Incremental Program Synthesis from Event Logs

JINWOO KIM*, University of California at San Diego, USA
VICTOR NICOLET, Amazon, USA
JOEY DODDS, Amazon, USA
LORIS D'ANTONI*, University of California at San Diego, USA

One primary goal of program synthesis is to enable non-expert users to write programs by providing a specification instead of an implementation. To truly realize this goal, the specification must require no expertise and no effort to generate. We consider the problem of synthesizing automation scripts from only the logs that are automatically collected by many systems. Using our approach, users can automate tasks they usually perform manually, without having to know how to program them.

Because logs are collected automatically, the synthesis approach needs to scale to large sets of logs. We present a new algorithm to solve this task by *incrementally* extending an API-calling script with behavior exemplified by a sequence of log events, adding one sequence at a time. By minimizing the program modifications at each step, we preserve user intent and synthesize a program as general as possible. We show that our approach, implemented in a tool LogLoom, scales to synthesis tasks with more traces and more complex programs than existing techniques. LogLoom synthesizes scripts that are identical to reference solutions for 60 out of 72 benchmarks, compared to 14 for an existing symbolic approach and 39 for an LLM.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; *Search-based software engineering*; • **Computer systems organization** → *Cloud computing*.

Additional Key Words and Phrases: Incremental Program Synthesis, Path Matching

ACM Reference Format:
Jinwoo Kim, Victor Nicolet, Joey Dodds, and Loris D'Antoni. 2026. Incremental Program Synthesis from Event Logs. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 328 (October 2026), 28 pages. https://doi.org/10.1145/3839460

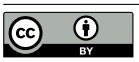
1 Introduction

In the modern era of software, users interact with cloud systems in nearly every context: through applications, development tools, and automated workflows. These interactions produce *logs* that record the sequence of cloud operations that took place. Many of these interactions are also repetitive: for example, database backups or initializing instances. Thus, it is only natural for a user to ask: can my cloud operations be automated? Traditionally, the answer requires one to write scripts in specialized automation languages (e.g. specific Python libraries, or AWS System Manager runbooks), but such an approach requires enough expertise to precisely translate workflows into code that is domain-specific for the cloud system, putting it out of reach for many end users.

In this paper, we focus on the task of automatically synthesizing automation scripts using automatically collected logs as a specification. Specifically, we use the term *log* in this paper to

*Work done while employed at Amazon Web Services.

Authors' Contact Information: Jinwoo Kim, University of California at San Diego, La Jolla, USA, jik083@ucsd.edu; Victor Nicolet, Amazon, Seattle, USA, victornl@amazon.com; Joey Dodds, Amazon, Seattle, USA, jldodds@amazon.com; Loris D'Antoni, University of California at San Diego, La Jolla, USA, lorisd@amazon.com.



ACM 2475-1421/2026/10-ART328
https://doi.org/10.1145/3839460


```
λ instanceId
let _ = ec2.StopInstances(
  InstanceIds=instanceId,
  Force=False
)
let s = ec2.DescribeInstanceStatus(
  InstanceIds=instanceId,
  IncludeAllInstances=True
)
if (s.InstanceState != "stopped") {
  let _ = ec2.StopInstances(
    InstanceIds=instanceId,
    Force=True
  )
}
```

Fig. 1. **Left**: Two simplified JSON logs automatically recorded by a cloud system: a log l_1 that a user would leave server-side when trying to stop a running instance, and a similar log l_2, where the first call to `ec2.StopInstances` failed to shut the instance down completely, and a user issues another call to `ec2.StopInstances` with the parameter `Force: true`. **Right**: A program `StopInstancesCond` that automates the user behavior expressed in logs such as those on the left.

refer to a list of events in a structured format (e.g. JSON) that contain the input, output, and name of each API called when trying to complete a certain task. Logs as specifications offer several key advantages. First and foremost, logs are generated regardless of whether a user has programming expertise— the same logs will be recorded for a given task, whether users interact through a graphical interface or program a script: using logs as specifications brings automation to a broad range of users. Second, large sets of logs can be accumulated over a period of time, leading to rich example-based specifications that would otherwise take much manual effort to create. Third, logs excel at capturing evolving workflows—as users add new steps to an existing workflow, updated logs can be used to extend automation scripts incrementally rather than requiring complete rewrites.

Figure 1 illustrates what we want our system to achieve. The left of Figure 1 contains two logs automatically recorded when a user manually performed the task of shutting down a running virtual machine (instance). In the leftmost log l_1, the user calls `ec2.StopInstances` on an instance `i-0000`, then checks its status via a call to `ec2.DescribeInstanceStatus`, which indicates that the instance has successfully shut down. In the log l_2 in the middle, the user initially performs the same two API calls, but the output of `ec2.DescribeInstanceStatus` indicates that the instance has not been fully shut down yet. The user then calls `ec2.StopInstances` once more, this time with the parameter `Force` set to `True`, to ensure the instance is stopped. The program `StopInstancesCond` on the right of Figure 1 illustrates the program we would like to synthesize given these two logs. An important feature of `StopInstancesCond` is how it *generalizes*, or captures user intent: not only can it replay the two given logs l_1 and l_2, it can also generate the desired sequence of API calls to shut down an instance given a different input `instanceId`. Synthesizing scripts that generalize well is crucial, as automation scripts must be able to handle variations in, e.g., resource names or system states, that occur naturally in real deployments.

Existing approaches cannot synthesize scripts such as `StopInstancesCond` using logs as the specification for several reasons. First, logs are an *output-only* specification for program synthesis: they show us what API calls a script should trigger, but they do not directly tell us on which *input* the script should trigger these calls. For example, log l_2 in Figure 1 tells us that `StopInstancesCond` should trigger three API calls, but does not tell us that `instanceId="i-9999"` should be the input to `StopInstancesCond` to trigger these calls. This makes it difficult to apply example-based symbolic synthesis techniques [3, 30, 47, 48], which operate over input-output example pairs.

Second, logs contain only *partial* execution information: they lack details about internal operations and control flow. For example, the log l_2 does not reveal that a script should contain an if-statement checking if `status.InstanceState!="stopped"` before the second call to `StopInstances`.

This makes Programming-By-Demonstration (PBD) techniques such as DemoMatch [50] unsuitable, as they typically require complete traces exemplifying all the operations executed by the script.

Third, logs collected in the real-world tend to be large and contain lots of information irrelevant to synthesis (e.g., randomly generated event IDs). This trait makes it challenging to use LLMs for synthesis, both in terms of high token costs, and also an increased risk of hallucinations due to the irrelevant information. Having a guarantee of correctness is essential in our scenario, where users lack the expertise to verify the synthesized script and where no automatic verification tool exists.

Because of these challenges, the only previous approach capable of synthesizing automation scripts from logs is Syren [20], which consolidates a set of logs into a program by repeatedly applying rewrite rules to the set of logs. However, even Syren has two crucial limitations. First, because Syren is based on global program rewriting, it becomes unscalable as the set of logs grows large, defeating the advantage of using large sets of logs as rich specifications. Second, Syren does not support capturing evolving workflows; Syren must generate programs from scratch whenever a new log is added, as opposed to incrementally extending an already existing automation script.

An Incremental Solution to Synthesizing Programs from Logs. In this paper, we identify an *incremental* approach to synthesizing automation scripts from a set of logs as the solution to the two biggest limitations of Syren. Precisely speaking, our approach in this paper focuses on solving the following core synthesis objective:

> *Given an automation script P that captures a set of existing logs L_{prev}, and a new log l, synthesize a new script P_{ext} such that (*i*) it captures the behavior of $L_{prev} \cup \{l\}$ and (*ii*) the difference between P and P_{ext} is syntactically minimal.*

By construction, this approach can both synthesize a completely new script from a set of logs by repeatedly adding new logs to an empty starting script, and extend an existing script with functionality exemplified by new logs. Minimizing the syntactic distance ensures our approach generates simpler programs at each step, which is important for two reasons: i) keeping programs simple is a key requirement to maintain scalability in our approach (see §4.3 for details), and ii) simpler programs better capture user intent, as they typically generalize better [21, 29].

