Computation Tree Logic Guided Program Repair

Yu Liu*, Yahui Song*, Martin Mirchev, and Abhik Roychoudhury

Abstract—Temporal logics like Computation Tree Logic (CTL) have been widely used as expressive formalisms to capture rich behavioural specifications. CTL can express properties such as reachability, termination, invariants and responsiveness, which are difficult to test. This paper suggests a mechanism for the automated repair of infinite-state programs guided by CTL properties. Our produced patches avoid the overfitting issue that occurs in test-suite-guided repair, where the repaired code may not pass tests outside the given test suite. To realise this vision, we propose a novel find-and-fix framework based on Datalog, a widely used domain-specific language for program analysis, which readily supports nested fixed-point semantics of CTL via stratified negation. Specifically, our framework encodes the program and CTL properties into Datalog facts and rules and performs the repair by modifying the facts to pass the analysis rules. In the framework, to achieve both analysis and repair results, we adapt existing techniques – including loop summarisation and Symbolic Execution of Datalog (SEDL) – with key modifications. Our approach achieves analysis accuracy of 56.6% on a CTL verification benchmark and 88.5% on a termination/responsiveness benchmark, surpassing the best baseline performances of 27.7% and 76.9%, respectively. Our approach repairs all detected bugs, which is not achieved by existing tools.

Index Terms—Program Analysis and Automated Repair, Datalog, Loop Summarisation

1 Introduction

Computational Tree Logic (CTL) is based on a branching notion of time — at each moment, there may be several different possible futures — which is sufficiently expressive to formulate a rich set of properties for infinite-state programs, such as reactive systems. CTL can specify many critical properties to prevent bugs, such as non-termination [1], in the form of $AF(Exit())$; and unresponsive behaviours [2], in the form of $AG(\phi_1 \rightarrow AF\, \phi_2)$, *e.g.* whenever an error occurs, the server will eventually respond with the corresponding error code. Here, ϕ_1 and ϕ_2 are the CTL subformulae; A and E are universal/existential quantifiers over the execution paths, and F and G stand for *finally* and *globally*, respectively.

Typically, when a program fails to satisfy a CTL property, developers must examine counterexample traces identified by a model checker and manually fix them iteratively. Here, we propose a mechanism that, instead of requiring iterative fixes, deals with all counterexamples at once and automatically. To realize this vision, we propose a Datalog-based framework that encodes the given program's control flow into Datalog rules, the abstract program states into Datalog facts, and the CTL checking into Datalog query rules. The presence of the expected output fact indicates whether the program satisfies the specified property. The repair is then achieved by modifying the input facts to allow the query rules to output the expected fact. We chose Datalog for its inherent purity, which sufficiently captures the entire spectrum of CTL properties. Additionally, its symbolic execution capability – namely *Symbolic Execution of Datalog* (SEDL) [3] — can identify potential modifications to the input facts that influence the output facts, thereby repairing the code to satisfy the given property.

Specifically, symbols denoting unknown constants and the truthfulness of facts are injected into the input facts. The outcome of SEDL on these symbolic inputs summarizes the logical constraints over the symbols that enable the output. Then, any valuation of symbols that produces the desired query output corresponds to a patch. This allows us to repair all the violations of the property at once and find all possible repairs within the defined search space. Besides, the relation between CTL and Datalog is longstanding [4], showing that the semantics of CTL properties – nested least and greatest fixed points – can be readily supported by Datalog with stratified negation without introducing approximation. However, the challenges are: the current SEDL supports only positive Datalog programs, and it is unclear how to precisely handle loops in a CTL analysis.

To address the first challenge, we extend SEDL to support stratified negation. SEDL operates by over-approximating the domain of symbolic constants, then using delta-debugging [5] to select the dependent input facts that enable the expected output fact. There are two reasons why this is limited to positive Datalog: (i) its over-approximation only handles positive rules; and (ii) delta debugging requires monotonicity, which is only applicable to positive rules. In this work, we propose a new solution in which the over-approximation takes account of stratified negations. It uses an Answer Set Programming (ASP) solver that works with non-monotonic rules, and the generated fact modifications are sound by construction, *i.e.* they lead to the expected output results.

To address the second challenge, we propose summarizing programs, including the loops, using an intermedi-

- *Yu Liu is with the National University of Singapore E-mail:liu.yu@u.nus.edu*
- *Yahui Song is with the National University of Singapore E-mail: yahui.song@u.nus.edu*
- *Martin Mirchev is with the National University of Singapore E-mail: martin.mirchev@u.nus.edu*
- *Abhik Roychoudhury is with the National University of Singapore E-mail: abhik@nus.edu.sg*

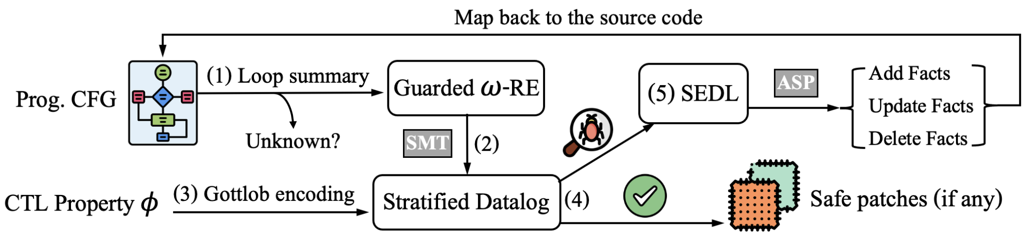

Fig. 1: System overview of CTLEXPERT

ate representation, denoted by Φ, which is a *guarded ω-regular language*. Unlike existing loop summarization techniques, which do not explicitly capture non-terminating behaviours, Φ enables us to capture both terminating and non-terminating behaviours in a (guarded) disjunctive form. Existing CTL analyzers suffer from imprecision in their termination analysis, primarily due to limitations of ranking function synthesis techniques [6], [7]. We show that our novel loop summarization enables an effective *linear ranking function* [8] generation by inspecting the guards of the disjunctive summaries. Next, the control flow and abstract program states represented by Φ can be encoded using a Datalog program, leveraging the Datalog execution for CTL property queries. Experimental results show that our tool CTLEXPERT can significantly improve the precision of the CTL checking. Our contributions are as follows:

- We design a novel find-and-fix framework that takes a program and a CTL property and returns the repaired program if the property is violated.
- We extend existing SEDL to support stratified negations, enabling repair guided by CTL properties that involve a mixture of least- and greatest-fix-point-defined analysis.
- We extend the existing loop summarisation by calculating both termination and non-termination summaries to prove both safety and liveness properties, which improves the precision of the analysis.
- We prototype our proposal and evaluate CTLEXPERT on two benchmarks: a CTL verification benchmark (featuring small-scale examples with complex loops) and a termination/responsiveness benchmark (derived from GitHub bug-fix commits). Our experiments demonstrate that CTLEXPERT surpasses state-of-the-art tools in detecting and fixing CTL bugs. The complete source code and benchmark dataset are publicly available at: https://doi.org/10.5281/zenodo.15896690

2 Overview and Illustrative Examples

As shown in Fig. 1, CTLEXPERT takes the program's control-flow graph (CFG) and a CTL specification and produces safe patches if the property does not hold. The main steps are highlighted in the rounded boxes and the arrows around them. In particular, we deploy an SMT solver [9] and an ASP solver [10] for solving linear arithmetic constraints and deciding the truth assignments for Datalog facts. The workflow of CTLEXPERT is as follows:

(1) Given any CFG, it is converted to our intermediate representation, *i.e.* the guarded ω-regular expression (or guarded ω-RE), demonstrated in Sec. 4.2. In this process, loops are managed using a summary calculus, as detailed in Sec. 4.3. Given the undecidable nature of loop termination analysis, the framework outputs "Unknown" when there exists a path for which we cannot conclusively prove either termination or non-termination.

(2) Given any guarded ω-RE, it is translated into a Datalog program, shown in Sec. 4.4.

(3) For any CTL property, we convert it into stratified Datalog rules, detailed in Sec. 4.1.

(4) The SEDL execution checks CTL properties precisely.

(5) When a given property does not hold, the erroneous Datalog program is sent for repair, as outlined in Sec. 5. The relationship between facts modifications and the source code is as follows: Adding facts leads to additional assignments along existing paths. Updating facts modifies the current assignments on those paths. Deleting facts necessitates the inclusion of conditional statements for early exits on problematic paths. Collectively, these modifications form patches that consist of iteratively adding or revising assignments and conditional statements.

CTL Analysis using Datalog. The program depicted in Fig. 2 initiates by assigning the value $y{=}1$ while allowing the variables i and x to assume any values. Here, the symbol $*$ makes all the nondeterminism explicit. Following this initialization, the program executes a conditional statement, which includes an assignment of $x{=}1$. Subsequently, the program enters a while loop, and once the while loop is entered, it results in infinite execution. Finally, before returning, it assigns the value $y{=}5$. There are three symbolic paths: (1) When $i{>}10$, it enters the infinite loop; (2) When $i{\leq}10 \land x{=}y$, it also enters the infinite loop; and (3) When $i{\leq}10 \land x{\neq}y$, it terminates normally with $y{=}5$. The CTL property ϕ of interest is "$AF(y{=}5)$", stating that *"for all paths, finally y's value is 5"*. However, the current implementation fails to satisfy ϕ as the first two paths (1) and (2) never reach the state where $y{=}5$.

The state-of-the-art tool FUNCTION [7] concludes 'unknown' for this example. To achieve a more precise CTL analysis for such infinite-state programs, we propose representing the program using Datalog facts and rules and leveraging the Datalog execution for sound and complete

```
void main(){ //AF(y=5)
  int y = 1;
  int i = *;
  int x = *;
  if (i > 10) {x = 1;}
  while (x == y) {}
  y = 5;
  return;}
```

Fig. 2: A Program Fails the CTL Specification

$$(y{=}1)@1\cdot(i{=}*)@2\cdot(x{=}*)@3\cdot\left(\vee\begin{array}{l}[i{>}10]@4\cdot(x{=}1)@5\cdot\left(\vee\begin{array}{l}[x{\neq}y]@7\cdot(y{=}5)@11\\ [x{=}y]@8\cdot((x{\geq}y)@12)^{\omega}\end{array}\right)\\ [i{\leq}10]@6\cdot\left(\vee\begin{array}{l}[x{\neq}y]@9\cdot(y{=}5)@11\\ [x{=}y]@10\cdot((x{\geq}y)@12)^{\omega}\end{array}\right)\end{array}\right)$$

Fig. 3: The Guarded ω-RE Representation, Φ_{main} (`@n` are uniquely assigned state numbers)

CTL checking. Specifically, we first convert programs into an intermediate representation, i.e., Φ_{main}, shown in Fig. 3, where $[\pi]$ denotes a guard upon pure constraints. These guards are derived from conditional, loop guards and assertions. For example, the loop in line 5 is summarized into the following disjunction $([x{\neq}y]\cdot\epsilon)\vee([x{=}y]\cdot(x{\geq}y)^{\omega})$, where "$\epsilon$" denotes an empty trace while "ω" denotes an infinite trace. This summary over-approximates the behaviours of the loop symbolically: when $x{\neq}y$ it terminates, and otherwise it infinitely repeats the state $x{\geq}y$.

Next, from Φ_{main}, the generated Datalog program is outlined in Fig. 4. For better visualization, here, the numbers in `red` refer to the state numbers in Fig. 3. Abstract predicates are over program variables, constants, and state numbers. The predicate `flow` represents the control flows; and some of them are persistent such as `flow(1,2)`, whereas some of them only exist if certain promises are satisfied, such as `flow(3,4)`, which only occurs when (transitivity) i is greater than 10 at state 3. Moreover, it is standard to encode finite traces into infinite traces by adding self-transition flows at last states, such as `flow(11,11)`. Facts are generated concerning the abstract states that lead to different paths, namely, whether $i{>}10$ and whether $x{=}y$. Also, since ϕ concerns the reachability of $y{=}5$, we generate Datalog facts *w.r.t.* the truth values of the following predicates: $i{>}10$, $i{\leq}10$, $x{=}y$, $x{\neq}y$, and $y{=}5$, which are abstracted using facts

```
// Persistent Transitions
flow(1,2). flow(2,3). flow(4,5).
flow(7,11). flow(8,12). flow(9,11).
flow(10,12). flow(11,11). flow(12,12).
// Conditional Transitions
flow(3,4) :- Gt(i,10,3).
flow(3,6) :- LtEq(i,10,3).
flow(5,7) :- NotEqVar(x,y,5).
flow(5,8) :- EqVar(x,y,5).
flow(6,9) :- NotEqVar(x,y,6).
flow(6,10):- EqVar(x,y,6).
// Abstract Predicates
Gt(i,10,2). LtEq(i,10,2). Eq(y,5,11).
EqVar(x,y,3). NotEqVar(x,y,3). EqVar(x,y,5).
```

Fig. 4: The (Simplified) Datalog Representation

```
yEQ5(S) :- Eq(y,5,S).

AFT_yEQ5(S,S1) :- !yEQ5(S), flow(S,S1).
AFT_yEQ5(S,S1) :- AFT_yEQ5(S,S2), !yEQ5(S2),
                  flow(S2,S1).

AFS_yEQ5(S) :- AFT_yEQ5(S,S).
AFS_yEQ5(S) :- !yEQ5(S),flow(S,S1),AFS_yEQ5(S1).

AF_yEQ5(S):- State(S), !AFS_yEQ5(S).
```

Fig. 5: Datalog Rules for "$AF(y{=}5)$"

`Gt`, `LtEq`, `EqVar`, `NotEqVar`, `Eq`, respectively.

