with X and summarized all path constraints in the premise. The naïve solution $\tilde{f}(X) := \texttt{"Reject"}$ from Example 2 no longer satisfies $trans^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$. Instead, a possible SyGuS solution for $enc^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{Q}}} \wedge trans^{\varphi_{trace}}_{\mathcal{P}_{\mathbb{P}},\mathcal{P}_{\mathbb{Q}}}$ is

$$\tilde{f}(X) := ite(\texttt{decision} = \texttt{"Accept"} \wedge \texttt{session} \neq \texttt{""}, \texttt{"Reject"}, \texttt{decision}).$$

This solution only changes the decision if the `decision` is `"Accept"` *and* the `session` does not equal the empty string, i.e., it changes the program's `decision` on exactly those traces that violate $\varphi_{trace} = \forall\pi.\bigcirc(\texttt{print}_\pi = \texttt{title}_\pi)$. △

5 Iterative Repair

Our full transparency definition only applies to $\forall^*$ properties, and, even on $\forall^*$ formulas, might yield undesirable results: In some instances, Definition 1 limits which traces may be changed by a repair, potentially resulting in the absence of any repair. In other instances (including the EDAS example), many paths (in the EDAS example, *all* paths) take part in *some* violation of the hyperproperty, so full transparency does not impose any additional constraints. In the EDAS example, this would again allow the naïve repair `decision = "Reject"`. To alleviate this, we introduce an *iterative repair* approach that follows the same philosophical principle as (full) transparency (i.e., search for repairs that are close to the original program), but allows for the *iterative* discovery of better and better repair patches.

Definition 2. *Assume* φ *is a HyperLTL formula and* $\mathbb{P}$, $\mathbb{Q}$, *and* $\mathbb{S}$ *are programs. We say repair* $\mathbb{Q}$ *is a better repair than* $\mathbb{S}$ *w.r.t.* $(\mathbb{P}, \varphi)$ *if (1)* $\mathbb{Q} \models \varphi$, *(2) for every* $\sigma \in Stores$, *where* $obs(\mathbb{P}, \sigma) \neq_{X_{out}} obs(\mathbb{Q}, \sigma)$, *we have* $obs(\mathbb{P}, \sigma) \neq_{X_{out}} obs(\mathbb{S}, \sigma)$, *and (3) for some* $\sigma \in Stores$, *we have* $obs(\mathbb{P}, \sigma) \neq_{X_{out}} obs(\mathbb{S}, \sigma)$ *but* $obs(\mathbb{P}, \sigma) =_{X_{out}} obs(\mathbb{Q}, \sigma)$.

Intuitively, $\mathbb{Q}$ is better than $\mathbb{S}$ if it preserves at least all those behaviors of $\mathbb{P}$ already preserved by $\mathbb{S}$, i.e., $\mathbb{Q}$ is only allowed to deviate from $\mathbb{P}$ on inputs where $\mathbb{S}$ already deviates. Moreover, it must be strictly better than $\mathbb{S}$, i.e., preserve at least one additional behavior.