The main technical contribution of this paper is a novel algorithm to solve the stated core synthesis objective. The key point behind our approach is that, although logs are partial specifications, a log l still contains enough sequential information—i.e., that an API call to A happened after a call to B—to allow us to try and construct *candidate execution paths* throughout P that display the same sequence of events. Even when a precisely matching candidate path does not exist, we can construct paths that exhibit a sequence of API calls with minimum edit distance from l, and then infer a minimal edit to P starting from the candidate path. Establishing such a candidate execution path considerably decreases the challenge imposed by logs being output-only specifications, because now one can also infer *candidate inputs* that trigger the desired candidate execution path.

Based on this idea of candidate paths, our approach works in a two-phase process: a first analysis phase, where we construct candidate paths and gather information about P and l (such as candidate inputs) to make the synthesis problem amenable to traditional example-based program synthesis. Then, based on the information collected in phase one, our approach generates a minimal example-based synthesis problem and passes this problem to an external synthesizer, which synthesizes the final edits that must be made to P for P to capture l. Overall, the insight behind the approach is that adding logs individually allows us to *reduce* the synthesis-from-logs problem into a simple example-based synthesis problem, which can be solved efficiently by an external synthesizer.

The overall complexity of our approach is exponential in the complexity of the final script to be generated, but *linear* in the number of logs, assuming that external queries to SMT solvers and

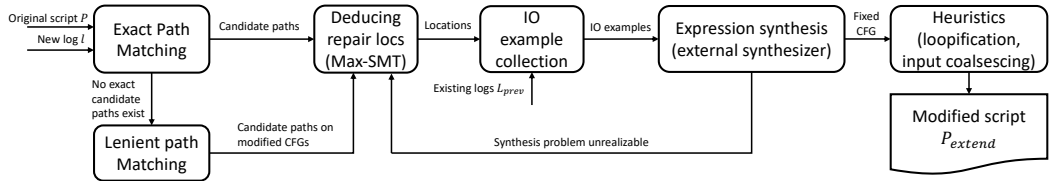


Fig. 2. A high-level system diagram that illustrates the flow of information in our tool, LogLoom. §2 provides an overview of the components; §4 discusses each component in detail.

synthesizers take constant time.[1] This linearity is the key to scalability in our problem setting, where the goal is to synthesize automation scripts (which are relatively small) from a large set of logs. In addition, our approach is *guaranteed* to generate optimal (in terms of program size) automation scripts for sets of logs that do not require a loop to automate.

We provide an implementation of our approach LogLoom and evaluate it on 72 benchmarks. While we focused on cloud API calls as our primary example in the introduction, our approach is compatible with any log-generating system, such as kernel-level system calls, which our benchmarks include. We evaluate LogLoom on the task of reconstructing human-written cloud automation scripts, such as Blink automation scripts [9] or AWS Systems Manager Runbooks [6], from the logs that are generated by their execution. Our evaluation shows that LogLoom is capable of automatically synthesizing scripts that are identical to human-written solutions for 83% of benchmarks, and fails to solve only 1 benchmark within the given time limit, showing that our incremental minimal-edit approach is indeed very effective for creating minimal programs that successfully capture user intent. LogLoom outperforms existing viable approaches to this problem (Syren [20] and LLMs) by a large margin, both in terms of the quality of the programs generated and the time consumed. Syren, the only existing symbolic approach, can only solve 14 benchmarks optimally and times out on 17 out of the 62 benchmarks we evaluate it on. Small LLMs such as Llama 3.1-8b [22] and Qwen 2.5-7B [28] fail to generate valid solutions for any of our benchmarks; even large cloud-based LLMs, such as Claude Sonnet 4.5, generate unsound solutions for 30 out of 72 benchmarks.

Contributions. To summarize, this paper makes two main contributions:

- A novel incremental algorithm for synthesizing programs from sets of logs, more scalable than previous approaches, and also allows adding functionality incrementally (§4). The approach is sound and complete, and also optimal when synthesizing loop-free scripts (§5).
- An implementation of our algorithm LogLoom. Our evaluation shows that LogLoom outperforms previous approaches by solving 1.5× as many benchmarks (§6).

§2 gives a motivating example on how LogLoom incrementally extends programs from new logs. §3 establishes the necessary preliminaries for this paper. §7 surveys related work, and §8 concludes. The full version of this paper can be found on ACM as supplementary material, and contains an appendix with details of our evaluation and proofs.

2 Motivating Example

In this section, we illustrate our overall approach using our running example (Figure 1). Recall our synthesis objective from §1. We present an example that instantiates it with the initial automation script `StopInstancesInitial` from the left of Figure 3, which captures an initial singleton set of logs $\{l_0\}$ with the events `ec2.StopInstances` and `ec2.DescribeInstanceStatus` having the argument

[1] A variety of factors may contribute to how much time a synthesizer requires to solve a specific synthesis problem, but in many cases (e.g., in CVC5 [8], which we used in our experiments), the dominant scaling factor comes from the size of the search space, which remains constant in our approach.

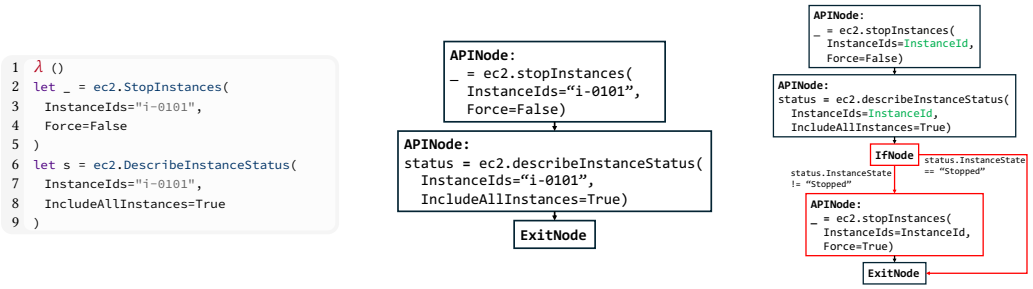


Fig. 3. **Left**: The initial program `StopInstancesInitial` for illustrating our approach in §2. **Middle**: The CFG representation of `StopInstancesInitial`. Each API call is assigned its own node. Unlabeled edges between nodes indicate transitions that are valid at all times. **Right**: The CFG representation of `StopInstancesCond`, which is the final object that LogLoom produces when given the two logs in the left of Figure 1 as input. Green expressions represent those that were synthesized when trying to repair incompatible expressions while adding the log on the top-left of Figure 1. Red nodes and edges represent those that were added to the CFG of `StopInstancesInitial` while adding the log on the bottom-left of Figure 1.

`InstanceIds = i-0101`. Our goal is to extend `StopInstancesInitial` with the new logs l_1 and l_2. Observe that `StopInstancesInitial` cannot capture the behavior of the logs l_1 and l_2 from Figure 1: all executions of `StopInstancesInitial` will generate only l_0. The user-intended solution is `StopInstancesCond` from Figure 1—which, when supplied with the correct input for `instanceId`, can generate l_0, l_1, and l_2.

Our approach operates on a control-flow graph (CFG) representation of scripts, where nodes correspond to program constructs: (*i*) API calls, (*ii*) other assignments such as expressions, (*iii*) branches, and (*iv*) loops. Edges between nodes represent transitions, and are labeled with Boolean conditions that denote when the transition is valid. For example, the CFG in the middle of Figure 3 is the CFG for `StopInstancesInitial`, which contains 2 API nodes, an exit node, and transitions between these nodes (which are always valid).