The Datalog query rules generated for ϕ are shown in Fig. 5, and after executing the Datalog engine, the expected output fact R is `AF_yEQ5(1)`, which would indicate that ϕ holds at state 1 — the starting point of the program. Intuitively, a predicate "`AFT_yEQ5(S,S1)`" indicates that the property $y{=}5$ fails to hold at every state along the segment from `S` to `S1`. Therefore, "`AFT_yEQ5(S,S)`" denotes a cycle from `S` back to itself where $y{=}5$ is never satisfied. Next, "`AFS_yEQ5(S)`" captures the existence of a stem path starting at `S` leading to a specific program location, followed by a cycle back to the same location with $y{=}5$ remaining false throughout both the stem and the cycle. Finally, "`AF_yEQ5(S)`" is the negation of "`AFS_yEQ5(S)`", indicating that $y{=}5$ finally becomes true in all paths.

Since the variables i and x can assume any values, all possible execution paths (1), (2), and (3) are enabled. Consequently, after executing the Datalog engine, it fails to produce R. This result indicates that the program does not satisfy the specified property. At this stage, CTLEXPERT successfully *disproves* the property, paving the way for subsequent repairs to be implemented.

Need for Negation. The semantics of Datalog is defined by least fixed point semantics, while greatest fixed point properties can be represented using negation over Datalog predicates. CTL properties involve a combination of least and greatest fixed point properties. For example, consider $AGAF\varphi$, which can be encoded in Datalog through stratified negation.

ξ_1 `Gt(i,10,2).` ξ_2 `LtEq(i,10,2).`
ξ_3 `Eq(y,5,11).` ξ_4 `EqVar(x,y,3).`
ξ_5 `NotEqVar(x,y,3).` ξ_6 `EqVar(x,y,5).` ξ_7 `Eq(`α_1`,`α_2`,`α_3`).`

Fig. 6: Symbolic Facts

$$\begin{aligned}\psi \triangleq (&\neg\xi_1\wedge\xi_2\wedge\xi_3\wedge\neg\xi_4\wedge\xi_5\wedge\xi_6\wedge\neg\xi_7) && (a)\\ &\vee(\xi_1\wedge\xi_2\wedge\xi_3\wedge\xi_4\wedge\xi_5\wedge\xi_6\wedge\xi_7\wedge\alpha_1{=}y\wedge\alpha_2{=}5\wedge\alpha_3{=}12) && (b)\\ &\vee\ldots\end{aligned}$$

Fig. 7: Logical Constraints Enabling `AF_yEQ5(1)`

Repairing Property with Negation via Extended SEDL. Continuing with the earlier example, given that the expected output fact R = `AF_yEQ5(1)`, CTLEXPERT converts input facts to *symbolic input facts* by injecting symbolic constants $\alpha_1, \alpha_2, \alpha_3$, and symbolic signs $\xi_1, \ldots \xi_7$. A symbolic constant, α, represents an unknown value from its domain (*e.g.*, integers, strings), while a symbolic sign, ξ, represents an unknown Boolean value indicating the presence of its associated fact. After injecting symbols, those facts that contain symbolic constants or signs are shown in Fig. 6. We choose to explore modifications to variable states. Therefore, we assign symbolic signs to variable state facts such as `Gt(i,10,2)` and `EqVar(x,y,3)`. There are in total six variable state facts in the original EDB. Assigning symbolic signs to them means that these variable states may be removed — that is, the variable may no longer fall within the value ranges described by the facts. One additional symbolic sign is introduced for the fact Eq(α_1,α_2,α_3), which represents that variable α_1 is assigned the value α_2 at α_3. This fact expresses the assignment of a specific value to a variable. Since it is unclear whether this assignment is necessary, we also associate a symbolic sign with it. By applying the rules outlined in Fig. 4 and Fig. 5 to these symbolic facts and the unchanged facts, our SEDL generates the *logical constraints* ψ for R, as shown in Fig. 7. Each disjunctive case corresponds to a patch that enables the generation of R. The satisfying assignment of case (b) introduces the fact `Eq(y,5,12)`, which effectively adds the assignment `y=5` along the existing path, specifically within the body of the while loop. This patch successfully allows the program to pass the CTL analysis.

The satisfying assignment of (a) drops the newly introduced symbolic fact Eq(α_1,α_2,α_3), and deletes the existing facts: `Gt(i,10,2)` and `EqVar(x,y,3)`, indicating that neither should i be greater than 10 at state 2 nor should x equal y at state 3. The deleted facts suggest that when the condition $i{>}10 \vee x{=}y$ is met, the program fails to satisfy the intended property. As a result, during the first iteration of the repair process, a conditional statement is added: `if (i>10||x==y) return;`. This line is placed before the main logic of the program to prevent it from following the problematic execution path represented by the removed facts. However, the currently patched program still does not satisfy the condition $AF(y{=}5)$ because the newly added path does not reach the state where $y{=}5$. Therefore, during the second iteration, the analysis shows that the property does not hold. As a result, the repair inserts the statement $y{=}5$ before the return statement, similar to the generation of (a). These iterative processes ultimately lead to the correct conditional patch being added. Both source code level patches, referred to as Patch (a) and Patch (b), are illustrated in Fig. 8.

```
// Patch (a)
+ if (i>10 || x==y) {y = 5; return;}
if (i > 10) {x = 1;}
while (x == y) { + y = 5;}  // Patch (b)
y = 5; return; }
```

Fig. 8: Patch Options, Revised from Fig. 2

Remark. While simple, this example showcases our technical contributions in the following two aspects: loop summarization for both terminating and non-terminating behaviours and achieving the SEDL that accommodates stratified negations. Moreover, we define how to interpret fact modifications at the Datalog level into meaningful and general source-code level patches. To our knowledge, this work is the first to generate repairs based on a novel CTL analysis.

3 Preliminary

In this section, we introduce the background knowledge of the techniques we use in this paper.

3.1 Computational Tree Logic and Datalog

$$
\begin{array}{lrcl}
(CTL) & \phi & ::= & ap \mid \neg\phi \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid EF\phi \mid \\
 & & & EX\phi \mid AF\phi \mid E(\phi_1 U \phi_2) \mid \phi_1{\rightarrow}\phi_2 \mid \\
 & & & AX\phi \mid AG\phi \mid EG\phi \mid A(\phi_1 U \phi_2) \\
(AP) & ap & ::= & (p, \pi) \\
(Pure) & \pi & ::= & T \mid F \mid bop(t_1,t_2) \mid R \mid \pi_1{\wedge}\pi_2 \mid \pi_1{\vee}\pi_2 \\
(Terms) & t & ::= & v \mid \_ \mid t_1{+}t_2 \mid \text{-}t \\
(Relation) & R & ::= & p\,(v^*)
\end{array}
$$

Fig. 9: CTL Syntax

Computational Tree Logic (CTL) is a branching-time logic for reasoning about multiple possible execution paths. As shown in Fig. 9, we capture each atomic proposition as a pure formula π with a unique identifier p. Pure formulas are quantifier-free arithmetic predicates over program variables and constants. Binary operators are represented using predicates, where $bop \in \{>, <, \geq, \leq, =\}$. Other uninterpreted relations are represented using relational predicates R. A predicate has a name (p) and a set of arguments (v^*). Terms consist of simple values, the wild card _, and simple computations of terms. Each temporal operator contains a quantifier over paths: "A" means *for all the paths*, while "E" means *there exists a path*; and a linear temporal logic [11] operator: "F" for finally, "G" for globally, "U" for until, and "X" for next time. We use the $*$ superscript to denote a finite set of items.

$$
\begin{array}{lrcl}
(Datalog) & \mathbb{D} & ::= & R^* +\!\!+\, Q^* \\
(Rule) & Q & ::= & R \,\text{:-}\, L^* \\
(Literal) & L & ::= & R \mid\, !\,R \\
(Relation) & R & ::= & p(v^*) \\
(Arguments) & v & ::= & c \mid X
\end{array}
$$

Fig. 10: A Core Syntax of Datalog

The core syntax of Datalog is shown in Fig. 10. A Datalog program consists of a set of facts (R^*) and rules (Q^*). Arguments are constants (c), or program variables (X). A Datalog rule is a Horn clause that comprises a head literal (an atom) and a set of body literals, with the head on the left side and the body on the right side of the arrow symbol (:−). A fact is a rule with an empty body, *i.e.* it is unconditionally true. A Datalog query is executed against a database of facts, known as the *extensional database* (EDB), and produces a set of derived facts, known as the *intensional database* (IDB). Unification in Datalog is a pattern-matching operation determining whether two arguments can be made identical through substitution. Arguments unify if they are identical constants or if one is a variable. The semantics of Datalog is based on the least fixed point computation, where facts are iteratively derived by applying rules to existing facts until no new facts can be generated. Datalog with stratified negation can be partitioned into a finite number of Datalog programs, capturing the different strata. If the rule producing the R contains R' negated in the body, then

R and R' are in different partitions, and R is in a higher strata than R'. The least fixed point of the lower strata is computed first and then used to compute the least fixed point in the higher strata.

3.2 Symbolic execution of Datalog

Symbolic Execution of Datalog (SEDL) aims to identify how varying database values impact query results by executing Datalog queries on a symbolic database, representing a range of concrete databases. SEDL uses symbolic constants and signs. Symbolic constants represent unknown arguments in facts, such as `flow(`α,β`)`, where α and β are symbolic constants over a finite domain of state numbers. Symbolic signs, denoted as ξ are Boolean values indicating the presence of facts. Collectively, these symbols form $\Sigma \triangleq \{\sigma_1,\ldots,\sigma_n\}$ over domains $D_1,\ldots,D_n$, ranging from integers, Booleans, strings. Any concrete *valuation* of Σ, is represented as $\{\sigma_1=v_1,\ldots,\sigma_n=v_n\}$, where $\forall\, i\in\{1\ldots n\}.\ v_i\in D_i$. We use ψ^* to represent all the *logical constraints* for Σ. Given any ψ, its satisfying assignment is a concrete valuation of Σ. A symbolic EDB, denoted by $\mathcal{E}$, is a set of input facts that contain symbolic constants and signs. The concretization of a symbolic EDB is obtained by applying a concrete valuation. The symbolic execution upon any symbolic EDB produces a set of pairs, and each pair contains an output fact R and the corresponding logical constraints enabling the generation of R. More specifically, it takes $\mathcal{E}$ and returns $(R\times\psi^*)^*$.