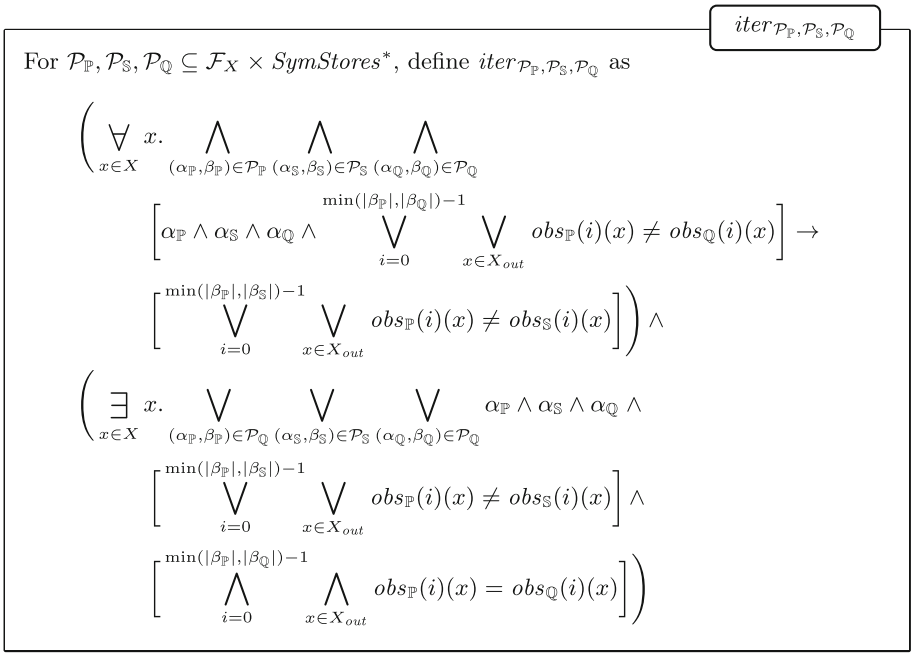

Fig. 8. Encoding for iterative repair.

5.1 Encoding for Iterative Repair

As before, we show that we can encode Definition 2 via a repair constraint. Let $\mathcal{P}_\mathbb{P}$, $\mathcal{P}_\mathbb{S}, \mathcal{P}_\mathbb{Q} \subseteq \mathcal{F}_X \times SymStores^*$ be finite sets of symbolic paths, and define $iter_{\mathcal{P}_\mathbb{P},\mathcal{P}_\mathbb{S},\mathcal{P}_\mathbb{Q}}$ as in Fig. 8.

Proposition 5. *If* $\mathbb{P}$, $\mathbb{Q}$, *and* $\mathbb{S}$ *are terminating programs and* $SymPaths_{max}(\mathbb{P})$, $SymPaths_{max}(\mathbb{S})$, *and* $SymPaths_{max}(\mathbb{Q})$ *are finite, then* $\mathbb{Q}$ *is a better repair than* $\mathbb{S}$, *w.r.t.,* $(\mathbb{P}, \varphi)$ *if and only if*

$$enc^{\varphi}_{SymPaths_{max}(\mathbb{Q})} \wedge iter_{SymPaths_{max}(\mathbb{P}),SymPaths_{max}(\mathbb{S}),SymPaths_{max}(\mathbb{Q})}.$$

5.2 Iterative Repair Loop

We sketch our iterative repair algorithm in Algorithm 1. In line 2, we infer the locations that we want to repair from user annotations. We leave the exploration of automated fault localization techniques specific for hyperproperties as future work, and, in our experiments, assume that the user marks potential repair locations. In line 3, we instrument $\mathbb{P}$ by replacing all repair locations in *locs* with fresh function symbols. At the same time, we record the original expression at all those locations as a vector $\boldsymbol{e}_\mathbb{P}$. Subsequently, we perform symbolic execution on

the skeleton program $\mathbb{Q}$ (i.e., the program that contains fresh function symbols), yielding a set of symbolic paths $\mathcal{P}$ containing function symbols (line 4). Initially, we now search for *some* repair of φ by using the SyGuS constraint $enc_{\mathcal{P}}^{\varphi}$, giving us an initial repair patch in the form of some expression vector $\boldsymbol{e}$ (line 5). Afterward, we try to iteratively improve upon the repair solution $\boldsymbol{e}$ found previously. For this, we consider the SyGuS constraint $enc_{\mathcal{P}}^{\varphi} \wedge iter_{\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}],\mathcal{P}[\boldsymbol{e}],\mathcal{P}}$ where we replaced each function symbol in $\mathcal{P}$ with $\boldsymbol{e}_{\mathbb{P}}$ to get the symbolic paths of the original program (denoted $\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}]$), and with $\boldsymbol{e}$ to get the symbolic paths of the previous repair (denoted $\mathcal{P}[\boldsymbol{e}]$) (line 7). If this SyGuS constraint admits a solution $\boldsymbol{e}'$, we set $\boldsymbol{e}$ to $\boldsymbol{e}'$ and repeat with a further improvement iteration (line 11). If the SyGuS constraint is unsatisfiable (or, e.g., a timeout is reached, or the number of iterations is bounded) (written $\boldsymbol{e}' = \bot$), we return the last solution we found, i.e., the program $\mathbb{Q}[\boldsymbol{e}]$ (line 9). By using a single set of symbolic paths $\mathcal{P}$ of the skeleton program $\mathbb{Q}$, we can optimize our query construction. For example, in $iter_{\mathcal{P}[\boldsymbol{e}_{\mathbb{P}}],\mathcal{P}[\boldsymbol{e}],\mathcal{P}}$, we consider all 3 tuples of symbolic paths leading to a potentially large SyGuS query. As we use a common set of paths $\mathcal{P}$ we can prune many path combinations. For example, on fragments preceding a repair location, we never have to combine contradicting branch conditions.