To add the target log l_1 to the CFG representation of `StopInstancesInitial`, our approach first attempts to find paths in `StopInstancesInitial` that call the exact sequence of APIs listed in l, *ignoring* their input values. For example, `StopInstancesInitial` executes the sequence of API calls ⟨`ec2.StopInstances`, `ec2.DescribeInstanceStatus`⟩, which matches exactly the sequence of API calls in the log l_1 in Figure 1. When such an exactly matching path exists, we can limit the required edits to input expressions to API calls and path conditions, *without* adding extra API calls. We then edit the problematic expressions by constructing a classic input-output synthesis problem and calling an external synthesizer, exploiting the fact that an exact path allows us to infer input examples using an SMT solver.

However, sometimes a script may not contain any path that exactly matches the API sequence in a given log l. For example, no path in `StopInstancesInitial` exactly matches the sequence in l_2 (Figure 1), as `StopInstancesInitial` only contains one call to `ec2.StopInstances`. For these cases, our approach first examines the CFG to deduce which API calls and edges need to be added to the CFG to match the log. We call this process *lenient path matching* and perform it via an extended notion of transitions throughout an automaton (§4.3). Lenient path matching identifies the required modifications on the *structure of the script*: in our example, it deduces that we need to add one more call to `ec2.StopInstances` at the end of the CFG. We then combine this with exact path matching to synthesize concrete expressions for the new API calls.

In this section, we illustrate how we attempt to construct exactly matching paths in §2.1, then proceed to show how we deduce a required *minimal* edit from a path in §2.2 and §2.3. We then illustrate how these ideas can be extended towards lenient path matching in §2.4.

2.1 Exact Path Matching Through Log Abstraction

We start by adding l_1 to `StopInstancesInitial`. Candidate paths that exactly match the sequence of API calls specified in a log can be computed by *abstracting away* the specific input arguments of API calls in both log and program, and searching for paths where only the *names* of called APIs need to match. For example, `StopInstancesInitial` cannot capture the top-left log l_1 of Figure 1 if we strictly attempt to match both API names and input arguments, but the name-only matching scheme shows that there exists a path in which the *correct APIs* are called, so hopefully one just needs to repair the arguments of the API calls for `StopInstancesInitial` to capture l_1.

Because the set of paths in a CFG forms a regular language when specific arguments are abstracted away, candidate path construction becomes an instance of *regex matching*, where one wishes to match the string of API names in the log to the CFG.

In our example, there is only one path through the CFG that accepts the string `ec2.StopInstances`, `ec2.DescribeInstanceStatus`.

2.2 Checking Log Inclusion and Deducing Repair Locations via Max-SMT Solving

Having obtained a candidate path *path*, the next step is to construct a formula $\phi_{\texttt{StopInstancesInitial},path}$ from the initial script `StopInstancesInitial` and the path *path*. The goal is to check if there exists an input x to `StopInstancesInitial`, such that (*i*) the target path *path* is exercised, and (*ii*) the program correctly supplies input arguments to the API calls in l_1.

Constructing the SMT formula is largely a one-to-one translation of `StopInstancesInitial` following *path*. This means that when constructing a formula to check if `StopInstancesInitial` can *already* replay l_1, we use the *concrete input and output values* stored in the log l_1 to model the behavior of API calls. In our example, $\phi_{\texttt{StopInstancesInitial},path}$ becomes:

$$\begin{aligned}&(\texttt{SI\_IIds} = \texttt{"i-0101"} \wedge \texttt{SI\_Force} = \texttt{False}) \wedge (\texttt{DIS\_IIds} = \texttt{"i-0101"} \wedge \texttt{DIS\_IncAll} = \texttt{True}) \wedge (\texttt{s} = \texttt{DIS\_res}) \wedge \\ &(\texttt{SI\_IIds} = \texttt{"i-0000"} \wedge \texttt{SI\_Force} = \texttt{False}) \wedge (\texttt{DIS\_IIds} = \texttt{"i-0000"} \wedge \texttt{DIS\_IncAll} = \texttt{True})) \wedge (\texttt{DIS\_res} = \{\ldots\})\end{aligned} \tag{1}$$

The first line of Equation (1) is essentially a symbolic execution of *path* in `StopInstancesInitial`: for example, the first clause `Si_IIds = "i-0101"` encodes that the API call `StopInstances` on the parameter `InstanceIds` receives as input `"i-0101"` (variable names are abbreviated for simplicity). The second line constrains the input-output values of API calls using the log l_1: for example, l_1 tells us that the call to `StopInstances` should be called with the value `"i-0000"` as `InstanceIds` (concrete values from the log are marked in blue). Thus ultimately, Equation (1) asks: "are the input-output arguments of the API calls in my script compatible with those in my log?"

In general, if such a formula is satisfiable for a script P and a log l, it means that P already captures l and thus P does not need to be modified. However, in our case, Equation (1) is unsatisfiable, meaning that `StopInstancesInitial` cannot capture l_1. There are two candidate reasons for unsatisfiability:

- The path constraints (e.g., branch conditions) to exercise the path are unsatisfiable.
- The input arguments to the API calls mismatch with the concrete values in l.

These mismatches can be fixed by replacing the offending expressions in the branch conditions or input arguments to 'correct' expressions that allow P to capture l. For example, in our scenario, the second reason is the culprit, and thus we should replace the inputs to `ec2.StopInstances` and `ec2.DescribeInstanceStatus`.

We perform this rectification in a two-step process. First, we compute the smallest set of replacements we need to perform for the constructed formula (e.g., Equation (1)) to become satisfiable using *Max-SMT solving* [42]. Second, we use program synthesis to try and synthesize actual replacements for the offending expressions (step 3). Encoding the execution of candidate paths and solving them as Max-SMT formulas is a key design choice for the scalability of our approach: Max-SMT allows

us to identify the *minimal* set of clauses in the candidate path formula that must be rectified for the path formula to become satisfiable, which in our scenario, corresponds directly to both (*i*) inferring the minimal edit required to the existing script, and (*ii*) minimizing the number of actual synthesis queries, which are expensive.

In our example, the Max-SMT solver will identify that the two clauses `SI_IIds = "i-0000"` (for `StopInstances`) and `DIS_IIds = "i-0000"` (for `ec2.DescribeInstanceStatus`) are the offending clauses, which can be mapped back to the input expression `"i-0101"` for both API calls. We observe that we do not allow the Max-SMT solver to identify clauses on the second line, e.g., `SI_IIds = "i-0000"` as offending, as these are constraints imposed by the logs that cannot be modified. Having identified these incompatible expressions, our job is now to try and synthesize replacement expressions for these locations that will allow `StopInstancesInitial` to agree with l_1.

2.3 Collecting Input-Output Examples and Synthesizing Replacements.

To synthesize replacement expressions, we take advantage of the fact that the Max-SMT solving from the previous phase also produces candidate input values for the program (e.g., x in Equation (1)). Having a candidate input allows us to construct concrete input-output examples at each point of `StopInstancesInitial` by tracing the execution of `StopInstancesInitial` on this input, which in turn allows us to use off-the-shelf example-based synthesizers for synthesizing replacement sub-expressions. In our example, solving Equation (1) does not give us any candidate inputs because `StopInstancesInitial` does not have any input parameters.

When synthesizing replacement expressions, the replacement must satisfy both the new log l_1 and any existing logs L_{prev} for `StopInstancesInitial`. In our example, there are currently 2 logs considered: l_1, and the initial log used to construct `StopInstancesInitial`.

Let us focus on how a correct expression for the parameter `InstanceIds` to `ec2.StopInstances` is synthesized. The input examples of the expression are the set of program variables that are live at the point of this expression (line 3 of `StopInstancesInitial`)—in our example, because there was no input variable, the input examples are empty sets. The output examples are the desired values of `InstanceIds` that are specified by the logs: `"i-0101"` for l_1 and `"i-0000"` for the initial log. Then the goal of the synthesis problem is to find an f such that:

$$f(\{\}) = \texttt{"i-0101"} \wedge f(\{\}) = \texttt{"i-0000"}$$

This instance has no solution: no function can return different outputs given the same input. Our approach deals with such scenarios by *adding an input parameter* to the script: we introduce a new input variable `x_1` to `StopInstancesInitial` and replace the input argument `"i-0101"` on line 3 with `x_1`.