3.2.1 Implementation

SEDL is implemented in SYMLOG [3], which uses a meta-programming approach. Specifically, given a Datalog query and a symbolic EDB, it converts them into a meta-query and a meta-EDB so that executing the meta-query on the meta-EDB with standard Datalog semantics produces the symbolic execution output of the original query on the original symbolic database. Consider the first rule in Fig. 5: `AF_yEQ5(S):- Eq(y,5,S).` When symbolic constants $\alpha_1,\ldots,\alpha_n$ are injected into the original EDB, SYMLOG converts this rule into the following meta-query: (where $C_1,\ldots,C_n$ are auxiliary variables)

```
AF_yEQ5(S, $C_1$, ..., $C_n$) :- Eq(y,5,S, $C_1$, ..., $C_n$).
```

Each concrete instantiation of $C_1,\ldots,C_n$ is a concrete valuation of $\alpha_1,\ldots,\alpha_n$. Given a symbolic EDB fact `Eq(y,5,`α_1`)`, it is transformed into the following meta-EDB rule: (where the predicate `domain_alpha_`i is true for all values from the domain of α_i)

```
Eq(y,5,$C_1$, $C_1$, ..., $C_n$) :- domain_alpha_1($C_1$),...,
    domain_alpha_n($C_n$).
```

SYMLOG computes the constraints in two steps: (1) Constraints over Symbolic Constants. When the meta-query is evaluated on the meta-EDB, each value of C_i represents a possible instantiation of the corresponding symbolic constant α_i, collectively forming the constraints over the symbolic constants. The output facts with the same assignments of auxiliary variables share the same constraints over the symbolic constants. While a symbolic constant could theoretically take any value from an infinite domain (such as integers or strings), in practice, we only need to consider values that could potentially match with terms in the existing facts through unification. SYMLOG overapproximates the domain of a symbolic constant by analyzing how values flow through the Datalog program, tracking dependencies between predicates and constants to determine which values each symbolic constant could potentially take during execution. This over-approximation ensures that all possible instantiations of a symbolic constant are considered. (2) Constraints over symbolic signs. For output facts sharing the same symbolic constant constraints, SYMLOG uses delta-debugging [5] to identify the sets of input facts (annotated with symbolic signs) that are necessary to derive each output fact. Delta-debugging is a divide-and-conquer technique identifying the minimal subset of inputs necessary to produce a target output. The symbolic signs of the identified input facts must be true, while the values of other symbolic signs are false.

Given each output fact associated with concrete instantiation of the auxiliary variables, the constraint ψ is constructed by the conjunction of two types of constraints: those over symbolic signs and those over symbolic constants. The final constraints for the expected output fact are formed by taking the disjunction of all such ψ, resulting in the pair (R,ψ^*).

3.2.2 Datalog Repair

Datalog is widely used in program analysis, where EDB represents the analyzed program, and IDB represents the analysis results. SEDL guides program repair in several ways. For instance, if SEDL produces (R,ψ^*), and R indicates the expected results, then a satisfying assignment of ψ^* suggests a patch that enables the desired output. Conversely, if R indicates a bug, the satisfying assignment of $\neg\psi^*$ points to a patch that can eliminate the bug.

3.2.3 Limitation of the Existing SEDL

SYMLOG only supports positive Datalog. For symbolic constants, its domain approximation cannot handle negation. We resolve this by removing negations from the rules and then applying its method. For symbolic signs, its deployed delta-debugging requires the program to be monotonic, *i.e.* when input facts increase, the output of a rule must not decrease. We resolve this using an ASP solver to compute truth assignments for the symbolic signs. Overall, we achieve a SEDL that accommodates stratified negations and thereby supports the repair for CTL-defined analysis, detailed in Sec. 5.

4 CTL ANALYSIS USING DATALOG

In this section, we outline the essential steps for conducting a CTL analysis using Datalog.

4.1 From CTL Properties to Datalog Rules

Given a CTL formula ϕ, the relation "$CTL2D(\phi)\rightsquigarrow(p,Q^*)$" holds if ϕ can be translated into a set of Datalog rules Q^*. The validity of ϕ against a program is then indicated by the presence of the IDB predicate p after executing Q^* against the program facts. Since most of the encoding rules are standard [12], we selectively present them in Fig. 12, where we use "++" for both string and list concatenation.

For instance, in $[CD\text{-}AP]$, given a CTL formula containing one atomic proposition (p, π), it produces the Datalog rule "$p(S) :\!- \pi(S)$", which generates the predicate p for all states that satisfy the pure constraint π.

Different from prior work [12], which relies on the "findall" operator – to encode the AF operator. However, since "findall" introduces logical impurity and is not supported by Datalog, we adopt the encoding from the work of [4], which enables the greatest fixed point encoding using the least fixed point semantics of Datalog while maintaining the Datalog purity. Intuitively, given a CTL formula $AF\,\phi$, the resulting Datalog rules from $[CD\text{-}AF]$ are to prove the absence of the *lasso-shaped* [13] counterexamples, *i.e.* "$p_{\mathtt{s}}$" – a stem path to a particular program location followed by a cycle that returns to the same program location, *i.e.* the "$p_{\mathtt{t}}(S, S)$", and the property ϕ does not hold along the stem and the cycle. One example for $AF(y{=}5)$ is shown in Fig. 5.

```
xEQ5(S)       :- Eq(x,5,S).
EF_x_EQ_5(S) :- xEQ5(S).
EF_x_EQ_5(S) :- flow(S,S1), EF_x_EQ_5(S1).
```

Fig. 11: Datalog Rules for "$EF(x{=}5)$"

The Datalog encoding rule for "EF" is omitted, but we show an example using $EF(x{=}5)$ in Fig. 11, which recursively searches for the reachability of states which satisfy $x{=}5$. Similarly, "AG" and "EG" can be encoded as the negation of "EF" and "AF", respectively; implications can be encoded using disjunction and negation, etc.

4.2 From Programs to Guarded Omega-Regular Expressions

As defined in Fig. 13, any program $\mathcal{P}$ consists of a set of functions, which is identified by a name p, and represented by a CFG with a starting node N and a transition function $\mathcal{T}$, which returns all the immediate successors of a given node. Each node is parameterized with a unique (integer) state identifier s. We assume that the sets of nodes in different functions are pairwise disjoint. Apart from the *Start* and *Exit*, *Join* is used to connect disjunctive paths (usually created by conditionals and loops), while *Prune* associates an arithmetic guard before a node. Finally, *Stmt* stores statements such as assignments, returns, and function calls where the return value is explicitly denoted by r.

As shown in Fig. 14, the Guarded ω-RE formulas are similar to those in the classic *Omega Regular Expressions*, containing $\bot$ for *false*, ϵ for empty traces, sequence concatenations $(\Phi_1 \cdot \Phi_2)$, disjunctions $(\Phi_1 \vee \Phi_2)$, and infinite repetitions of a trace by Φ^{ω}. The main difference is, in Guarded ω-RE, singleton events can be either π or $[\pi]$. The former updates the program states with respect to the π formula, while the latter serves as program guards. Program guards are the sole means of introducing and controlling non-determinism. For the formula "$[\pi] \cdot \Phi$", the guarded trace Φ is executed only when the guard does not fail. When checking a guard, a Boolean expression is evaluated. If it denotes false, the guard fails. If it denotes true, it allows the execution to proceed without affecting the program states. Moreover, Guarded ω-RE does not contain Kleene stars as it aims to eliminate the unknown number of repetitions using either fixed-number-length finite traces or ω formulas for infinite executions.

Each function is converted to a Guarded ω-RE formula via Algorithm 1, which enumerates all paths from the CFG and replaces the cycles using the loop summaries. If the current node is an *exiting* node (line 2), it returns its corresponding Φ formula. Here, $\circlearrowleft$(N) maps from nodes to Φ. If the current node is a *Join*, it firstly checks if it leads to a cycle via calling EXISTSCYCLE (line 4), which returns "None" if none of its successors directly leads to cycles; otherwise, it returns "Some $(\pi_g$, Φ_{cycle}, $N_{nonCycleSucc})$", indicating that one successor of N leads to a path with guard π_g that comes back to itself, and Φ_{cycle} describes the behaviour of one iteration of the cycle, and $N_{nonCycleSucc}$ is the other successor, which does not directly lead to a cycle. The EXISTSCYCLE is implemented using a depth-first search algorithm. The cycle behaviours are sent to a summary calculus (line 8). In all other cases (lines 5 and 9), it uses an auxiliary function MOVEFORWARD to accumulate the effects of the current node and recursively combine the behaviours of all the following nodes. If there are no successors, MOVEFORWARD terminates; otherwise, it goes through successors and applies CFG2GWRE to calculate the formulas for the remainder of the path. It then disjunctively combines the outcomes.

Definition 4.1 (CFG Nodes to Guarded ω-RE)**.** Given any

Algorithm 1 CFG2GWRE

Require: A node N, and a transition function $\mathcal{T}$
Ensure: The final Guarded ω-RE, Φ

```
1:  if $N$ matches $Exit(s)$ or $Stmt(return(x), s)$ then
2:      return $\circlearrowleft(N)$
3:  else if $N$ matches $Join(s)$ then
4:      cycleInfo ← EXISTSCYCLE $(N, \mathcal{T})$
5:      if cycleInfo is None then
6:          return MOVEFORWARD $(N, \mathcal{T})$
7:      else
8:          Extract $(\pi_g, \Phi_{cycle}, N_{nonCycleSucc})$ from cycleInfo
9:          $\Phi_{rest} \leftarrow$ MOVEFORWARD$(N_{nonCycleSucc}, \mathcal{T})$
10:         return LOOPSUMMARY$(\pi_g, \Phi_{cycle}, \Phi_{rest})$
11:     end if
12: else
13:     return MOVEFORWARD $(N, \mathcal{T})$
14: end if
15:
16: Function MOVEFORWARD$(N, \mathcal{T})$
17: if $\mathcal{T}(N) = []$ then
18:     return $\circlearrowleft(N)$
19: else
20:     $\Phi_{acc} \leftarrow \bot$
21:     for all $N' \in \mathcal{T}(N)$ do
22:         $\Phi_{acc} \leftarrow \Phi_{acc} \vee$ CFG2GWRE$(N', \mathcal{T})$
23:     end for
24:     return $(\circlearrowleft(N) \cdot \Phi_{acc})$
25: end if
```

$$\frac{Q{=}p(S) \mathrel{:-} \pi(S)}{\mathit{CTL2D}((p,\pi)) \rightsquigarrow (p,[Q])}\ [\mathit{CD\text{-}AP}] \qquad \frac{\mathit{CTL2D}(\phi) \rightsquigarrow (p,Q^*) \qquad p_{\mathtt{new}}{=}\text{“}\mathit{NOT\_}\text{”} \mathbin{++} p \qquad Q'{=}p_{\mathtt{new}}(S) \mathrel{:-} !\,p(S)}{\mathit{CTL2D}(\neg\phi) \rightsquigarrow (p_{\mathtt{new}}, Q^* \mathbin{++} [Q'])}\ [\mathit{CD\text{-}Neg}]$$

$$\frac{\begin{array}{c}\mathit{CTL2D}(\phi) \rightsquigarrow (p,Q_1^*) \qquad p_{\mathtt{new}}{=}\text{“}\mathit{AF\_}\text{”} \mathbin{++} p \qquad p_{\mathtt{s}}{=}\text{“}\mathit{AFS\_}\text{”} \mathbin{++} p \qquad p_{\mathtt{t}}{=}\text{“}\mathit{AFT\_}\text{”} \mathbin{++} p \\ Q^*{=}\begin{bmatrix} p_{\mathtt{t}}(S,S') \mathrel{:-} !\,p(S), \mathtt{flow}(S,S') & p_{\mathtt{t}}(S,S') \mathrel{:-} p_{\mathtt{t}}(S,S''), !\,p(S''), \mathtt{flow}(S'',S') \\ p_{\mathtt{s}}(S) \mathrel{:-} p_t(S,S) \qquad p_{\mathtt{s}}(S) \mathrel{:-} !\,p(S), \mathtt{flow}(S,S'), p_{\mathtt{s}}(S') & p_{\mathtt{new}}(S) \mathrel{:-} !\,p_{\mathtt{s}}(S)\end{bmatrix}\end{array}}{\mathit{CTL2D}(AF\ \phi) \rightsquigarrow (p_{\mathtt{new}}, Q_1^* \mathbin{++} Q^*)}\ [\mathit{CD\text{-}AF}]$$

Fig. 12: Selected CTL-to-Datalog Encoding (The positive predicate $\mathtt{State}(S)$ is implicitly inserted to ground the variables)

N, its Φ formula is defined as follows:

$$\circlearrowleft(\mathit{Stmt}(x{:=}t,s)) = (x{=}t)\mathtt{@s}$$
$$\circlearrowleft(\mathit{Stmt}(\mathit{return}(\_),s)) = ((\mathit{Exit}())\mathtt{@s})^\omega$$
$$\circlearrowleft(\mathit{Prune}(\pi,s)) = [\pi]\mathtt{@s} \qquad \circlearrowleft(\mathit{Exit}(s)) = ((\mathit{Exit}())\mathtt{@s})^\omega$$
$$\circlearrowleft(\mathit{Stmt}(p(x^*,r),s)) = \begin{cases} (p(x^*))\mathtt{@s} \cdot (r{=}*)\mathtt{@s} \\ \qquad \mathit{when}\ (p \notin \mathcal{P}) \\ \Phi_p[x^*/y^*, r/\mathit{ret}] \\ \qquad \mathit{when}\ (\Phi_p(y^*,\mathit{ret}) \in \mathcal{P}) \end{cases}$$

The function $\circlearrowleft$, as described in Definition 4.1, maps the Prune nodes into guards and excludes the not mentioned node constructs (Start and Join), using ϵ. Function calls with undefined callees are modelled as non-deterministic choices, denoted by $r{=}*$. When the callee is defined, we retrieve its summary Φ_p and instantiate it by substituting formal arguments with actual arguments, denoted by $\Phi_p[x^*/y^*, r/\mathit{ret}]$.