6 Implementation and Evaluation

We have implemented our repair techniques from Sects. 3 to 5 in a proof-of-concept prototype called `HyRep`, which takes as input a HyperLTL formula and a program in a minimalist `C`-like language featuring Booleans, integers, and strings. We use `spot` [20] to translate LTL formulas to NSAs. `HyRep` can use any solver supporting the SyGuS input format [2]; we use `cvc5` (version 1.0.8) [4] as the default solver in all experiments. In `HyRep`, the user can determine what SyGuS grammar to use, guiding the solver towards a particular (potentially domain-specific) solution. By default, `HyRep` repairs integer and Boolean expressions using piece-wise linear functions (similar to Example 1), and string-valued expressions by a grammar allowing selected string constants and concatenation of string variables. All results in this paper were obtained using a Docker container of `HyRep` running on an Apple M1 Pro CPU and 32 GB of memory.

Algorithm 1. Iterative repair algorithm

```
1  def iterativeRepair(ℙ,φ):
2    locs := faultLocalization(ℙ,φ)
3    ℚ,e_ℙ := instrument(ℙ,locs)
4    𝒫 := symbolicExecution(ℚ)
5    e := SyGuS(enc_𝒫^φ)
6    repeat:
7      e' := SyGuS(enc_𝒫^φ ∧ iter_{𝒫[e_ℙ],𝒫[e],𝒫})
8      if (e' = ⊥) then
9        return ℚ[e]
10     else
11       e := e'
```

Scalability Limitations. As we repair for hyperproperties, we necessarily need to reason about the combination of paths, requiring us to analyze multiple paths simultaneously. Unsurprisingly, this limits the scalability of our repair. Consequently, we cannot tackle programs with hundreds of LoC, where existing

```
login(int password, bool attack) {
  if (password == 366) {
    if (attack == true) {
      request = 2 // hidden request (unsafe)
    } else {
      request = 1 // user request (safe)
    }
  } else {
    request = 0 // empty request
  }
  request = request
  observe
}
```

```
request = 0
```

(a)

```
if (password == 366) {
  request = 1
} else {
  request = request
}
```

(b)

Fig. 9. A CSRF attack and repair candidates by HyRep.

(*functional*) APR approaches collect a small summary that only depends on the number of input-output examples (see, e.g., *angelic forests* [44]). However, our experiments with HyRep attest that – while we can only handle small programs – our approach can find *complex* repair solutions that go beyond previous repair approaches for hyperproperties (cf. Sect. 7).

6.1 Iterative Repair for Hyperproperties

We first focus on HyRep's ability to find, often non-trivial, repair solutions using its iterative repair approach. Table 1 depicts an overview of the 5 benchmark families we consider (explained in the following). For some of the benchmarks, we also consider small variants by adding additional complexity to the program.

Table 1. We depict the number of improvement iterations, the number repair locations, and the repair time (in seconds).

Instance	#Iter	#Locations	t
EDAS	2	1	2.5
CSRF	2	1	17.9
LOG	1	1	0.9
LOG′	1	1	1.0
LOG″	1	1	7.4
ATM	3	2	4.2
REVIEWS	3	2	18.5
REVIEWS′	3	2	151.6

EDAS. As already discussed in Sect. 1, HyRep is able to repair (a simplified integer-based version of) the EDAS example in Fig. 1 and derive the repairs in Fig. 2.

CSRF. Cross Site Request Forgery (CSRF) [35] attacks target web session integrity. As an abstract example, consider the simple login program as shown in Fig. 9(left), where we leave out intermediate instructions that are not necessary to understand the subsequent repair. If the user attempts to log in and enters the correct password, we either set `request = 1` (modeling a login on the original page), or `request = 2` (modeling an attack, i.e., a login request

```
1  log(string password, string username,
2    string date) {
3    if(password == userPassword){
4      // password flows to credentials
5      credentials = username + password
6    } else {
7      credentials = username
8    }
9    // then flows to info
10   info = date + credentials
11   // then flows to LOG
12   LOG = info
13   observe
14 }
```

```
LOG = ""
```

(a)

```
LOG = date + username
```

(b)

Fig. 10. Privacy leakage by logging and repair candidates by HyRep.

at some untrusted website). We specify that the request should only depend on the (correctness of the) password. When repairing line 11, HyRep first discoverers the trivial repair that always overwrites request with a fixed constant (Fig. 9a). However, in the second improvement iteration, HyRep finds a better repair (Fig. 9b), where the request is only overwritten after a successful login. The potential attack request (request = 2) is thus deterministically overwritten.