Likewise, a synthesis problem is also constructed for the offending input argument on line 7 of `StopInstancesInitial`. The synthesis problem here is also unrealizable, so we add another input parameter `x_2` to `StopInstancesInitial` and replace the input argument on line 7 with `x_2`. The resulting modifications (highlighted in green in Figure 1) capture l_1.

Coalescing Input Variables. We observe that we added two input parameters `x_1` and `x_2` to replace program expressions, despite the fact that they evaluate to the same value in both logs. To avoid redundant parameters, we run a coalescing step where we merge newly synthesized input variables where possible. We perform coalescing by reconstructing the path formula on the modified CFG, then conjoining the path formula with the constraint that the parameters should be equal for each log considered. In our example, we check that the following two formulae are satisfiable (one for each log):

$$\exists x_1, x_2.(x_1 = \texttt{"i-0101"} \wedge \texttt{False} = \texttt{False}) \wedge (x_2 = \texttt{"i-0101"} \wedge \texttt{True} = \texttt{True}) \wedge (\texttt{status} = \{\texttt{...}\})$$

$$\exists x_1, x_2.(x_1 = \texttt{"i-0000"} \wedge \texttt{False} = \texttt{False}) \wedge (x_2 = \texttt{"i-0000"} \wedge \texttt{True} = \texttt{True}) \wedge (\texttt{status} = \{\texttt{...}\})$$

Both formulae are indeed satisfiable, meaning that `x_1` and `x_2` can be merged into a single parameter `InstanceId` for both logs. We get the modifications in green on the right of Figure 3. This wraps up the addition of l_1 to `StopInstancesInitial`.

2.4 Lenient Path Matching

We now illustrate how l_2, the log in the middle of Figure 1, is added to the extended version of `StopInstancesInitial` we finished constructing in the previous section. Recall that l_2 requires us to add an additional call to `ec2.StopInstances` at the end of `StopInstancesInitial`, which, at a high level, can be deduced by comparing the path in `StopInstancesInitial` that is closest to the sequence of APIs in l_2. §4.3 precisely describes how to perform lenient path matching; here, we focus on illustrating how to insert the new `ec2.StopInstances` call in `StopInstancesInitial`.

Inserting the new call to `ec2.StopInstances` consists of two steps: synthesizing an if-then-else statement to house the new call to `ec2.StopInstances`, then re-synthesizing input expressions to API calls that *path* exercises [4]. We must first synthesize an if-then-else statement for `ec2.StopInstances` to ensure that the modified CFG can still capture existing logs such as l_1: for example, naively appending a call to `ec2.StopInstances` after line 9 of `StopInstancesInitial` would render the resulting program unable to capture l_1. Placing `ec2.StopInstances` in a separate else-branch, as depicted in the right of Figure 3, allows the modified version of `StopInstancesInitial` to capture both l_1 and l_2 by sending l_1 to the true-branch and l_2 to the false-branch. Thus the synthesis problem becomes one of finding a branch condition that evaluates to `True` for l_0 and l_1, and `False` for l_2:

$$f(\{\texttt{x} \mapsto \texttt{"i-0101"}, \texttt{s} \mapsto \{\ldots\}\}) = \texttt{True} \wedge f(\{\texttt{x} \mapsto \texttt{"i-0000"}, \texttt{s} \mapsto \{\ldots\}\}) = \texttt{True} \wedge$$
$$f(\{\texttt{x} \mapsto \texttt{"i-9999"}, \texttt{s} \mapsto \{\ldots\}\}) = \texttt{False}$$

The synthesizer then discovers the branch condition `s.InstanceState != "Stopped"`, based on the responses of the calls to `ec2.DescribeInstanceStatus` in all three logs.

After synthesizing such a branch condition, synthesizing suitable API input expressions for the new call to `ec2.StopInstances` becomes a process identical to §2.3, where we are guaranteed to be able to construct exactly matching candidate paths to collect input-output examples for API arguments. `StopInstancesCond` depicted in the right of Figure 3 is the CFG that our approach produces through this process, where the nodes and edges highlighted in red indicate those that were added while adding l_2.

On a final note, we observe that checking for log inclusion as described in §2.2 also allows our approach to work as a *verifier*, that can efficiently check whether an externally constructed script can capture a given set of logs or not. While not the main focus of this paper, we will illustrate how this capability can be useful in itself when verifying the validity of externally generated scripts (e.g., by an LLM) in §6.

3 Preliminary Definitions

We now give some preliminary definitions required for our approach.

3.1 Core Language Definition

Figure 4 defines the core scripting language ScriptL we use for this paper, which is a variation of the language from Syren [20] and has a JavaScript-like syntax. Our approach is agnostic to the exact choice of expressions in the language; in practice, our expressions are centered over JSON objects and the lists, integers, Booleans, and strings they contain, since APIs use this format to serialize requests and responses.

ScriptL has straightforward semantics: a script takes as input primitive values (e.g., integers or strings) for variables in $\overline{V}$ and executes a sequence of statements. Semantics for individual

$$
\begin{array}{llll}
\text{Script} & P & ::= & \lambda \overline{V}.\, S \\
\text{Stmt} & S & ::= & C \mid \texttt{let}\ V = E \mid S;\ S \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{retry}\ S\ \texttt{until}(E) \mid \texttt{for}\ x\ \texttt{in}\ E\ \{S\} \\
\text{VsFn} & C & ::= & \texttt{let}\ V = \mathbb{A}(\overline{E}) \\
\text{Expr} & E & ::= & n \in \mathbb{Z} \mid s \in \texttt{string} \mid b \in \{\texttt{True}, \texttt{False}\} \mid l \in \texttt{List} \mid j \in \texttt{Json} \cdots \\
\text{Var} & V & ::= & x \mid y \mid \cdots
\end{array}
$$

Fig. 4. The definition of the core language ScriptL for this paper. The notation $\overline{X}$ denotes a vector of elements drawn from X.

constructs in ScriptL are mostly straightforward; for example, the retry loop `retry` S `until`(E) repeats the statement S until E is satisfied. During execution, we assume visible function calls and statements in ScriptL update a program state st, which is a map from program variables to values (which are concrete, primitive values from the expression domain).

Calls to visible functions `let` $V = \mathbb{A}(E)$ represent the function calls that are logged when executing a script P; for example, the call to `ec2.StopInstances` on line 2 of `StopInstancesCond` is a visible function call. Visible functions are assumed to always return a value in the sense that they may "fail" by returning an error message, but will not crash or time out and fail to produce a return value entirely. We note that these visible functions are similar to external library functions, in the sense that they are externally defined functions for which the precise implementation or semantics are unavailable. In this paper, we thus assume that all visible functions are uninterpreted functions that are also nondeterministic. For example, on the first line of Equation (1) (the path formula), observe how there are no constraints on the variable `DIS_res`, which captures the output of the call to `ec2.DescribeInstanceStatus`: in particular, `DIS_res` is unrelated to the API inputs (`DIS_iids`, `DIS_IncAll`). This is because when constructing the path formula, we do *not* have information on the precise semantics of `ec2.DescribeInstanceStatus`.

We define what it means for a script P to capture a log l under these semantics.

Definition 3.1 (Log Capture). Let l be a log $(\mathbb{A}_1, in_1, out_1), \cdots, (\mathbb{A}_k, in_k, out_k)$, where a tuple $(\mathbb{A}_i, in_i, out_i)$ indicates that a visible function named $\mathbb{A}_i$ was called with the input in_i and produced the output out_i.

We say that a script P *captures* l iff there *exists* an interpretation of all visible functions in P, and an input x, such that $P(x)$ calls $\mathbb{A}_1(in_1), \cdots, \mathbb{A}_k(in_k)$ in order during execution and each $\mathbb{A}_i(in_i)$ produces out_i as output. We say that P captures a set of logs L iff P captures every log $l \in L$.