4.3 From CFG Cycles to Guarded Omega-Regular Expressions

Targeting sequential non-recursive infinite-state programs, summaries are constructed from the innermost loop. In the case of nested loops, inner loops are expected to be replaced with summaries first. At any point during the analysis, the problem is therefore reduced to the analysis of a single loop. Intuitively, our loop summaries aim to replace terminating behaviours using their final states and convert non-terminating behaviours using ω constructs over finite traces.

The LOOPSUMMARY function is detailed in Algorithm 2, which takes three arguments: a loop guard π_g, the repetitive behaviours of the loop Φ_{cycle}, and the behaviours after the loop Φ_{rest}. Instead of being directly dedicated to the ranking function synthesis problem, the algorithm obtains a set of *candidate ranking functions* (CRFs) from the guard of the repetitive case π_g, denoted by $\mathcal{CRF}(\pi_g)$.

$(\mathit{Program})$	$\mathcal{P}$	$::=$	func^*
$(\mathit{Func.\ Def.})$	func	$::=$	$p = (N, \mathcal{T})$
$(\mathit{Statements})$	e	$::=$	$x{:=}t \mid \mathit{return}(x) \mid p(x^*, r)$
$(\mathit{Trans.\ Fun.})$	$\mathcal{T}$	$::=$	$N \to N^*$
(Nodes)	N	$::=$	$\mathit{Start}(s) \mid \mathit{Exit}(s) \mid \mathit{Join}(s) \mid \mathit{Prune}(\pi, s) \mid \mathit{Stmt}(e, s)$

Fig. 13: CFG Structure of Target Programs

$$\Phi ::= \bot \mid \epsilon \mid \pi\mathtt{@s} \mid [\pi]\mathtt{@s} \mid \Phi_1 \cdot \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi^\omega$$

Fig. 14: The Syntax of Guarded ω-RE

For each CRF, we compute the weakest precondition (WPC) for termination and non-termination, denoted by π^T_{wpc} and π^{NT}_{wpc}, respectively. Here, the difference between the initial value of rf and the updated value rf' after the transition Φ is denoted by $\Delta rf(\Phi)$, *i.e.* $\Delta rf(\Phi) = rf\text{ - }rf'$. Intuitively, under π^T_{wpc}, rf can be proven to be strictly decreasing for all the transitions in the loop body; thus, it leads to terminating executions. Likewise, under π^{NT}_{wpc}, rf can be proven to be strictly not-decreasing for all the transitions in the loop body; thus, it leads to non-terminating executions.

To achieve a sound CTL analysis, we only obtain conclusive results when the union of these two WPCs covers the full path upon entering the loop (checked in line 4). This means that the summary should not contain any paths for which we cannot determine whether they lead to termination or non-termination. Otherwise, we report "Unknown" and quit the analysis. In the loop summary, Φ^1_{term} denotes the case when the execution did not enter into the loop at all, Φ^2_{term} denotes the case when entering the loop and terminates eventually, and $\Phi_{nonTerm}$ denotes the case when entering the loop and getting into an infinite execution where $rf{\geq}0$ is the recurrent state [8] that witnesses non-termination. The status of other (non-CRF) program variables can be included in the summary relative to terminating and non-terminating cases, respectively.

Algorithm 2 Loop Summary Computation

```
Require: $\pi_g$, $\Phi_{\text{cycle}}$, $\Phi_{\text{rest}}$
Ensure: A loop summary $\Phi$ or Unknown
  for all rf $\in$ RF($\pi_g$) {Obtain CRFs} do
    $\pi^T_{\text{wpc}} \leftarrow \Delta\text{rf}(\Phi_{\text{cycle}}) \geq 1$
    $\pi^{\text{NT}}_{\text{wpc}} \leftarrow \Delta\text{rf}(\Phi_{\text{cycle}}) < 1$
    if $\pi_g \Rightarrow \left(\pi^T_{\text{wpc}} \vee \pi^{\text{NT}}_{\text{wpc}}\right)$ {Full Path Conclusive} then
      $\Phi^1_{\text{term}} \leftarrow [\neg\pi_g] \cdot \Phi_{\text{rest}}$
      $\Phi^2_{\text{term}} \leftarrow [\pi_g \wedge \pi^T_{\text{wpc}}] \cdot (\text{rf} < 0) \cdot \Phi_{\text{rest}}$
      $\Phi_{\text{nonTerm}} \leftarrow [\pi_g \wedge \pi^{\text{NT}}_{\text{wpc}}] \cdot (\text{rf} \geq 0)^\omega$
      return $\Phi^1_{\text{term}} \vee \Phi^2_{\text{term}} \vee \Phi_{\text{nonTerm}}$
    end if
  end for
  return Unknown {Inconclusive Result}
```

Example 1. Revisiting the loop shown in Fig. 2, when triggering LOOPSUMMARY, the arguments are $\pi_g{=}(x{=}y)$, $\Phi_{cycle}{=}\epsilon$ and $\Phi_{rest}{=}(y{=}5)$. By Definition 4.2, we obtain: $rf{=}(x\text{-}y)$ and $rf \in \mathcal{CRF}(x{=}y)$. Based on rf, we compute $\pi^{T}_{wpc}{=}F$ and $\pi^{NT}_{wpc}{=}T$, as rf never decreases. Thus, we conclude the final summary to be $([x{\neq}y] \cdot (y{=}5)) \vee ([x{=}y \wedge F] \cdot (rf{<}0) \cdot (y{=}5)) \vee ([x{=}y \wedge T] \cdot (rf{\geq}0)^{\omega})$; which reduces to the summary concluded in Fig. 3, *i.e.* $([x{\neq}y] \cdot (y{=}5) \vee [x{=}y] \cdot (x{\geq}y)^{\omega})$.

Definition 4.2 (Generating CRFs from Pure)**.** Given any loop guard π on CFG, we propagate a set of terms which are candidate ranking functions: ($\emptyset$ for unmentioned constructs)

$$\mathcal{CRF}(t_1{\geq}t_2){=}\{t_1\text{-}t_2\} \qquad \mathcal{CRF}(t_1{\leq}t_2){=}\{t_2\text{-}t_1\}$$
$$\mathcal{CRF}(t_1{>}t_2){=}\{t_1\text{-}t_2\text{-}1\} \qquad \mathcal{CRF}(t_1{<}t_2){=}\{t_2\text{-}t_1\text{-}1\}$$
$$\mathcal{CRF}(t_1{=}t_2){=}\{(t_1\text{-}t_2);(t_2\text{-}t_1)\}$$

Theorem 4.1 (Soundness of the Generation of CRFs)**.** If the generated CRFs, from Definition 4.2, decreases at each iteration of the cycle, the cycle does terminate.

Proof sketch. By case analysis of the possible loop guard π. For example, when $\pi{=}(t_1{\geq}t_2)$, and $rf{=}t_1\text{-}t_2$: to enter the loop, the state must satisfy $rf{\geq}0$, if rf is decreasing at each iteration, it will finally reach the state $rf{<}0$, *i.e.* $t_1\text{-}t_2{<}0$, which no longer satisfy the loop guard; thus, the loop is terminating. Similar proofs for the rest of the cases. □

Since all the conjunctions and disjunctions in arithmetic constraints are systematically decomposed by the CFG construction, we soundly over-approximate the set of CRFs using Definition 4.2, meaning that if one of them concludes termination, the loop must be terminating. In particular, there are two possible cases for a loop guard of the form $t_1{=}t_2$: either $t_1\text{-}t_2$ is decreasing, or $t_2\text{-}t_1$ is decreasing. Consequently, the resulting set contains exactly two elements. The soundness is defined in Theorem 4.1. However, this approach may lack completeness, as we focus on loops that can be proven terminating through *linear ranking functions* (LRFs) [14], where loops are ranked linearly. As a result, the actual ranking functions — when there exist leaking paths in the cycle or those that progress through phases — may not be generated adequately. These situations will be classified as Unknown in the current setup. Computing loop summaries for other types of ranking functions is considered future work.

4.4 From Guarded ω-RE to Datalog Programs

There are two tasks for generating a Datalog program given a Guarded ω-RE Φ, and a CTL property ϕ: produce the rules for conditional flows, and map concrete program states into abstract predicates in the form of facts. First, we provide the definitions of the deployed auxiliary functions. Informally, the *Nullable* function $\delta(\Phi)$ returns a Boolean value indicating whether Φ contains the empty trace; the *First* function $fst(\Phi)$ computes a set of possible initial trace segments from Φ; the *Derivative* function $\mathcal{D}_f(\Phi)$ eliminates a segment f from the head of Φ and returns what remains.

Definition 4.3 (Nullable)**.** Given any Φ, we define $\delta(\Phi)$ as follows: (*false* for unmentioned constructs)

$$\delta(\epsilon){=}true \qquad \delta(\Phi_1 \cdot \Phi_2){=}\delta(\Phi_1) \wedge \delta(\Phi_2)$$
$$\delta(\Phi_1 \vee \Phi_2){=}\delta(\Phi_1) \vee \delta(\Phi_2)$$

Definition 4.4 (First)**.** We define $fst(\Phi)$ to be the set of initial segments derivable from a Φ formula.

$$fst(\pi@\mathtt{s}){=}\{\pi@\mathtt{s}\} \qquad fst([\pi]@\mathtt{s}){=}\{[\pi]@\mathtt{s}\}$$
$$fst(\Phi_1 \vee \Phi_2){=}fst(\Phi_1) \cup fst(\Phi_2) \qquad fst(\Phi^{\omega}){=}\{\Phi^{\omega}\}$$
$$fst(\Phi_1 \cdot \Phi_2){=}\begin{cases} fst(\Phi_1) \cup fst(\Phi_2) & when \quad \delta(\Phi_1){=}true \\ fst(\Phi_1) & when \quad \delta(\Phi_1){=}false \end{cases}$$

Definition 4.5 (Derivative)**.** The derivative $\mathcal{D}_f(\Phi)$ subtracts a trace segment f from the head of Φ and returns what remains, defined as follows: ($\perp$ for unmentioned constructs)

$$\mathcal{D}_{\pi@\mathtt{s}}(\pi@\mathtt{s}){=}\epsilon \qquad \mathcal{D}_{[\pi]@\mathtt{s}}([\pi]@\mathtt{s}){=}\epsilon$$
$$\mathcal{D}_{\Phi^{\omega}}(\Phi^{\omega}){=}\epsilon \qquad \mathcal{D}_f(\Phi_1 \vee \Phi_2){=}\mathcal{D}_f(\Phi_1) \vee \mathcal{D}_f(\Phi_2)$$
$$\mathcal{D}_f(\Phi_1 \cdot \Phi_2) = \begin{cases} (\mathcal{D}_f(\Phi_1) \cdot \Phi_2) \vee \mathcal{D}_f(\Phi_2) \\ \quad when\ \delta(\Phi_1) = true \\ \mathcal{D}_f(\Phi_1) \cdot \Phi_2 \\ \quad when\ \delta(\Phi_1) = false \end{cases}$$

As shown in Fig. 15, translation rules are in the form of $\Pi \vdash GWRE2D(s_p, \Phi) \rightsquigarrow \mathbb{D}$, where Π is a context containing a set of abstract predicates, s_p is the preceding state, and the formula Φ will be converted into the Datalog program $\mathbb{D}$. The translation is initially invoked with $\Pi{=}Pure(\Phi) \cup Pure(\phi)$, $s_p{=}\text{-}1$ and $\pi_{path}{=}T$. We use $Pure(\Phi)$ to extract the predicates from the guards in Φ, and use $Pure(\phi)$ to extract the atomic propositions in ϕ. In total, Π gathers all the abstract predicates of interest.