LOG. We investigate privacy leaks induced by *logging* of credentials. We depict a simplified code snipped in Fig. 10. Crucially, in case of a successful login, the secret password flows into the public LOG (via credentials and info). We specify that the LOG may only depend on public information (i.e., everything except the password) and use HyRep to overwrite the final value of LOG (i.e., to repair line 12). As shown in Fig. 10a, HyRep first finds a trivial repair that does not log anything. In the first improvement iteration, HyRep automatically finds the more accurate repair in Fig. 10b. That is, it automatically infers that LOG can contain the date and username (as in the original program) but not the password.

```
1  atm(int balance, int amount) {
2    if (balance < amount){
3      ErrorLog = "overdraft"
4    } else {
5      balance = balance - amount
6      TransactionLog = "success"
7    }
8    ErrorLog = ErrorLog
9    TransactionLog = TransactionLog
10   observe
11 }
```

Fig. 11. An ATM that leaks the balance to ErrorLog and TransactionLog.

ATM. Many cases require repairing *multiple* lines of code simultaneously. We use cases derived from open-source security benchmarks [26,30,41] and mark multiple repair locations in the input programs. For example, consider the ATM

```
1  reviews(int reviewerAid, int reviewerBid,
2    string reviewA, string reviewB) {
3    notification = "Your␣CAV24␣reviews:"
4    if (reviewerAid <= reviewerBid){
5      order = 1
6    } else {
7      order = 2
8    }
9    order = order
10   if (order == 1) {
11     notification = notification + reviewA
12     notification = notification + reviewB
13   } else {
14     notification = notification + reviewB
15     notification = notification + reviewA
16   }
17   observe
18 }
```

```
order = 0
```

(a)

```
if (reviewerAid < 2) {
  order = order
} else {
  order = 2
}
```

(b)

Fig. 12. A review system that leaks the reviewer ids via the review order and repair candidates by `HyRep`.

program in Fig. 11. Depending on whether the withdraw `amount` is greater than `balance` (secret), different messages will be logged (public). To repair it, we need to repair both `ErrorLog` and `TransactionLog` under different conditions (i.e., do not update `ErrorLog` in the if-clause and do not update `TransactionLog` in the else-clause). By indicating lines 8 and 9 as two repair locations, `HyRep` is able to synthesize the correct multiline repair.

REVIEWS. We also investigate the review system depicted in Fig. 12(left). Here the `id` of each reviewer determines in which `order` the reviews are displayed to the author. We assume that the PC chair always has the fixed ID 1 (so if he/she submits a review, it will always be displayed first). We want to avoid that the author can infer which review was potentially written by the PC chair. When asked to repair line 9, `HyRep` produces the repair patches displayed in Figs. 12a and b. In particular, the last repair infers that if `reviewerAid < 2` (i.e., reviewer `A` is the PC chair), we can leave the order; otherwise, we use some fixed constant.

6.2 Scalability in Solution Size

Most modern SyGuS solvers rely on a (heavily optimized) *enumeration* of solution candidates [3,19,33,48]. The synthesis time, therefore, naturally scales in the size of the smallest solutions. Our above experiments empirically show that most repairs can be achieved by small patches. Nevertheless, to test the scalability in the solution size, we have designed a benchmark family that only admits large solutions. Concretely, we consider a program that computes the conjunction of

Table 2. In Table 2a, we evaluate HyRep's scalability in the SyGuS solution size. The timeout (denoted "-") is 120 s. In Table 2b, we repair a selection of k-safety instances from [7,23,49,53]. In Table 2c, we evaluate on a selection of functional repair instances from [27,44]. All times are given in seconds.