For example, Equation (1) looks for an interpretation of the API calls `ec2.StopInstances` and `ec2.DescribeInstanceStatus` such that the input-output variables `SI_IIds`, $\cdots$, `DIS_res` in the program agree with those in the logs. In this case, there does not exist a valid interpretation because the input expressions to the API calls are wrong, which leads us to replace the inputs as in §2.3.

We observe that Theorem 3.1 takes an *angelic* [10] view of the semantics of visible functions, in contrast to the demonic view for uninterpreted functions commonly taken in program verification [5, 49]. The angelic interpretation is what we want for our task, because, for example, our goal is to generate a script that can make a series of API calls under *some* unknown cloud configuration, as opposed to verifying correctness with respect to all possible cloud configurations.

Control Flow Graphs. In this paper, we operate over the control-flow graph (CFG) representation of programs in ScriptL (e.g., the right of Figure 3). A CFG for a program P is defined as a 4-tuple $(N, E, \text{Start}, \text{Exit})$, where N denotes the nodes of the CFG, and $E \subseteq N \times N$ denotes edges between nodes, $\text{Start} \in N$ denotes the program entry node, and $\text{Exit} \in N$ denotes the program exit node. Edges inside a CFG are labeled with the boolean expression that must be satisfied in order to take the edge (e.g., the branch condition for branching edges). Nodes inside a CFG preserve all semantic information from P. Thus given a program state st and a node n, it is possible to compute the

updated program state st′ that would result from executing the original fragment of P. For simplicity, we use the phrase 'the script P' to refer both to the script itself and also its CFG representation, which, for the purposes of this paper, are equivalent objects.

We assume that blocks inside retry-loops and for-loops are scoped to their parent loops, similar to scoping for variables in programming languages. The scoping mechanism is used in lenient path matching to prevent constructing candidate paths that contain, e.g., transitions from a RetryNode scope to outside a RetryNode scope.

CFG Distance. In this paper, the only modifications that we make to a CFG are *additive* modifications, in the sense that we will add nodes and edges to a CFG, but never remove existing elements of a CFG (except for replacing expressions in inputs to visible functions or branch conditions). We define additive modifications because, as illustrated in §2.4, our approach will only add new constructs to CFGs as to preserve existing paths inside the CFG that match existing logs.

Because our modifications are additive, we can define a distance between a CFG P_1 and a CFG P_2 based on the number of components one needs to add to P_1 to obtain P_2. In this paper, we define the distance $D_{edit}(P_1, P_2)$ as a four-tuple $(d_{node}, d_{edge}, d_{exp}, d_{input})$, where d_{node} refers to the number of nodes in P_2 but not in P_1, d_{edge} refers to the number of edges in P_2 but not in P_1, d_{exp} refers to the number of modified expressions, and d_{input} refers to the number of script-input arguments in P_2 not in P_1. For example, the distance between `StopInstancesInitial` and `StopInstancesCond` from Figure 3 is $(2, 2, 2, 1)$. Distance tuples are ordered lexicographically.

3.2 Max-SMT Solving

Our approach depends on Max-SMT [42] solving to deduce minimal edits to a program.

Definition 3.2 (Max-SMT). A Max-SMT formula ϕ is defined as the pair of a set of *hard constraints* $\{H_1, \cdots, H_i\}$ and a set of *weighted soft constraints* $\{(C_1, w_1), \cdots, (C_j, w_j)\}$ where $H_1, \cdots, H_i$ and $C_1, \cdots, C_j$ are SMT clauses defined over the free variables $\overline{x}$, and $w_1, \cdots, w_n$ are positive numbers.

A *solution* to the Max-SMT formula ϕ is a model over the variables $\overline{x}$ such that all hard constraints $H_1, \cdots, H_i$ are satisfied, but soft constraints $C_1, \cdots C_j$ may be unsatisfied. An *optimal solution* to ϕ is a solution that minimizes $\sum_{C_i=\texttt{False}} w_i$, the sum of the weights of falsified soft clauses.

When writing Max-SMT formulae in this paper, we will use the construct `soft-assert` to denote that a certain clause is a soft constraint. For example, the formula $(x = 42) \wedge (\texttt{soft-assert}\, x{+}y = 37)$ denotes a Max-SMT formula where $x = 42$ is a hard constraint and $x + y = 37$ is a soft constraint. We always set the weight of a soft constraint to 1, thus solving problems where the goal is to minimize the number of falsified soft constraints. Since our encoding maps program locations to soft assertions, this amounts to solving for the minimum edits required to satisfy a path formula.

4 Incrementally Synthesizing Programs from Sets of Logs

We formalize our approach in Algorithm 1. At a high level, Algorithm 1 first tries to construct candidate paths through exact path matching (line 3) then performs a minimal edit based on the exact path (lines 4-9). If an exactly matching path is unavailable, Algorithm 1 performs lenient path matching (line 11) to construct candidate paths instead. Algorithm 1 optimizes the script (lines 16-18). We provide detailed information on the core elements below.

4.1 Line 3: Exact Path Matching

PathMatchExact from Algorithm 1 takes as input a script P and a target log l, then produces a set of candidate paths, i.e., paths through the CFG that call the same sequence of visible functions as in l. We define this concept formally using an automaton.

Definition 4.1 (Exact Path Matching). Let $P = (N, E, \text{Start}, \text{Exit})$ be a script represented as a CFG. We define the *exact match automaton* $\mathsf{A}_P^{\text{Exact}}$ as the 5-tuple $(Q_P, \Sigma, I, F, \Delta)$ where:

- $Q_P = N$, i.e., the set of states in $\mathsf{A}_P^{\text{Exact}}$ are the nodes of P,
- $\Sigma = \text{VsFns}(P)$, i.e., the alphabet of $\mathsf{A}_P^{\text{Exact}}$ is the set of visible function calls that occur in P;
- $I = \{\text{Start}\}$ and $F = \{\text{Exit}\}$, i.e., $\mathsf{A}_P^{\text{Exact}}$ starts at Start and the final node Exit is the only accepting state;
- A transition $\delta = (Q_1, \sigma, Q_2) \in \Delta$ iff $(Q_1, Q_2) \in E$ and the following extra conditions hold:
  - If Q_2 is not an VsFnNode, $\sigma = \epsilon$,
  - If Q_2 is an VsFnNode with name $\mathbb{A}$, then $\sigma = \mathbb{A}$.

A *path* through $\mathsf{A}_P^{\text{Exact}}$ is defined as an accepting run $(Q_1, \sigma_1, Q_2), (Q_2, \sigma_2, Q_3), \cdots, (Q_k, \sigma_k, Q_{n+1})$, where $Q_1 = \text{Start}$, $Q_{n+1} = \text{Exit}$ and $\sigma_1, \sigma_2, \cdots, \sigma_n$ is a string of visible function names. We use the notation $\mathsf{A}_P^{\text{Exact}}(l)$ to denote the set of all accepting runs of $\mathsf{A}_P^{\text{Exact}}$ on the input consisting of the string of visible functions logged in l.

Following Theorem 4.1, computing $\mathsf{A}_P^{\text{Exact}}(l)$ consists of finding the set of all paths that accept a string in an NFA, since $\mathsf{A}_P^{\text{Exact}}$ is a nondeterministic automaton.

We observe that in scenarios where P contains loops, the set of candidate paths *CandidatePaths* can become infinite if the loop does not contain a visible function, because we can take infinitely many empty transitions. To avoid this problem, we make the following assumptions on loops:

- **Retry-loops** make at least *one visible function call per iteration.*
- The input list for **for-loops** is *statically resolvable* while computing $\mathsf{A}_P^{\text{Exact}}(l)$. This means that the input list to a for-loop cannot depend on the input argument, as the value of the input argument is unresolvable when computing paths.