When the given Φ contains no *initial* elements, *i.e.* it is already the end of the trace, $[GD\text{-}Base]$ adds a self-transition flow. Otherwise, $[GD\text{-}Ind]$ unions the Datalog programs generated from each initial segment and their derivatives via the relation $\Pi \vdash (f, s_p, \Phi) \hookrightarrow \mathbb{D}$. There are three kinds of initial segments: When $f{=}\Phi^{\omega}$, apart from the Datalog program generated from Φ, $[GD\text{-}Omega]$ generates the flow facts connecting end and start states of Φ; When $f{=}\pi@\mathtt{s}$, $[GD\text{-}Pure]$ generates a flow rule from the previous state to the current state and generates facts for predicates, which are entailed by the current state, denoted by $\{\pi'(s) \mid \forall \pi' \in \Pi \ .\ \pi@\mathtt{s} {\Rightarrow} \pi')\}$ where the implication of $\pi@\mathtt{s}{\Rightarrow}\pi$ is solved by a SMT solver [9]. For example, if the concrete state is $y{=}1$ at state s and Π includes $y{\geq}1$ and $y{<}1$, then it generates one fact `GtEq("y",1,s)`, since $(y{=}1){\Rightarrow}(y{\geq}1)$ and $(y{=}1){\not\Rightarrow}(y{<}1)$. Lastly, when $f{=}[\pi]@\mathtt{s}$, $[GD\text{-}Guard]$ generates a conditional flow from the previous state to the current state, with the premise to be π holds at the previous state. It then continues to generate the Datalog program for the remainder of the trace.

4.5 Soundness Discussion

Our tool significantly improves the precision of CTL analysis compared to the state-of-the-art tool **Function**, based on abstract interpretation. While abstract interpretation tends to be overly conservative, our method reduces the "unknown" results by leveraging a refined trace representation. Crucially, our termination analysis is provably sound, as

$$\frac{fst(\Phi)=\{\} \quad \mathbb{D}=[\texttt{flow}(s_p,s_p)]}{\Pi \vdash GWRE2D(s_p,\Phi) \rightsquigarrow \mathbb{D}}\ [GD\text{-}Base] \qquad \frac{F=fst(\Phi) \quad (\forall f_i \in F).\ \Pi \vdash (f_i, s_p, \mathcal{D}_{f_i}(\Phi)) \hookrightarrow \mathbb{D}_i}{\Pi \vdash GWRE2D(s_p,\Phi) \rightsquigarrow \bigcup \mathbb{D}_i}\ [GD\text{-}Ind]$$

$$\begin{array}{c}[GD\text{-}Omega]\\ \dfrac{\begin{array}{c}\mathbb{D}_1 = tailToHeadFlows(\Phi)\\ \Pi \vdash GWRE2D(s_p,\Phi) \rightsquigarrow \mathbb{D}_2\end{array}}{\Pi \vdash (\Phi^{\omega}, s_p, \_) \hookrightarrow \mathbb{D}_1{+}{+}\mathbb{D}_2}\end{array} \qquad \begin{array}{c}[GD\text{-}Guard]\\ \dfrac{\begin{array}{c}\mathbb{D}_1=[\texttt{flow}(s_p,s) :\!- \pi(s_p)]\\ \Pi \vdash GWRE2D(s,\Phi) \rightsquigarrow \mathbb{D}_2\end{array}}{\Pi \vdash ([\pi]@\texttt{s}, s_p, \Phi) \hookrightarrow \mathbb{D}_1{+}{+}\mathbb{D}_2}\end{array} \qquad \dfrac{\begin{array}{c}\mathbb{D}_1=[\texttt{flow}(s_p,s)]\\ \mathbb{D}_2=\{\pi'(s) \mid \forall \pi' \in \Pi\ .\ \pi@\texttt{s} \Rightarrow \pi'\}\\ \Pi \vdash GWRE2D(s,\Phi) \rightsquigarrow \mathbb{D}_3\end{array}}{\Pi \vdash (\pi@\texttt{s}, s_p, \Phi) \hookrightarrow \mathbb{D}_1{+}{+}\mathbb{D}_2{+}{+}\mathbb{D}_3}\ [GD\text{-}Pure]$$

Fig. 15: Translating a Guarded ω-RE to a Datalog Program

guaranteed by the sound generation of CRFs. For general CTL analysis, however, soundness is intentionally relaxed to achieve better precision. Specifically, we over-approximate program states when summarising infinite executions using ω-regular expressions — a well-established trade-off in model checking. Empirical results demonstrate that our approach strikes a better balance between precision and performance than purely sound alternatives.

5 Program Repair

We present an approach for repairing CTL violations via SEDL. We cannot directly leverage the existing implementation of SEDL, namely Symlog, since it is limited to least fixed-point defined analyses, considering only positive Datalog programs. However, the CTL analysis involves nested least and greatest fixed points; thus, stratified negations frequently occur. In this section, we present our solution, enabling SEDL to repair CTL violations. We also separate the computation of the constraints related to symbolic constants and symbolic signs. The former is computed using an over-approximation method, while the latter is computed using ASP.

5.1 Symbolic Constants

The logical constraints related to symbolic constants involve assigning these symbolic constants to specific concrete constants, enabling the generation of the expected output facts. While a symbolic constant can represent any concrete constant, in practice, we only need to consider the concrete constants that can match the arguments in the existing facts through unification, called the *domain* of a symbolic constant. For example, given facts $a(\alpha)$ and $b(1)$, and rule `c(X):-a(X),b(X)`, α must be 1 to generate $c(1)$ through unification with $b(1)$. Thus, 1 belongs in α's domain. For each constant c in α's domain, $\alpha = c$ is a condition in the logical constraint ψ.

To estimate the domain of a symbolic constant, we first remove all the negative literals in the rules and then use Symlog's method, which computes the "depend" relations, defined in Fig. 16. Here Q^*_{pos} is the set of rules with all negative literals removed. We use $p(..., w_i, ...)$ to mean that w appears at the i-th argument in a literal p, and w can be constants, symbolic constants, or variables. Relation "$\text{depend}(p,i,c)$" says that the constant c may appear at the i-th position of the fact p during evaluation, which is designed to over-approximate the possible constants that appear at i-th argument of fact p. This over-approximation

$$\begin{array}{lr}
p(...,\alpha_i,...) \in \mathcal{E} \Rightarrow \text{depend}(p,i,n) & [D0]\\
\quad \text{where n is a placeholder for } \alpha & \\
p(...,c_i,...) \in \mathcal{E} \Rightarrow \text{depend}(p,i,c) & [D1]\\
R :\!- ..., p(...,c_i,...), ... \in Q^*_{pos} \Rightarrow \text{depend}(p,i,c) & [D2]\\
R :\!- ..., p_1(...,X_i,...), ..., p_2(...,X_j,...), ... \in Q^*_{pos}, X_i \equiv X_j & \\
\quad \Rightarrow \forall c.\, \text{depend}(p_1,i,c) \Leftrightarrow \text{depend}(p_2,j,c) & [D3]
\end{array}$$

Fig. 16: The "depend" Relation. ($\mathcal{E}$ is the symbolic EDB, $X_i \equiv X_j$ denotes that X_i and X_j are identical variables. α is a symbolic constant, c is a concrete constant, and n is the placeholder for α)

is introduced because it is impractical to compute the exact set of constants at the position after introducing symbolic constants [3].

For any fact $p(...,\alpha_i,...)$, rule $[D0]$ generates $\text{depend}(p,i,n)$, where n is the placeholder for α. Every symbolic constant has a corresponding placeholder, which is useful for 'inventing' new output facts whose arguments are unseen in the current EDB. Usually, the placeholders will be instantiated by the constants in the target facts. Similarly, rule $[D1]$ generates $\text{depend}(p,i,c)$ for all the concrete constant arguments. Rule $[D2]$ states that if there is a positive literal $p(...,c_i,...)$, then $\text{depend}(p,i,c)$ is generated. This is because the additional facts instantiated from symbolic constants may enable this occurrence. Rule $[D3]$ states that if variables X_i and X_j are identical across different literals in a rule, then any c that can appear at the position of X_i in a p_1 fact may also appear at the position of X_j in a p_2 fact. This propagation of potential constants happens regardless of whether the literals are in the head or body of the rule.

Symlog also over-approximates the positions where the instantiation of α is used for unification with the constant in a fact for generating new output facts. These positions, denoted as $\text{pos}(\alpha)$, include: (i) the position where the symbolic constant α appears, (ii) all argument positions in the same rule positions in the same rule that refer to the same variable as the one bound at α's position; (iii) all positions in other rules that transitively receive this variable through predicate calls. For a symbolic constant α, the over-approximation of its domain is defined as:

$$\text{domain}^{\sharp}(\alpha) \triangleq \{\, c \mid \text{depend}(p,i,c), (p,i) \in \text{pos}(\alpha)\,\} \quad [Dom]$$

$[Dom]$ computes the over-approximation of α's domain by taking the union of all potential constants that α is used for unification during the Datalog program evaluation.

After removing all the negative literals, the $\mathrm{domain}^\sharp(\alpha)$ computed by $[Dom]$ is an over-approximation of the domain of α in the original stratified Datalog program. A rule Q containing negative literals is more restrictive than its positive-only version Q_{pos} because Q_{pos} only requires matching positive conditions. In contrast, Q must ensure that no facts correspond to the negative literals. So, given the same input facts, at each step during the evaluation, the set of facts that can be generated from Q^*_{pos} is a superset of that from Q^*. Therefore, when other symbolic constants and symbolic signs are fixed, the set of constants that appear at each position (p, i) in the facts generated from Q^*_{pos} is a superset of that from Q^* at each step of the evaluation. The set of positions (p, i) where α is used for unification with the i-th argument in p in Q^*_{pos} is also a superset of that in Q^*. This is also because the conditions for allowing α to be propagated to a position in Q are less than that of Q_{pos}. Furthermore, since $\mathrm{domain}^\sharp(\alpha)$ over-approximates the domain of α in Q_{pos}, it does so in Q as well.

Example 2: Constraints over Symbolic Constants. To illustrate how the domains of symbolic constants are computed, we use the motivating examples shown in Fig. 4, Fig. 5, and the simplified symbolic EDB from Fig. 6. The simplified EDB is as follows:

```
Eq(α1, α2, α3).
```

The over-approximated positions of α_1 consist only of (`Eq`, 0), which means the first position of Eq. This is because the only rule involving `Eq`, i.e., `yEQ5(S) :- Eq(y,5,S).`, uses a constant in the first argument position, not a variable. Therefore, there are no shared variables, and hence no other positions. Similarly, $\mathrm{pos}(\alpha_2)$ is (`Eq`, 1). The positions of α_3 include many locations, since multiple rules transitively receive the variable S. In the first rule mentioned above, (`yEQ5`, 0) is included. Through transitive propagation, the second rule adds (`flow`, 0) and (`AFT_yEQ5`, 0), because they all share the variable S with `!y_EQ5(S)`. By applying the same logic, $\mathrm{pos}(\alpha_3)$ includes all positions except for $pos(\alpha_1)$ and $\mathrm{pos}(\alpha_2)$. Next, we compute the "depend" outputs, which indicate all possible values that may appear at each position, as shown in Fig. 16. The "depend" values at each position in $\mathrm{pos}(\alpha_3)$ are the values appearing in the `flow` facts, which essentially include all program states, i.e., 1 through 12. The "depend" value at $\mathrm{pos}(\alpha_1)$, i.e., (`Eq`, 0), is "`y`", because it is the only value that appears at this position in both facts and rules. The "depend" value at $\mathrm{pos}(\alpha_2)$, i.e., (`Eq`, 1), is "`5`", for the same reason.

5.2 Symbolic Signs

After instantiating all the symbolic constants, we next compute the Boolean values of the symbolic signs to generate the final ψ. SYMLOG converts the problem into finding a set of dependent facts, such that the expected output fact can only be inferred when their signs are positive. To find these dependent facts, SYMLOG uses delta-debugging (DD) [5]. However, this approach can lead to incorrect results for rules with negations.

Example 3: Incapacity of **SYMLOG.** In Fig. 17, assume that $R{=}a(1)$ is the target output fact, and all the following facts are associated with symbolic signs: $\{b(1), c(1), e(1), d(1)\}$. DD divides the fact set evenly and tests if the right half still can produce R. The subset $\{e(1), d(1)\}$ still can generate R, so DD further divides it to $\{d(1)\}$, which still produces R according to the second rule. Thus, DD returns $\{d(1)\}$ as a dependent fact set. Since SYMLOG needs to find all dependent fact sets, it iteratively selects one fact from the returned dependent facts and removes it from the fact set, then it searches for new dependent facts from the updated fact set, continuing this process until no new dependent facts can be found. In this example, removing $d(1)$ leaves $\{b(1), c(1), e(1)\}$, which cannot generate R, so DD concludes that $\{d(1)\}$ is the only dependent fact set. However, $\{b(1), c(1)\}$ and $\{e(1)\}$ are also dependent fact sets. If R represents a bug, removing only $d(1)$ would not be able to disable it.