(a)

n	#Iter	t	Size
0	0	0.8	1
1	0	0.8	1
2	1	1.1	3
3	2	1.4	5
4	3	1.8	7
5	4	5.1	9
6	5	89.8	11
7	-	-	-

(b)

Instance	t
CollItemSym	1.4
CounterDet	4.9
DoubleSquareNiFF	4.2
DoubleSquareNi	2.9
Exp1x3	1.1
Fig2	2.4
Fig3	1.1
MultEquiv	2.0

(c)

Instance	t
Assignment	0.7
Deletion	0.7
Guard	0.6
Long-Output	0.7
Multiline	0.8
Not-Equal	0.6
SimpleExample	1.0
OffByOne	2.1

n Boolean inputs $i_1, \ldots, i_n$. We repair against a simple $\forall^2$ HyperLTL property which states that the output may not depend on the last input, guiding the repair towards the optimal solution $i_1 \wedge \cdots \wedge i_{n-1}$. We display the number of improvement iterations, the run time, and the solution size (measured in terms of AST nodes) in Table 2a. We note that one of the main features of SyGuS is the flexibility in the input grammar. When using a less permissive (domain-specific) grammar, HyRep scales to even larger repair solutions.

6.3 Evaluation on k-Safety Instances

To demonstrate that HyRep can tackle the repair problem in the size-range supported by current *verification* approaches for hyperproperties, we collected a small set of k-safety verification instances from [7,23,49,53]. We modify each program such that the k-safety property is violated and use HyRep's plain (non-iterative) SyGuS constraints to find a repair. The results in Table 2b demonstrate that (1) existing off-the-shelf SyGuS solver can repair programs of the complexity studied in the context of k-safety *verification*, and (2) even in the presence of loops (which are included in all instances in Table 2b), finite unrolling often suffices to generate repair constraints that yield repair patches that work for the full program.

6.4 Evaluation on Functional Properties

While we cannot handle the large programs supported by existing APR approaches for functional properties, we can evaluate HyRep on (very) small test cases. We sample instances from Angelix [44] and GenProg [27], and apply HyRep's direct (non-iterative) repair. We report the run times in Table 2c.

7 Related Work

APR. Existing APR approaches for functional properties can be grouped into search-based and constraint-based [25,28]. Approaches in the former category use a heuristic to explore a set of possible patch candidates. Examples include `GenProg` [27] and `PAR` [36], `SPR` [42], `TBar` [40], or machine-learning-based approaches [21,57]. These approaches typically scale to large code bases, but might fail to find a solution (due to the large solution space). Our approach falls within the latter (constraint-based) category. This approach was pioneered by `SemFix` [46] and later refined by `DirectFix` [43], `Angelix` [44], and `S3` [39]. To the best of our knowledge, we are the first to employ the (more general) SyGuS framework for APR, which leaves the exact search to an external solver. Most APR approaches rely on a finite set of input-output examples. To avoid overfitting [50] these approaches either use heuristics (to, e.g., infer variables that a repair should depend on [55]) or employ richer (e.g., MaxSMT-based) constraints [44]. Crucially, these approaches are *local*, whereas our repair constraints reason about the entire (*global*) program execution by utilizing the entire symbolic path. Any repair sequence generated by our iterative repair is thus guaranteed to increase in quality, i.e., preserve more behavior of the original program.

APR for Hyperproperties. Coenen et al. [15] study *enforcement* of alternation-free hyperproperties. Different from our approach, enforcement does not provide guarantees on the functional behavior of the enforced system. Bonakdarpour and Finkbeiner [9] study the repair-complexity of hyperproperties in *finite*-state transition systems. In their setting, a repair consists of a substructure, i.e., a system obtained by removing some of the transitions of the system, so the repair problem is trivially decidable. Polikarpova et al. [47] present `Lifty`, and encoding of information-flow properties using refinement types. `Lifty` can automatically patch a program to satisfy an information-flow requirement by assigning *all* private variables some public dummy default constant. In contrast, our approach can repair against complex temporal hyperproperties (possibly involving quantifier alternations), and our repair often goes beyond insertion of constants.

8 Conclusion

We have studied the problem of automatically repairing an (infinite-state) software program against a temporal hyperproperty, using SyGuS-based constraint generation. To enhance our basic SyGuS-based approach, we have introduced an iterative repair approach inspired by the notion of transparency. Our approach interprets "closeness" rigorously, encodes it within our constraint system for APR, and can consequently derive non-trivial repair patches.

Acknowledgments. This work was partially supported by the European Research Council (ERC) Grant HYPER (101055412), by the German Research Foundation (DFG) as part of TRR 248 (389792660), and by the United States NSF SaTC Awards 210098 and 2245114.