Algorithm 1 Extending a Script with a New Log

```
1:  procedure Extend(P, L_prev, l)
2:      P_ext ← None
3:      CandidatePaths ← PathMatchExact(P, l)
4:      for path in CandidatePaths do
5:          RepairLocs ← SolvePath(P, path, l)
6:          if RepairLocs is Empty then return P
7:          P' ← RepairProg(P, L_prev, l, RepairLocs)
8:          if D_edit(P, P') < D_edit(P, P_ext) then
9:              P_ext ← P'
10:     if !(P_ext is None) then goto Line 18
11:     CandidatePaths ← PathMatchLenient(P, l)
12:     for path in Sort_{D_edit}(CandidatePaths) do
13:         Edits ← SolvePathLenient(P, l, path)
14:         P' ← ExtendProg(P, L_prev, l, path, Edits)
15:         if !(P' is None) then
16:             P_ext ← Loopify(P', path, L_prev, l);
17:             break
18:     return InputConsolidate(P_ext, L_prev, l)
```

Then for retry-loops, an iteration of the loop must consume at least one character from the input log l, making the number of matches on $\mathsf{A}_P^{\text{Exact}}$ finite. For for-loops, we first statically resolve the number of iterations k, then only allow paths with exactly k iterations when computing $\mathsf{A}_P^{\text{Exact}}(l)$.

These assumptions have negligible practical impact: retry-loops are typically used to retry a certain call until it succeeds (e.g., retrying a call to `StopInstances`), while for-loops are often used to iterate over the result of an API call (e.g., iterating over the files returned by a call to `os.ls`). In practice, because input lists for for-loops can be resolved by consulting the log l, none of the benchmarks in the evaluation violate these assumptions.

4.2 Lines 5-7: Solving Paths via MAX-SMT and Generating Replacement Expressions

Following §4.1, a candidate path $path \in CandidatePaths$ is a sequence of transitions $(Q_1, \sigma_1, Q_2), \cdots, (Q_k, \sigma_k, Q_{k+1}) \in \mathsf{A}_P^{\text{Exact}}(l)$. On line 5 of Algorithm 1, SolvePath$(P, path, l)$ converts this path into a Max-SMT formula $\phi_{path}(l)$ to check its feasibility, and if infeasible, identifies program repair locations. The translation exploits the fact that Q_i and Q_{i+1} are nodes inside P, so the 'transition'

from Q_i to Q_{i+1} can be encoded as the following Max-SMT formula ϕ_i^{tr}:

$$\phi_i^{\text{tr}} \triangleq (loc_i^{edge} \implies e_i) \wedge (loc_i^{in} \implies in_i) \wedge (\text{st}_i = [\![Q_i]\!](\text{st}_{i-1})) \wedge (\texttt{assert-soft}\ loc_i^{edge}) \wedge (\texttt{assert-soft}\ loc_i^{in}) \quad (2)$$

In Equation (2), e_i denotes the edge condition between Q_i and Q_{i+1} in P, st_i denotes the program state after executing node Q_i on the state st_{i-1}, and in_i denotes an input constraint to a visible function. For example, if Q_i is a visible function `x = os.cd(dir="./")`, then in_i becomes the constraint `dir_i = "./"`. loc_i^{edge} and loc_i^{in} denote soft-asserted location variables that map the constraints e_i and in_i to actual program locations. The semantics of the entire path, denoted as the formula ϕ_{path}, becomes $\phi_{path} = \bigwedge_i \phi_i^{\text{tr}}$. We take the conjunction of ϕ_{path} and the constraints from the log l to obtain the final Max-SMT formula $\phi_{path}(l)$, which asks whether P can capture l through the path *path*.

Example 4.2 (Generating $\phi_{path}(l)$). Recall the example from §2.2, where we illustrated adding the log l_1 to `StopInstancesInitial`. We focus on the second transition in the cfg of `StopInstancesInitial` (from the node for `ec2.DescribeInstanceStatus` to Exit). The transition formula $\phi_{\texttt{DSI}}^{\text{tr}}$ here becomes:

$$\begin{aligned}&(loc_{\texttt{DSI}}^{edge} \implies \texttt{True}) \wedge (loc_{\texttt{DSI}}^{in} \implies (\texttt{DIS\_IIds = "i-0101"})) \wedge (\texttt{DIS\_IncAll = "True"})) \wedge \\ &(\texttt{status = DIS\_res}) \wedge (\texttt{soft-assert}\ loc_{\texttt{DSI}}^{edge}) \wedge (\texttt{soft-assert}\ loc_{\texttt{DSI}}^{in})\end{aligned} \quad (3)$$

The first line in Equation (3) represents the edge and input constraints, while the second line represents the context constraint and the soft assertions. Conjoining $\phi_{\texttt{DSI}}^{\text{tr}}$ with the formula for the rest of the transitions yields a version of the first line of Equation (1).

The second line of Equation (1) represents the constraints generated by the log-to-add l_1: thus conjoined, Equation (1) represents $\phi_{path}(l_1)$ constructed for `StopInstancesInitial`.

Following Theorem 3.2, a Max-SMT solver will try to find a model for $\phi_{path}(l)$ in which as many as possible loc_i^{edge} and loc_i^{in} are true (line 7 of Algorithm 1). The falsified loc_i^{edge} or loc_i^{in} denote the edge conditions or input expressions in P that must be replaced for P to accept *path*. We thus attempt to synthesize replacement expressions for these locations, by constructing input-output examples for these locations. Input examples can be created by evaluating the program state st_i for loc_i using the input parameter that the Max-SMT solver returns, as discussed in §2.[2]

For output examples , we first note that only edge constraints e_i and inputs to events in_i are associated with a selector variable loc_i, making these expressions the only candidates for replacement. Output parameters for these elements are simple: input expressions to visible functions must always result in the input value specified in the log, while path conditions in ϕ_{path} must always evaluate to `True`.

We note that limiting where expressions may be replaced does mean that in some cases, we drift away from the desideratum of replacing a minimal number of expressions. However, we allow replacements only at input expressions to visible functions and edge conditions because these are the locations where an output example for synthesis can be *uniquely determined*. For example, consider the following code fragment where `os.echo` is a visible function:

```
a = f(x); b = g(y); c = os.echo(a + b)
```

Even if a given log l specifies a unique, concrete value for `a + b` at `os.echo`, there can be a potentially infinite number of candidates for the value `b` returned by the second call to f. Thus if the expression `b = g(y)` is identified as the potential expression to replace, we cannot deduce a unique output value to use as an input-output example to a synthesizer. Our evaluation in §6 shows that even if

[2]It is possible in theory that there exist multiple input parameters that satisfy $\phi_{path}(l)$, but this almost never happens in practice as the log l contains concrete API input and output values, and thus constrains the possible inputs to one that aligns with these concrete values. In this paper we assume that there is only one possible input parameter.

we limit replacement locations to inputs and edge conditions, Algorithm 1 almost always succeeds in generating an optimal solution.

The replacement expression synthesized at this step must be correct on both the new log l and the set of existing logs L_{prev}, so we must construct input-output examples for the existing logs as well. We do this using the same process as the new logs: by running PathMatchExact on the existing logs and solving the resulting path formula. Although it may seem inefficient to have to run PathMatchExact for existing logs, in practice the runtime of Algorithm 1 is dominated by the time consumed by the call to an external synthesizer, and deducing input arguments for existing logs L_{prev} does not pose an additional bottleneck to the scalability of the system.

If the synthesizer succeeds in finding a replacement expression, we replace the corresponding expression to generate the final program P' on line 7 of Algorithm 1.

If the synthesizer fails to synthesize a replacement, there are two candidate scenarios: (*i*) the target expression is *unrealizable*, i.e., has no solution, (*ii*) a solution exists but the synthesizer failed to find the solution within the time limit. In LogLoom, we treat both scenarios as equivalent and assume that the problem is unrealizable if the synthesizer cannot synthesize a replacement (precisely identifying when a synthesis problem is unrealizable is both undecidable in theory, and very hard to solve efficiently in practice [31, 39]). As an example of unrealizability, consider:

```
λx. c = messenger.SendMessage(sender=x, receiver=x)
```

Such a script might be generated when one starts with a log that contains a call to `SendMessage` where `sender` and `receiver` are the same (i.e., a user sends a message to themselves). Then, when adding a new log where `sender` and `receiver` are different, the input-output examples for `receiver` might contain, e.g., (`foo`, `foo`) (existing) and (`foo`, `bar`) (new). This synthesis problem is *unrealizable*—there does not exist a deterministic function that satisfies the two input-output examples.