To support both positive rules and rules with stratified negations, we encode these rules using Answer Set Programming (ASP) [10], a declarative programming for solving complex search problems. ASP solvers can find sets of facts that satisfy given constraints, even if the rules are non-monotonic. An ASP program includes Prolog/Datalog-style rules and facts, and allows for specifying constraints with an empty head rule. It also supports *choice* rules, enabling alternative solutions. A solution of the ASP program is referred to as an *answer set*.

ASP can be used to compute the sets of facts that enable the target output fact R. Specifically, given any concrete valuation of the symbolic constants, the procedure for computing dependent fact sets is: (i) transform the Datalog rules to ASP rules; (ii) transform the facts instantiated with the valuation to ASP facts and surround the facts with symbolic signs using the choice structure; and (iii) specify the expected output fact via R and the facts with placeholders. The answer sets of the converted ASP program correspond to the sets of facts enabling R. The union of answer sets gathered from each valuation is the complete dependent fact set for R.

Example 4: Constraints over Symbolic Signs. Continue from Example 3, taking $\{\xi_1 b(n_1), \xi_2 c(n_1)\}$ instantiated from $\mathcal{E}_0$ and the first rule in Fig. 17 as an example. Assuming the target fact is $a(1)$, the corresponding ASP program is shown in Fig. 18. The choice structure, `{}`, indicates that any of the enclosed facts can be selected for the answer set. The constraint "`:- not a(1), not a(`n_1`)`" is to prevent all $a(1)$ and $a(n_1)$ from not being generated simultaneously, i.e., at

```
a(X):-b(X),c(X),!d(X),!e(X).
a(X):-d(X).
a(X):-e(X),!c(X).
```

Fig. 17: Example for Illustrating the Incapacity of SYMLOG

```
a(X) :- b(X), c(X), not d(X), not e(X).
{b(n1); c(n1)}.
:- not a(1), not a(n1).
```

Fig. 18: An ASP Program Computing the Answer Sets ({ } is a choice structure representing any elements within it that can be included in the answer set)

least one of them should be generated. Such a constraint eliminates any answer set that satisfies the constraint. The fact $a(n_1)$ is also included in the constraint, as it is possible that the target output cannot be produced by the 'seen' constants in the given fact set, as shown in this example. The placeholders can be replaced with the constants in the target fact to generate the final dependent facts. The solution of this constraint is $\{b(n_1), c(n_1)\}$. In this example, n_1 can be replaced with 1, and $a(n_1)$'s dependent fact set $\{b(n_1), c(n_1)\}$ correspondingly becomes $\{b(1), c(1)\}$. Since $b(n_1)$ and $c(n_1)$ are selected, the truth assignments for ξ_1 and ξ_2 are both *true*. Combining the constraints related to symbolic constants, $\alpha_1{=}n_1 \wedge \alpha_2{=}n_1$, and the truth assignments for ξ_1 and ξ_2, the logical constraints for $a(1)$ is $\psi : \alpha_1{=}n_1 \wedge \alpha_2{=}n_1 \wedge n_1{=}1 \wedge \xi_1 \wedge \xi_2$. Computing ψ for $\{\xi_1\, b(n_2), \xi_2\, c(n_2)\}$ similarly, our method returns:

$$(a(1),(\alpha_1{=}n_1 \wedge \alpha_2{=}n_1 \wedge n_1{=}1 \wedge \xi_1 \wedge \xi_2) \vee (\alpha_1{=}n_2 \wedge \alpha_2{=}n_2 \wedge n_2{=}1 \wedge \xi_1 \wedge \xi_2))$$

To compute the truth assignments that prevent a given output fact from being generated, we can remove the '`not`' in front of the ASP constraint. The ASP results can directly serve our symbolic sign assignments, as the semantics of stratified Datalog coincide with the answer set semantics, where the facts cannot be inferred from existing facts are considered *false* [15].

5.3 Patch Generation

5.3.1 Atomic Templates

We introduce three atomic templates for fact modifications: (1) Fact Addition introduces facts along existing paths to satisfy the CTL property. These added facts map to inserting assignments in the source code. For this template, we inject symbolic constants only into "assignment" facts without injecting any symbolic signs into the EDB. (2) Fact Update revises current assignments on existing paths to satisfy the CTL property. These fact modifications map to removing and adding assignments in the source code. Similar to the first template, we only inject symbolic constants into "assignment" facts, but we also associate symbolic signs with the existing "assignment" facts. (3) Fact Deletion highlights symbolic paths that do not satisfy the CTL property. These modifications essentially involve inserting conditional statements that prevent the program from reaching the paths described by the deleted facts. In this template, we associate symbolic signs with facts which are generated to model the non-deterministic values of the program variables. Since the generated patches may impact the program's transition structures, CTLEXPERT must re-analyze the modified program to verify whether the CTL property is satisfied.

5.3.2 Repair Configuration

After applying an atomic template, if the CTL property is still not satisfied, CTLEXPERT can switch to a different atomic template to either repair the original program or continue to address the updated, yet still incorrect, program. There are two common strategies for proceeding: applying atomic templates in a depth-first manner (where one template is exhaustively applied before moving on to another) or in a breadth-first manner (where all templates are applied to the program before addressing the updates). When multiple patches are generated from applying an atomic template, we select the ones requiring the fewest modifications. Among the selected patches that involve inserting assignments, we further choose the option where the inserted assignments are closest to the exit points. This approach minimizes the scope affected by the patch.

6 Implementation and Evaluation

We prototype our proposal into a tool CTLEXPERT, using approximately 5K lines of OCaml (for the program analysis) and 5K lines of Python code (for the repair). In particular, we employ Z3 [9] as the SMT solver, clingo [10] as the ASP solver, and Souffle [16] as the Datalog engine. To show the effectiveness, we design the experimental evaluation to answer the following research questions (RQ): (Experiments ran on a server with an Intel® Xeon® Platinum 8468V, 504GB RAM, and 192 cores. Source code and benchmark dataset are publicly available from [17])

- **RQ1:** How effective is CTLEXPERT in verifying CTL properties for relatively small but complex programs, compared to the state-of-the-art tool FUNCTION [7]?
- **RQ2:** What is the effectiveness of CTLEXPERT in detecting bugs that can be encoded using both CTL and linear temporal logic (LTL), such as non-termination gathered from GitHub [1] and unresponsive behaviours in protocols [2], compared with ULTIMATE LTL AUTOMIZER [18]?
- **RQ3:** How effective is CTLEXPERT in repairing CTL violations identified in RQ1 and RQ2? which has not been achieved by any existing tools.

6.1 RQ1: Verifying CTL Properties

The programs listed in Table 1 and their CTL specifications were obtained from the evaluation benchmark of FUNCTION, which includes a total of 83 test cases across over 2,000 lines of code. We categorise these test cases into six groups, labelled according to the types of their CTL properties. These programs are short but challenging, as they often involve complex loops or require a more precise analysis of the target properties. The FUNCTION tends to be conservative, often leading it to return "unknown" results, resulting in an accuracy rate of 27.7%. In contrast, CTLEXPERT demonstrates advantages with improved accuracy, particularly in 56.6%. The failure cases faced by CTLEXPERT highlight our limitations when loop guards are not explicitly defined or when LRFs are inadequate to prove termination. Although both FUNCTION and CTLEXPERT struggle to obtain meaningful invariances for infinite loops, the benefits of our loop summaries become more apparent when proving properties related to termination, such as reachability and responsiveness.

6.2 RQ2: CTL Analysis on Termination/Responsiveness Properties

The first 13 programs in Table 2 are from public repositories, each associated with a GitHub commit number where developers identify and fix the bug manually. In particular, the CTL property used for programs 1-9 (drawn from a

TABLE 1: Accuracy comparison for CTL property verification. For each property type, we show the percentage of successfully verified properties, the number of files, representative examples, and total verification time.

	Property Type	#Files	LoC	Examples	FUNCTION		CTLExpert	
					Accuracy	Total Time(s)	Accuracy	Total Time(s)
1	Termination	15	402	AF(Exit())	40.0% (6/15)	0.357	66.7% (10/15)	3.082
2	Reachability	25	470	EF(resp≥5), EF(r=1)	36.0% (9/25)	0.303	68.0% (17/25)	2.423
3	Responsive	32	1,027	AG(t=0→AF(o=1))	18.8% (6/32)	3.279	50.0% (16/32)	0.937
4	Invariance	2	30	AG(AF(t=1)∧AF(t=0))	0.0% (0/2)	0.226	0.0% (0/2)	0.045
5	Until	6	193	AU(i=0)(AU(i=1)(AG(i=3)))	0.0% (0/6)	6.756	33.3% (2/6)	0.223
6	Next	3	18	AX(AX(x=0))	66.7% (2/3)	0.006	66.7% (2/3)	0.299
	Total	83	2,140		27.7% (23/83)	10.927	56.6% (47/83)	7.008

TABLE 2: Experimental results for detecting termination/responsiveness bugs.

	Program	LoC	ULTIMATE		CTLEXPERT	
			Res.	Time	Res.	Time
1 ✗	libvnc...	25	✗	2.85	✗	0.86
1 ✓	(c311535)	27	✓	3.74	✓	0.48
2 ✗	Ffmpeg	40	✗	15.25	✗	0.61
2 ✓	(a6cba06)	44	✓	40.18	✓	0.40
3 ✗	cmus	87	✗	6.90	✗	0.58
3 ✓	(d5396e4)	86	✓	33.57	✓	0.99
4 ✗	e2fsprogs	58	✗	5.95	✗	0.92
4 ✓	(caa6003)	63	✓	4.53	✓	0.84
5 ✗	csound-...	43	✗	3.65	✗	0.78
5 ✓	(7a611ab)	45	TO	-	✓	0.65
6 ✗	fontcon...	25	✗	3.86	✗	0.77
6 ✓	(fa741cd)	25	Error	-	✓	0.65
7 ✗	asterisk	22	?	12.69	?	0.20
7 ✓	(3322180)	25	?	11.33	?	0.34
8 ✗	dpdk	45	✗	3.71	✗	0.45
8 ✓	(cd64eeac)	45	✓	2.97	?	0.48
9 ✗	xorg-s...	19	✗	3.11	✗	0.58
9 ✓	(930b9a06)	20	✓	3.10	✓	0.41
10 ✗	pure-f...	42	✓	2.56	✗	0.93
10 ✓	(37ad222)	49	✓	2.29	✓	0.38
11 ✗	live555$_a$	34	✓	2.72	✗	0.51
11 ✓	(181126)	37	✓	2.84	✓	0.34
12 ✗	openssl	88	✗	4.15	✗	0.78
12 ✓	(b8d2439)	88	✓	3.81	✓	0.99
13 ✗	live555$_b$	83	✗	2.84	✗	0.60
13 ✓	(131205)	84	✓	2.39	✓	0.57
14 ✓	while_i...c	37	✓	2.95	✓	0.79
15 ✓	array-1.c	33	✓	6.98	✓	1.08
16 ✓	array-2.c	33	✓	7.16	✓	1.09
17 ✓	count_u...1.c	25	?	8.97	✓	0.97
18 ✓	count_u...2.c	24	?	9.41	✓	0.99
19 ✓	eureka_0...c	45	✓	31.28	✓	1.97
20 ✓	for_bou...c	34	✓	3.12	✓	1.24
21 ✗	for_inf...1.c	30	?	8.22	✗	0.56
22 ✗	for_inf...2.c	30	?	7.99	✗	0.63
23 ✓	insertion...1.c	36	✓	13.24	✓	1.99
24 ✓	insertion...2.c	36	✓	14.03	✓	1.97
25 ✓	invert_s...-1-2.c	40	✓	6.61	✓	1.72
26 ✓	invert_s...-2-2.c	38	✓	16.64	✓	1.26
27 ✓	matrix-1.c	35	✓	10.31	✓	0.87
28 ✓	matrix-2...c	39	?	14.39	✓	1.02
29 ✗	n.c11.c	36	?	10.72	?	0.00
30 ✓	n.c40.c	27	✓	3.14	?	0.00
31 ✗	nec11.c	35	✗	4.29	✗	0.66
32 ✓	nec20.c	47	?	12.15	✓	2.18
33 ✓	nec40.c	33	✓	3.16	?	0.00
34 ✓	string-1.c	65	✓	104.41	?	5.43
35 ✓	string-2.c	67	✓	86.73	?	5.33
36 ✓	sum01_b...c	25	✓	4.04	✓	1.03
37 ✓	sum01-2.c	23	✓	3.29	✓	1.16
38 ✓	sum03-1.c	28	?	96.76	✓	1.25
39 ✗	sum03-2.c	26	?	8.08	✗	0.64
40 ✓	sum04-1.c	22	✓	3.69	✓	0.76
41 ✓	sum04-2.c	21	✓	3.36	✓	0.85
42 ✓	termina...1.c	26	✓	3.47	✓	0.61
43 ✓	termina...2-2.c	43	✓	4.42	✓	2.40
44 ✓	termina...3-2.c	37	✓	5.64	✓	1.05
45 ✓	trex01-1.c	47	✓	8.06	?	0.00
46 ✓	trex01-2.c	56	✓	36.39	?	0.00
47 ✓	trex02-1.c	33	✓	3.30	✓	0.52
48 ✓	trex02-2.c	33	✓	3.29	✓	0.72
49 ✓	trex03-1.c	34	?	9.76	?	0.01
50 ✓	trex03-2.c	33	?	9.77	?	0.01
51 ✗	trex04.c	50	✗	3.78	✗	6.64
52 ✓	veris.c_...p.c	41	✓	3.78	✓	0.72
53 ✓	veris.c_...r.c	205	?	2.25	?	0.00
54 ✓	veris.c_...p.c	41	?	10.20	?	0.00
55 ✓	verisec_...p.c	42	✓	3.61	✓	0.61
56 ✓	verisec_...r.c	200	?	2.13	?	0.00
57 ✗	while_i...1.c	22	✗	3.10	✗	0.35
58 ✗	while_i...2.c	21	✗	3.15	✗	0.35
59 ✗	while_i...3.c	36	✗	3.16	✗	0.34