References

1. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: Computer Security Foundations Symposium, CSF 2016 (2016). https://doi.org/10.1109/CSF.2016.24
2. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013 (2013)
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017 (2017). https://doi.org/10.1007/978-3-662-54577-5_18
4. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
5. Baumeister, J., Coenen, N., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: A temporal logic for asynchronous hyperproperties. In: International Conference on Computer Aided Verification, CAV 2021 (2021). https://doi.org/10.1007/978-3-030-81685-8_33
6. Beutner, R., Finkbeiner, B.: Prophecy variables for hyperproperty verification. In: Computer Security Foundations Symposium, CSF 2022 (2022). https://doi.org/10.1109/CSF54842.2022.9919658
7. Beutner, R., Finkbeiner, B.: Software verification of hyperproperties beyond k-safety. In: International Conference on Computer Aided Verification, CAV 2022 (2022). https://doi.org/10.1007/978-3-031-13185-1_17
8. Beutner, R., Finkbeiner, B.: AutoHyper: explicit-state model checking for HyperLTL. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023. LNCS, vol. 13993, pp. 145–163. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_8
9. Bonakdarpour, B., Finkbeiner, B.: Program repair for hyperproperties. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 423–441. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_25
10. Bonakdarpour, B., Sanchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 8–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_2
11. Bozzelli, L., Peron, A., Sánchez, C.: Asynchronous extensions of HyperLTL. In: Symposium on Logic in Computer Science, LICS 2021 (2021). https://doi.org/10.1109/LICS52264.2021.9470583
12. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity and robustness of programs. Commun. ACM **55**(8) (2012). https://doi.org/10.1145/2240236.2240262
13. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: International Conference on Principles of Security and Trust, POST 2014 (2014). https://doi.org/10.1007/978-3-642-54792-8_15
14. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Computer Security Foundations Symposium, CSF 2008 (2008). https://doi.org/10.1109/CSF.2008.7
15. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J., Schillo, Y.: Runtime enforcement of hyperproperties. In: International Symposium on Automated Technology for Verification and Analysis, ATVA 2021 (2021). https://doi.org/10.1007/978-3-030-88885-5_19

16. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 121–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_7
17. Daniel, L., Bardin, S., Rezk, T.: Binsec/Rel: efficient relational symbolic execution for constant-time at binary-level. In: Symposium on Security and Privacy, SP 2020 (2020). https://doi.org/10.1109/SP40000.2020.00074
18. Daniel, L., Bardin, S., Rezk, T.: Hunting the haunter - efficient relational symbolic execution for Spectre with haunted RelSE. In: Annual Network and Distributed System Security Symposium, NDSS 2021 (2021)
19. Ding, Y., Qiu, X.: Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations. Proc. ACM Program. Lang. (POPL) (2024). https://doi.org/10.1145/3632913
20. Duret-Lutz, A., et al.: From spot 2.0 to spot 2.10: what's new? In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, CAV 2022. LNCS, vol. 13372, pp. 174–187. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13188-2_9
21. Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H.: Automated repair of programs from large language models. In: International Conference on Software Engineering, ICSE 2023 (2023). https://doi.org/10.1109/ICSE48619.2023.00128
22. Farina, G.P., Chong, S., Gaboardi, M.: Relational symbolic execution. In: International Symposium on Principles and Practice of Programming Languages, PPDP 2019 (2019). https://doi.org/10.1145/3354166.3354175
23. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 200–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_11
24. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
25. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. IEEE Trans. Softw. Eng. **45**(1) (2019). https://doi.org/10.1109/TSE.2017.2755013
26. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in DroidSafe. In: Annual Network and Distributed System Security Symposium, NDSS 2015 (2015)
27. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering, ICSE 2012 (2012). https://doi.org/10.1109/ICSE.2012.6227211
28. Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12) (2019). https://doi.org/10.1145/3318162
29. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Automata and fixpoints for asynchronous hyperproperties. Proc. ACM Program. Lang. (POPL) (2021). https://doi.org/10.1145/3434319
30. Hamann, T., Herda, M., Mantel, H., Mohr, M., Schneider, D., Tasch, M.: A uniform information-flow security benchmark suite for source code and bytecode. In: Nordic Conference on Secure IT Systems, NordSec 2018 (2018). https://doi.org/10.1007/978-3-030-03638-6_27
31. Hsu, T.-H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: TACAS 2021. LNCS, vol. 12651, pp. 94–112. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_6