Unrealizable synthesis problems in LogLoom indicate that the current script does not have enough input parameters: in our example, the parameter `x` can capture only one of `sender` or `receiver`, and the unrealizable problem occurred when trying to derive both `sender` and `receiver` from only `x`. Thus, when the synthesizer encounters an unrealizable problem, Extend adds a new input variable x to P, and replaces the offending expression directly with x instead to generate P' (as shown in §2.1), wrapping up RepairProg.

We observe that we wish to minimize the number of input parameters added to a script: otherwise one may synthesize a degenerate script with spurious input parameters for each API argument / branch condition. This minimality constraint is why we must search for all possible exact paths *CandidatePaths* in line 3, instead of just a single path. The minimality of the edit is guaranteed by the for-loop (line 4), where we test all $path \in CandidatePaths$ to generate the minimal-edit program (line 8). *CandidatePaths* is often a singleton set and rarely contains more than two elements in practice, so the overall approach remains scalable.

4.3 Lenient Path Matching

We now move to describing PathMatchLenient on line 11 of Algorithm 1, which is invoked if P does not contain an exact match to l. PathMatchLenient can be defined as an extended form of path matching on $\mathsf{A}_P^{\mathsf{Exact}}$ when trying to accept l.

Definition 4.3 (Lenient Path Matching). Let $P = (N, E, \mathsf{Start}, \mathsf{Exit})$ be a program represented as a CFG, $\mathsf{A}_P^{\mathsf{Exact}}$ be its exact match automaton, and l a log.

We define a *lenient match* of l on $\mathsf{A}_P^{\mathsf{Exact}}$ as a series of transitions $(Q_1, \overline{\sigma}_1, Q_2), \cdots, (Q_k, \overline{\sigma}_k, Q_{k+1})$, where $\overline{\sigma}_1, \cdots, \overline{\sigma}_k$ are *strings* such that the concatenation of $\overline{\sigma}_1 \cdot \overline{\sigma}_2 \cdots \overline{\sigma}_k = l$, and the following conditions hold for each transition $\delta_i = (Q_i, \overline{\sigma}_i, Q_{i+1})$:

- Either $\overline{\sigma}_i$ = "σ" (the single character σ) and (Q_i, σ, Q_{i+1}) is a valid transition in $\mathsf{A}_P^{\mathsf{Exact}}$, *or*
- Q_{i+1} is a VsFnNode with name $\mathbb{A}$ or an Exit, and the following extra conditions hold:
  - Q_{i+1} is not an ancestor of Q_i in P,
  - Q_i and Q_{i+1} are in the same scope,
  - The input $\overline{\sigma}_i = l_{prefix}$, where l_{prefix} is a *prefix* (a string as opposed to a single character) of the remaining input string that ends with the first occurrence of $\mathbb{A}$ (if Q_{i+1} is an VsFnNode) or the full remaining input string (if Q_{i+1} is an Exit).

A lenient match of l through $\mathsf{A}_P^{\mathsf{Exact}}$ can be thought of as a match in $\mathsf{A}_P^{\mathsf{Exact}}$ that allows *extra transitions* that are not in the original automaton. These extra transitions are added only between nodes from the same scope, and only if no new loops will be created. The key idea is that an extra transition from Q_i to Q_{i+1} consumes the *sequence* of visible function calls that need to be 'skipped' in l when moving between two states Q_i and Q_{i+1} of the program; these are the visible functions we should add between Q_i and Q_{i+1} in order to modify P to capture l.

Example 4.4 (Lenient Path Matching). Consider the program Copy from Figure 5. This program creates a linear six-node CFG with a Start, an Exit, and one node for each of the calls. We illustrate how a lenient match of a new log $l_{\texttt{file}} = \langle \texttt{os.cd}, \texttt{os.ls}, \texttt{os.file}, \texttt{os.cp} \rangle$ through $\mathsf{A}_{\textsc{Copy}}^{\mathsf{Exact}}$ is performed. $l_{\texttt{file}}$ calls `os.file` instead of `os.stat` as the third command to check whether the file to be copied is a directory or not.

Start with the starting node Start. There are five extra transitions that can start from Start: one to the call to `os.cd`, one to `os.ls` one to `os.stat`, one to `os.cp`, and one directly to Exit. Of these, one is *invalid*, in the sense that it meets neither of the conditions for a transition in Theorem 4.3: the extra transition to `os.stat`, since our new log does not contain a call to `os.stat` (and instead contains one to `os.file`). The four valid extra transitions are marked in red in Figure 5. A lenient match of $l_{\texttt{file}}$ through $\mathsf{A}_{\textsc{Copy}}^{\mathsf{Exact}}$ considers the remaining four transitions in addition to the existing transition from Start to `os.cd` as the first transition of a candidate path.

```
λ x
① let _ = os.cd(dir=x)
② let files = os.ls()
③ let perm = os.stat(file=files[0])
④ let _ = os.cp(from=files[0], to="~/")
```

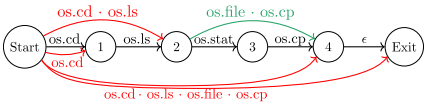


Fig. 5. An example program Copy that cds into the input x, checks the permissions of the first item in x, then copies the item into the home directory.

We note that the lenient match considers extra transitions such as those from Start to `os.cd`, even though there is already a transition from Start to `os.cd` in $\mathsf{A}_{\textsc{Copy}}^{\mathsf{Exact}}$. This is because the synthesizer may fail to synthesize an input argument for `os.cd` that unifies $l_{\texttt{file}}$ and the existing logs for Copy (either because the problem has no solution, or the synthesizer times out). In these cases, allowing the extra transition from Start to `os.cd` allows us to introduce a separate call to `os.cd` with different input arguments, and instead synthesize a condition to pick which of the two calls to trigger.

Now consider the extra transition from Start to `os.cp`. The "skipped" visible functions between Start and `os.cp` are `os.cd`, `os.ls`, and `os.file`, so the extra transition becomes: ⟨Start, `os.cd·os.ls·os.file·os.cp`, VsFnNode(`os.cp`)⟩. Taking this transition enters the node for `os.cp` with no visible function calls remaining in the input log. At this point, there is only one transition possible to Exit (note that the lenient path match will not consider transitions from, e.g., `os.cp` back to `os.cd` as such transitions will introduce new loops). Taking the final transition puts us into an accepting state with the input log fully consumed, and thus this path becomes a candidate path.

Similarly, there exist additional candidate paths generated by the lenient match that utilize different extra transitions. The path that corresponds to the minimal edit in this scenario is the one that follows $\mathsf{A}_{\textsc{Copy}}^{\mathsf{Exact}}$ until reaching `os.ls`, at which point an extra transition (marked in green in Figure 5) that skips `os.stat` and instead goes to `os.cp` while consuming `os.file` is introduced.

The extra transitions in a path *path* capture which visible functions and edges we need to add to get a script P to accept a log l. We observe that lenient paths can be partially *sorted* according to our distance metric D_{edit} (line 12 of Algorithm 1), as the extra transitions in each candidate path tell us how many extra VsFnNodes and edges must be added to a script P for the path to be realized. This helps reduce the number of calls made to the synthesizer, which is helpful as lenient path matching typically results in a larger number of candidate paths compared to exact path matching. Algorithm 1 thus tries candidate paths in order of their required edit distances and breaks the for loop immediately if a solution is found. Although breaking immediately can potentially yield scripts slightly suboptimal in the number of expressions changed, as these require a call to the SMT solver to count and cannot be pre-sorted, we find it makes negligible difference in the generated solutions during our experiments.