termination benchmark [1]) is $AF(Exit())$, preventing non-termination bugs. The properties used for programs 10-13 (drawn from [2]) are of the form $AG(\phi_1 \rightarrow AF(\phi_2))$, capturing unresponsive behaviours from the protocol implementation. The remaining 46 programs are from the termination benchmark of SV-COMP [19]. All the test cases are under 100 lines of code (LoC), preserving features like data structures and pointer arithmetic. Our evaluation includes both buggy (*e.g.* 1 ✗) and developer-fixed (*e.g.* 1 ✓) versions. After converting the CTL properties to LTL formulas, we compared our tool with the latest release of UltimateLTL (v0.2.4), a regular participant in SV-COMP with competitive performance. Table 3 summarizes the overall performance of the two tools. Both tools demonstrate high accuracy in bug detection, while ULTIMATE often requires longer processing time. This experiment indicates that LRFs can effectively handle commonly seen loops, and CTLEXPERT performs a more lightweight summary computation without compromising accuracy.

TABLE 3: Overall results of Ultimate and CTLEXPERT

Tool	Avg. Time (s)	Acc.(%)	LoC	#Programs
Ultimate	11.19	72.2	3,219	72 (13*2+46)
CTLEXPERT	0.93	79.2		

6.3 RQ3: Repairing CTL Property Violations

Table 4 gathers all the program instances (from Table 1 and Table 2) that violate their specified CTL properties and are sent to CTLEXPERT for repair. The **Symbols** column records the number of symbolic constants + symbolic signs, while the **Facts** column records the number of facts allowed to be removed + added. We gradually increase the number of symbols and the maximum number of facts that can be added or deleted. The **Configuration** column shows the first successful configuration that led to finding patches, and we record the total searching time till reaching such configurations. We configure CTLEXPERT to apply three atomic templates in a breadth-first manner with a depth limit of 1, *i.e.*, Table 4 records the patch result after one iteration of the repair. The templates are applied sequentially in the order: delete, update, and add. The repair stops upon finding a correct patch or exhausting all three templates.

Due to the current configuration, CTLEXPERT only finds patch (b) for Program 1 (AF_yEQ5), while the patch (a) shown in Fig. 8 can be obtained by allowing two iterations of the repair: the first iteration adds the conditional then a second iteration to add a new assignment on the updated program. Non-termination bugs are resolved within a single iteration by adding a conditional statement that provides an earlier exit. For instance, Fig. 19 illustrates the main logic of 1✗, which enters an infinite loop when $linesToRead \leq 0$. CTLEXPERT successfully provides a fix that prevents $linesToRead \leq 0$ from occurring before entering the loop. Note that such patches are more desirable which fix the non-termination bug without dropping the loops completely. Fig. 20 shows a similar example in which infinite execution occurs when the non-deterministic loop guard evaluates to true. Consequently, a similar patch is successfully generated to prevent such cases. Unresponsive bugs involve adding more function calls or assignment modifications. The program trex04.c is not successfully repaired because the patch requires removing 26 facts representing a series of variable states. Allowing the deletion of 26 facts leads to an enormous search space, prohibiting CTLEXPERT from finishing in a reasonable time (we set a 5-minute timeout).

```
void main(){ //AF(Exit())
  int lines ToRead = *;
  int h = *;
  + if ( linesToRead <= 0 ) return;
  while(h>0){
    if(linesToRead>h)
        linesToRead=h;
    h-=linesToRead;}
  return;}
```

Fig. 19: Fixing a Possible Hang Found in libvncserver [20]

On average, the time taken to solve ASP accounts for 49.6% (18.4/37.086) of the total repair time. We also keep track of the number of patches that successfully eliminate the CTL violations. More than one patch is available for non-termination bugs, as some patches exit the entire program without entering the loop. While all the patches listed are valid, those that intend to cut off the main program logic can be excluded based on the minimum change criteria. After a manual inspection of each buggy program shown in Table 4, we confirmed that at least one generated patch is "correct". Generally, a patch is considered correct if the resulting program satisfies the given CTL property. In the context of repairing termination bugs, to avoid property overfitting, a stronger correctness criterion is that the patch should prevent infinite execution from occurring and leave the terminating paths unchanged. As the first tool to achieve automated repair of CTL violations, CTLEXPERT successfully resolves 22 out of 23 bugs.

```
int main(){
  int a[5]; int len=0;
  _Bool c=*;
  + if (c == true) return;
  while(c){
    if (len==4) len=0;
    a[len]=0;
    len++; }
  return 1; }
```

Fig. 20: A Non-Terminating Program (nec11.c) from SV-COMP

7 RELATED WORK

Analyses for CTL Properties. Existing approaches for proving CTL properties either do not support CTL formulas with arbitrary nesting of universal and existential path quantifiers [21], or support existential path quantifiers indirectly by building upon the prior works for proving non-termination [22], or by considering their universal dual (T2) [6]. In particular, the latter approach is problematic since the universal dual of an existential "until" formula is non-trivial to define. FUNCTION [7] presents a CTL properties analyser via abstract interpretation. It deploys a backward analysis to propagate the weakest preconditions, which make the program satisfy the property. While being the first work to deal with a full class of CTL properties, it has several sources of the loss of precision, such as the *dual widening* [23] technique for proving the termination of loops; as well as the alternatively applied over/under approximation to deal with existential/universal quantifiers.

Loop Summarization and Conditional Termination. Loop summarization is widely used in termination analysis [24], [25], [26], primarily focusing on summarizing the terminating behaviours. Additionally, partial loop summarization has been applied in dynamic test generation [27], where the loop structure and induction variables are identified on the fly. However, little attention has been given to summarizing non-terminating program executions. The construction of our dual summaries is based on the concept of *conditional termination* presented in previous works [28], [29],

TABLE 4: Experimental results for repairing CTL bugs. Time spent by the ASP solver is separately recorded.

Program	LoC (Datalog)	Configuration			Fixed	#Patch	ASP(s)	Total(s)
		Symbols	Facts	Template				
AF_yEQ5 (Fig. 2)	115	3+0	0+1	Add	✓	1	0.979	1.593
test_until.c	101	0+3	1+0	Delete	✓	1	0.023	0.498
next.c	87	0+4	1+0	Delete	✓	1	0.023	0.472
libvncserver	118	0+6	1+0	Delete	✓	3	0.049	1.081
Ffmpeg	227	0+12	1+0	Delete	✓	4	13.113	13.335
cmus	145	0+12	1+0	Delete	✓	4	0.098	2.052
e2fsprogs	109	0+8	1+0	Delete	✓	2	0.075	1.515
csound-android	183	0+8	1+0	Delete	✓	4	0.076	1.613
fontconfig	190	0+11	1+0	Delete	✓	6	0.098	2.507
dpdk	196	0+12	1+0	Delete	✓	1	0.091	2.006
xorg-server	118	0+2	1+0	Delete	✓	2	0.026	0.605
pure-ftpd	258	0+21	1+0	Delete	✓	2	0.069	3.590
live$_a$	112	3+4	1+1	Update	✓	1	0.552	0.816
openssl	315	1+0	0+1	Add.	✓	1	1.188	2.277
live$_b$	217	1+0	0+1	Add	✓	1	0.977	1.494
for_inf...1.c	179	0+4	2+0	Delete	✓	16	0.139	0.256
for_inf...2.c	179	0+4	2+0	Delete	✓	16	0.139	0.259
nec11.c	136	0+2	1+0	Delete	✓	11	0.138	0.22
sum03-2.c	137	0+2	1+0	Delete	✓	1	0.132	0.27
trex04.c	2732	-	-	-	✗	-	TO	TO
while_i...1.c	114	0+2	1+0	Delete	✓	1	0.148	0.213
while_i...2.c	114	0+2	1+0	Delete	✓	1	0.135	0.201
while_i...3.c	161	0+2	1+0	Delete	✓	1	0.132	0.213
Total	6243						18.4	37.086

[30]. We extend this approach by computing the preconditions that lead to non-termination and only proceeding with CTL analysis when all paths yield conclusive results, which is shown practical for verifying both safety and liveness properties, effectively separating termination analysis from temporal analysis. We share an algebraic perspective with previous work [30], where ω-REs are generated to represent the paths through a program. However, their approach represents the cycles in the CFG directly into ω formulas and then focuses on a specific termination analysis through recursion on that expression. In contrast, we construct ω formulas only after proving that the cycles (conditionally) lead to non-terminating behaviours.

Logic Programming for Temporal Analysis. To enable the expressivity for CTL properties using Datalog, prior work [4] presents Datalog LITE, a new deductive query language. We borrow their encoding of the AF operator, which requires the finiteness of the input Kripke structure. This encoding also follows from the facts that, over finite structures, CTL can be embedded into transitive closure logic [31] and that transitive closure logic has the same expressive power as stratified linear Datalog programs [32], [33]. Prior work [12] encodes CTL analysis in ASP using “findall” to encode AF. However, ”findall” is a logical impurity requiring second-order logic programming, which is not supported by declarative Datalog. This makes it incompatible with SEDL-based repair solutions that operate on first-order logic formulae. In contrast, our work encodes AF using Datalog with stratified negation, enabling greatest fixpoint encoding without relying on “findall.”

Model Repair. Prior work [34] proposed a CTL model update algorithm based on primitive operations and a minimal change criterion in Kripke structure models; Subsequently, [35] present a model repair solution for bounded and deadlock-free Petri nets, which is guided by CTL specifications via two basic repair operations: modifying transitions and the truth value of atomic propositions. Both these operations can be reflected in our approach by deleting and adding/updating facts. Prior work, [36] maps an instance of the repair program, *i.e.* a Kripke structure model and a CTL property to a Boolean formula, and the satisfiability resulted from the SAT solver indicates a patch exists or not. When satisfiable, the returned model will be mapped to a patch solution to remove transitions/states. In [37], the repair problem for CTL is considered and solved using abductive reasoning. Their method generates repair suggestions based on each concrete counter-example, which needs an iterative process to address all the counterexamples. Our approach also employs a repair-verify iterative process; however, it differs from previous methods by symbolically addressing all the CTL violations and progressively constructing source-code level patches during each iteration. Additionally, unlike model repair, which focuses on models, our approach is the first to target infinite-state programs and generate source-level patches.

Prior work [38] investigated how to automatically replace the components of a system to make it satisfy a given Linear Temporal Logic (LTL) specification. Interestingly, they defined the repair process as a game between the system, which needs to be repaired, and the environment, which acts against the system to hinder the repair. They assume that any given program can be encoded as a game, i.e., a deterministic finite-state system, without exploring how to encode a possible infinite-state program into such a finite-state game. On the contrary, we provide a practical solution via loop summarisation. Another difference is that they focus on the LTL formula, which requires a doubly exponential blowup when translated into an automaton. In contrast, this paper focuses on CTL properties, which operate directly on the finite-state model using fixed-point algorithms that can be naturally encoded using Datalog. Furthermore, it employs a “memoryless strategy,” where the structure and logic of the program remain unchanged, and it avoids introducing new variables. Such a strategy can only generate patches which mutate either the left-hand side or the right-hand side of an existing assignment statement. In contrast, our patch results can be inserting, deleting assignments, or adding conditionals.