32. Hsu, T., Sánchez, C., Sheinvald, S., Bonakdarpour, B.: Efficient loop conditions for bounded model checking hyperproperties. In: Sankaranarayanan, S., Sharygina, N. (eds.) International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2023. LNCS, vol. 13993, pp. 66–84. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_4
33. Huang, K., Qiu, X., Shen, P., Wang, Y.: Reconciling enumerative and deductive program synthesis. In: International Conference on Programming Language Design and Implementation, PLDI 2020 (2020). https://doi.org/10.1145/3385412.3386027
34. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. In: Weirich, S. (eds.) European Symposium on Programming Languages and Systems, ESOP 2024. LNCS, vol. 14577, pp. 212–241. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57267-8_9
35. Khan, W., Calzavara, S., Bugliesi, M., De Groef, W., Piessens, F.: Client side web session integrity as a non-interference property. In: Prakash, A., Shyamasundar, R. (eds.) ICISS 2014. LNCS, vol. 8880, pp. 89–108. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13841-1_6
36. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: International Conference on Software Engineering, ICSE 2013 (2013). https://doi.org/10.1109/ICSE.2013.6606626
37. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976). https://doi.org/10.1145/360248.360252
38. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_17
39. Le, X.D., Chu, D., Lo, D., Goues, C.L., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017 (2017). https://doi.org/10.1145/3106237.3106309
40. Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: TBar: revisiting template-based automated program repair. In: International Symposium on Software Testing and Analysis, ISSTA 2019 (2019). https://doi.org/10.1145/3293882.3330577
41. Livshits, B.: SecuriBench Micro (2014). https://github.com/too4words/securibench-micro
42. Long, F., Rinard, M.C.: Automatic patch generation by learning correct code. In: Symposium on Principles of Programming Languages, POPL 2016 (2016). https://doi.org/10.1145/2837614.2837617
43. Mechtaev, S., Yi, J., Roychoudhury, A.: DirectFix: looking for simple program repairs. In: International Conference on Software Engineering, ICSE 2015 (2015). https://doi.org/10.1109/ICSE.2015.63
44. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering, ICSE 2016 (2016). https://doi.org/10.1145/2884781.2884807
45. Ngo, M., Massacci, F., Milushev, D., Piessens, F.: Runtime enforcement of security policies on black box reactive programs. In: Symposium on Principles of Programming Languages, POPL 2015 (2015). https://doi.org/10.1145/2676726.2676978
46. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: International Conference on Software Engineering, ICSE 2013 (2013). https://doi.org/10.1109/ICSE.2013.6606623
47. Polikarpova, N., Stefan, D., Yang, J., Itzhaky, S., Hance, T., Solar-Lezama, A.: Liquid information flow control. Proc. ACM Program. Lang. (ICFP) (2020). https://doi.org/10.1145/3408987

48. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: CVC4SY: smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 74–83. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_5
49. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 161–179. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_9
50. Smith, E.K., Barr, E.T., Goues, C.L., Brun, Y.: Is the cure worse than the disease? Overfitting in automated program repair. In: Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015 (2015). https://doi.org/10.1145/2786805.2786825
51. Tiraboschi, I., Rezk, T., Rival, X.: Sound symbolic execution via abstract interpretation and its application to security. In: International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023 (2023). https://doi.org/10.1007/978-3-031-24950-1_13
52. Tsoupidi, R., Balliu, M., Baudry, B.: Vivienne: relational verification of cryptographic implementations in WebAssembly. In: Secure Development Conference, SecDev 2021 (2021). https://doi.org/10.1109/SECDEV51306.2021.00029
53. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 742–766. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_35
54. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **42**(8) (2016). https://doi.org/10.1109/TSE.2016.2521368
55. Xiong, Y., et al.: Precise condition synthesis for program repair. In: International Conference on Software Engineering, ICSE 2017 (2017). https://doi.org/10.1109/ICSE.2017.45
56. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Computer Security Foundations Workshop, CSFW 2003 (2003). https://doi.org/10.1109/CSFW.2003.1212703
57. Zhu, Q., et al.: A syntax-guided edit decoder for neural program repair. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021 (2021). https://doi.org/10.1145/3468264.3468544