General Automated Program Repair. Automated program repair has been extensively studied, with prior work falling mainly into three main categories. Test-based approaches, including search-based techniques such as GenProg [39] and Prophet [40], and semantic-based techniques such as SemFix [41] and Angelix [42], rely on test suites to validate candidate patches but often overfit due to test incompleteness. Static analysis-based methods,systems (e.g., MemFix [43], Hippodrome [44], Phoenix [45], ProveNFix [46]) exploit static analyzers to repair specific bug classes such as memory errors, null pointer exceptions, and concurrency violations. More recently, learning-based approaches, including those leveraging large language models [47], have shown strong performance across diverse benchmarks by training on massive code corpora, but face challenges of explainability and reliability.

Our approach presents a static analysis-based repair technique for correcting violations of CTL properties. In contrast to test-based methods, which are constrained by specific test cases, our method leverages static analysis to exhaustively traverse all possible execution paths. This guarantees that generated patches are comprehensive, resolving all instances of a property violation. To our knowledge, this is the first work to specifically address the automated repair of CTL properties.

Temporal Property guided Automated Program Repair. Prior work [48] synthesizes a ”repairable” past-time Signal Temporal Logic (ptSTL) formula (representing the buggy behaviors) from simulation traces and performs the repair on discrete-time systems or timed automata. An inherent characteristic of this trace-based approach is that its repair capability is confined to the behaviors captured by the provided simulation traces. ProveNFix [46] repairs C programs against LTL specifications over single traces by inserting or deleting function calls. In contrast, CTLEXPERT operates at a finer granularity, capable of modifying statements like assignments. Moving beyond single traces, both [49] and [50] address hyperproperties, which relate multiple traces; the former operates on abstract models (finite-state Kripke structures), while the latter advances to repairing real code using symbolic execution and syntax-guided synthesis (SyGuS), but requires the user to specify fault locations. In contrast, CTLEXPERT automatically locates faults and employs a novel two-stage repair: it first derives patch semantics in Datalog and then translates them into concrete source code.

8 Threats to Validity and Limitations

Several factors may threaten the validity of this work: (1) the limited scale of the subject programs, (2) the incompleteness of the specified properties, and (3) the overfitting caused by the incomplete specified properties. While our evaluation uses focused subject programs rather than full-scale projects, they incorporate complex control structures like loops that represent significant repair challenges. Our experimental results demonstrate the effectiveness of our approach in handling this complexity. The completeness of the specified properties affects the quality of the generated patches. Our approach generates patches that satisfy the given properties, therefore a patch may subtly violate other unspecified properties, that is, it overfits the incomplete property. A natural mitigation strategy is to enrich the property set, for example by combining additional specifications from other static analyses or developer annotations, thereby reducing the risk of property-level overfitting. Nevertheless, this limitation differs from the overfitting suffered by many test-based program repair methods [51], which cover only a subset of feasible program execution paths. In contrast, our approach ensures that the specified properties hold across all execution paths.

As noted above, our approach prioritizes generating correct patches that satisfy the target properties over producing patches that are identical to a developer’s original fix. This focus on property-based correctness is intentional, as it ensures the repaired program’s behavior is aligned with its specification. Future research could focus on synthesizing patches that are not only correct but also align with human-like coding styles and practices, potentially by learning from historical fix patterns.

9 Conclusion

We demonstrate the feasibility of identifying and repairing CTL violations for infinite-state programs. We propose a method that transforms a given program into a Datalog program, enabling its repair by adjusting Datalog facts. Our technical contribution includes support for repairing both safety and liveness properties. This work advances existing Datalog-based repair techniques to encompass analyses defined by both least-fixpoint and greatest-fixpoint semantics. We have developed a prototype to illustrate our proposal and present experimental results that showcase its utility. Instead of generating counterexamples and fixing them individually, our tool provides a comprehensive find-and-fix framework for addressing CTL violations.

Acknowledgments

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant ”Automated Program Repair”, MOE-MOET32021-0001.

References

[1] X. Shi, X. Xie, Y. Li, Y. Zhang, S. Chen, and X. Li, “Large-scale analysis of non-termination bugs in real-world OSS projects,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 256–268. [Online]. Available: https://doi.org/10.1145/3540250.3549129

[2] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, “Linear-time temporal logic guided greybox fuzzing,” in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1343–1355. [Online]. Available: https://doi.org/10.1145/3510003.3510082

[3] Y. Liu, S. Mechtaev, P. Subotic, and A. Roychoudhury, "Program repair guided by datalog-defined static analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1216–1228. [Online]. Available: https://doi.org/10.1145/3611643.3616363

[4] G. Gottlob, E. Grädel, and H. Veith, "Datalog lite: A deductive query language with linear time model checking," *ACM Transactions on Computational Logic (TOCL)*, vol. 3, no. 1, pp. 42–79, 2002.

[5] A. Zeller, "Yesterday, my program worked. today, it does not. why?" *ACM SIGSOFT Software engineering notes*, vol. 24, no. 6, pp. 253–267, 1999.

[6] B. Cook, H. Khlaaf, and N. Piterman, "Faster temporal reasoning for infinite-state programs," in *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, pp. 75–82. [Online]. Available: https://doi.org/10.1109/FMCAD.2014.6987598

[7] C. Urban, S. Ueltschi, and P. Müller, "Abstract interpretation of CTL properties," in *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 11002. Springer, 2018, pp. 402–422. [Online]. Available: https://doi.org/10.1007/978-3-319-99725-4_24

[8] A. M. Ben-Amram, J. J. Doménech, and S. Genaim, "Multiphase-linear ranking functions and their relation to recurrent sets," in *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, B. E. Chang, Ed., vol. 11822. Springer, 2019, pp. 459–480. [Online]. Available: https://doi.org/10.1007/978-3-030-32304-2_22

[9] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[10] V. Lifschitz, *Answer Set Programming*. Springer, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-24658-7

[11] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[12] A. Rocca, N. Mobilia, E. Fanchon, T. Ribeiro, L. Trilling, and K. Inoue, "Asp for construction and validation of regulatory biological networks," *Logical Modeling of Biological Systems*, pp. 167–206, 2014.

[13] M. Heizmann, J. Hoenicke, and A. Podelski, "Termination analysis by learning terminating programs," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 797–813. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_53

[14] A. M. Ben-Amram and S. Genaim, "On multiphase-linear ranking functions," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10427. Springer, 2017, pp. 601–620. [Online]. Available: https://doi.org/10.1007/978-3-319-63390-9_32

[15] K. T. Tekle and Y. A. Liu, "Extended magic for negation: Efficient demand-driven evaluation of stratified datalog with precise complexity guarantees," *arXiv preprint arXiv:1909.08246*, 2019.

[16] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 196–206.

[17] Zenodo, "Benchmark and source code," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15896690

[18] D. Dietsch, M. Heizmann, V. Langenfeld, and A. Podelski, "Fairness modulo theory: A new approach to LTL software model checking," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 49–66. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_4

[19] "Competition on software verification (sv-comp)," 2025. [Online]. Available: https://sv-comp.sosy-lab.org/

[20] LibVNCClient, "Libvncclient commit c311535: fix possible infinite loop," 2018. [Online]. Available: https://github.com/Sugon-Beijing/libvncserver/commit/c3115350eb8bb635d0fdb4dbbb0d0541f38ed19c

[21] B. Cook, E. Koskinen, and M. Y. Vardi, "Temporal property verification as a program analysis task," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 333–348. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_26

[22] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu, "Proving non-termination," in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, G. C. Necula and P. Wadler, Eds. ACM, 2008, pp. 147–158. [Online]. Available: https://doi.org/10.1145/1328438.1328459

[23] N. Courant and C. Urban, "Precise widening operators for proving termination by abstract interpretation," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10205, 2017, pp. 136–152. [Online]. Available: https://doi.org/10.1007/978-3-662-54577-5_8

[24] H. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter, "Bit-precise procedure-modular termination analysis," *ACM Trans. Program. Lang. Syst.*, vol. 40, no. 1, pp. 1:1–1:38, 2018. [Online]. Available: https://doi.org/10.1145/3121136

[25] A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening, "Loop summarization and termination analysis," in *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, ser. Lecture Notes in Computer Science, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 81–95. [Online]. Available: https://doi.org/10.1007/978-3-642-19835-9_9

[26] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le, and X. Li, "Automatic loop summarization via path dependency analysis," *IEEE Trans. Software Eng.*, vol. 45, no. 6, pp. 537–557, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2788018

[27] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 23–33. [Online]. Available: https://doi.org/10.1145/2001420.2001424

[28] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv, "Proving conditional termination," in *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 328–340. [Online]. Available: https://doi.org/10.1007/978-3-540-70545-1_32

[29] C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving termination through conditional termination," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10205, 2017, pp. 99–117. [Online]. Available: https://doi.org/10.1007/978-3-662-54577-5_6

[30] S. Zhu and Z. Kincaid, "Termination analysis without the tears," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 1296–1311. [Online]. Available: https://doi.org/10.1145/3453483.3454110

[31] N. Immerman and M. Y. Vardi, "Model checking and transitive-closure logic," in *Computer Aided Verification, 9th International*

Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254. Springer, 1997, pp. 291–302. [Online]. Available: https://doi.org/10.1007/3-540-63166-6_29

[32] M. P. Consens and A. O. Mendelzon, "Low complexity aggregation in graphlog and datalog," *Theor. Comput. Sci.*, vol. 116, no. 1, pp. 95–116, 1993. [Online]. Available: https://doi.org/10.1016/0304-3975(93)90221-E

[33] E. Grädel, "On transitive closure logic," in *Computer Science Logic, 5th Workshop, CSL '91, Berne, Switzerland, October 7-11, 1991, Proceedings*, ser. Lecture Notes in Computer Science, E. Börger, G. Jäger, H. K. Büning, and M. M. Richter, Eds., vol. 626. Springer, 1991, pp. 149–163. [Online]. Available: https://doi.org/10.1007/BFb0023764

[34] Y. Ding and Y. Zhang, "CTL model update: Semantics, computations and implementation," in *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, ser. Frontiers in Artificial Intelligence and Applications, G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, Eds., vol. 141. IOS Press, 2006, pp. 362–366.

[35] U. Martínez-Araiza and E. López-Mellado, "Ctl model repair for bounded and deadlock free petri nets," *IFAC-PapersOnLine*, vol. 48, no. 7, pp. 154–160, 2015.

[36] P. C. Attie, A. Cherri, K. Dak-Al-Bab, M. Sakr, and J. Saklawi, "Model and program repair via SAT solving," in *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*. IEEE, 2015, pp. 148–157. [Online]. Available: https://doi.org/10.1109/MEMCOD.2015.7340481

[37] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing model checking in verification by AI techniques," *Artif. Intell.*, vol. 112, no. 1-2, pp. 57–104, 1999. [Online]. Available: https://doi.org/10.1016/S0004-3702(99)00039-9

[38] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 226–238. [Online]. Available: https://doi.org/10.1007/11513988_23

[39] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[40] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL. ACM, 2016, pp. 298–312.

[41] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[42] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[43] J. Lee, S. Hong, and H. Oh, "Memfix: static analysis-based repair of memory deallocation errors for c," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 95–106.

[44] A. Costea, A. Tiwari, S. Chianasta, A. Roychoudhury, and I. Sergey, "Hippodrome: Data race repair using static analysis summaries," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–33, 2023.

[45] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 613–624.

[46] Y. Song, X. Gao, W. Li, W. Chin, and A. Roychoudhury, "Provenfix: Temporal property-guided program repair," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 226–248, 2024. [Online]. Available: https://doi.org/10.1145/3643737

[47] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[48] M. Ergurtuna, B. Yalcinkaya, and E. A. Gol, "An automated system repair framework with signal temporal logic," *Acta Informatica*, vol. 59, no. 2-3, pp. 183–209, 2022. [Online]. Available: https://doi.org/10.1007/s00236-021-00403-z

[49] B. Bonakdarpour and B. Finkbeiner, "Program repair for hyperproperties," *CoRR*, vol. abs/2101.08257, 2021. [Online]. Available: https://arxiv.org/abs/2101.08257

[50] R. Beutner, T. Hsu, B. Bonakdarpour, and B. Finkbeiner, "Syntax-guided automated program repair for hyperproperties," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*, ser. Lecture Notes in Computer Science, A. Gurfinkel and V. Ganesh, Eds., vol. 14683. Springer, 2024, pp. 3–26. [Online]. Available: https://doi.org/10.1007/978-3-031-65633-0_1

[51] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 international symposium on software testing and analysis (ISSTA)*, 2015.