We note that exact path matching is a special case of lenient path matching where the algorithm is not allowed to make extra transitions. We keep exact path matching as a separate prior step so logs that do not require lenient matching, which are very common in practice, have a much more efficient path to completion before triggering the full lenient match algorithm. Lenient path matching also illustrates why maintaining the minimality of each edit is a highly desirable property in our approach: both in the sense that (*i*) minimal programs are more likely to be generalizable [21, 29], allowing more logs to be captured by exact path matching without having to fall back to lenient matching, and (*ii*) the size of the CFG directly contributes to the complexity of lenient path matching *exponentially* in the worst case, making it very compelling to maintain a minimal CFG.

4.4 Modifying the CFG

Having obtained a path *path* through a lenient match of l through $\mathsf{A}_P^{\mathsf{Exact}}$, we now illustrate how the resulting path can be used to deduce modifications to P. The overall approach is the same as for the exact path match: we construct and solve a Max-SMT formula that corresponds to the given path. We assume extra transitions in *path* are labeled, which allows us to deduce which VsFnNodes and edges we must add to P.

Creating New Visible Function Calls. Given a lenient match $path_{lnt}$, ExtendProg on line 14 of Algorithm 1 creates new CFG elements for extra transitions (Q_i, σ_i, Q_{i+1}):

- If Q_{i+1} is a VsFnNode with name $\mathbb{A}$ and $\sigma_i =$ "$\mathbb{A}$", only an extra edge is generated;
- If Q_{i+1} is a VsFnNode with name $\mathbb{A}$ and $\sigma_i = \mathbb{A}_1, \cdots, \mathbb{A}_m, \mathbb{A}$, extra VsFnNodes for $\mathbb{A}_1, \cdots, \mathbb{A}_m$ are generated and are inserted between Q_i and Q_{i+1};
- If Q_{i+1} is an Exit and $\sigma_i = \mathbb{A}_1, \cdots, \mathbb{A}_m$, extra VsFnNodes for $\mathbb{A}_1, \cdots, \mathbb{A}_m$ are generated and inserted between Q_i and Q_{i+1}.

Note that because σ_i does not record input arguments for the new events or branch conditions, these must be synthesized in a similar process to §4.2; we explain how this can be done after inserting the created events through an if-then-else statement.

Inserting Branches for New Calls. After creating the sequence of VsFnNodes for $\mathbb{A}_1, \cdots, \mathbb{A}_m$, ExtendProg connects these VsFnNodes to the original CFG for P by inserting an IfNode after Q_i. The `True` branch of the new IfNode preserves the original edge that flows outside of Q_i, while the `False` branch leads to the new sequence of events created (or directly transitions to Q_{i+1} if no new events were created). For example, in Figure 3, an IfNode was added after the node for `ec2.DescribeInstanceStatus`; the `True` edge of the IfNode points to Exit (what `ec2.DescribeInstanceStatus` originally pointed to), while the `False` edge points to the new inserted API.

ExtendProg inserts new IfNodes because all existing paths inside P should be preserved in order to maintain that P captures the set of existing logs L_{prev}. Inserting an IfNode ensures that we *add* a new path to P, as opposed to modifying an existing path in P, thus maintaining compatibility with both L_{prev} and the new log l.

Synthesizing Branch Conditions and Inputs. After the CFG is modified with the new VsFnNodes and IfNode, we run PathMatchExact on the modified CFG P' for both the new log l and the existing logs L_{prev}. Because the CFG has been modified to accept l, $\mathrm{A}^{\mathsf{Exact}}_{P'}(l)$ is guaranteed to be a singleton, and thus we can collect input-output examples for the new edges and input arguments for VsFnNodes added to P'. If the synthesizer fails to synthesize suitable branch conditions or input arguments, Algorithm 1 discards the path in question and moves on to the next lenient match. As stated, *CandidatePaths* is sorted according to D_{edit}, and Algorithm 1 breaks from the for-loop as soon as a valid solution P' is synthesized.

4.5 Loopification

In Theorem 4.3, lenient path matching does not allow extra transitions that would introduce new loops inside a CFG. This means that the resulting program P' on line 14 of Algorithm 1 cannot contain new loops, even if a new log exhibits functionality that would best be captured by a loop.

In general, deducing when certain parts of a program can be rolled into a loop is a hard problem that, to the best of our knowledge, is still an active area of research across different domains [40, 46] and has no easy solution. We introduce two simple heuristics that attempt to deduce if events from a new log l can be rolled into a new loop. Loopify on line 16 of Algorithm 1 operates on the extra transitions $(Q_i, \overline{\sigma}_i, Q_{i+1})$ in the path *path*. Loopify attempts to roll the visible function calls in $\overline{\sigma}_i$ into a loop if either of the following two conditions hold:

- If σ_i contains two or more visible function calls with the same name $\mathbb{A}$ *and* the same input arguments, Loopify attempts to roll l_{prefix} into a retry-loop,
- If there exist two or more visible function calls in $\overline{\sigma}_i$ with the same name $\mathbb{A}$, but different inputs, Loopify first creates a list l_{input} where the i-th element is equivalent to the input of the i-th call to $\mathbb{A}$. Loopify issues a synthesis query using the program state prior to entering the sequence of new events, to see if the list l_{input} can be synthesized from existing variables. If the issued synthesis query has a solution, Loopify attempts to create a new for-loop for the event sequence using l_{input} as the iteration list.

Trying to synthesize a loop body is analogous to trying to synthesize an entire program P from scratch: Loopify first splits the sequence of visible functions $\overline{\sigma}_i$ using the repeated event $\mathbb{A}$, then treats each fragment as a separate log that the loop body must be able to capture. When trying to synthesize a retry loop, the loop body is a program that only has access to the original program input variables, where the body for a for-loop is assumed to have one additional input variable that corresponds to the list element currently being iterated over.

Our two heuristics are motivated by the usage patterns of retry- and for-loops in automation scripts briefly mentioned in §3: retry-loops are often used to retry a certain operation until it succeeds, and thus are often passed the same input arguments, while for-loops are often used to iterate over a list of objects returned as the result of some other call, which is why we try to synthesize a concrete list from the input arguments to create the for loop. These two heuristics are far from complete: for example, they cannot handle nested loops. However, such complex patterns are rarely required for simple scripts synthesized from logs of events. Our experiments in §6 suggest that the given heuristics are strong enough to generate programs that successfully capture user intent in many cases. We also note that although Algorithm 1 relies on heuristics to create *new* loops, it has no problem in modifying the body of *existing* loops.

Sensitivity to the Order of Logs. The loop heuristics explained in this section cause our approach to become sensitive to the *order* in which logs are supplied. Note that without loopification, our approach will generate the same CFG given a set of logs L, regardless of the order in which the logs are incrementally added to the program. The same unfortunately does not hold when Loopify is used to modify the CFG. For example, consider a script, the intended solution, that first calls $\mathbb{A}$, then enters a retry loop that calls $\mathbb{B}$ and $\mathbb{A}$ depending on the result of the call to $\mathbb{B}$. This intended solution can generate, for example, the logs (*i*) $\mathbb{A}$-$\mathbb{B}$-$\mathbb{A}$-$\mathbb{B}$, and (*ii*) $\mathbb{A}$-$\mathbb{B}$-$\mathbb{B}$. Given this set of logs, the final CFG will differ on which log is used first in the incremental synthesis. Starting with $\mathbb{A}$-$\mathbb{B}$-$\mathbb{A}$-$\mathbb{B}$ will generate a (suboptimal) retry loop with a body that calls $\mathbb{A}$ and $\mathbb{B}$ in sequence, whereas starting with $\mathbb{A}$-$\mathbb{B}$-$\mathbb{B}$ will allow our loop heuristics to correctly guess the intended CFG, which is to call $\mathbb{A}$ and then enter a retry loop that calls $\mathbb{B}$. After falling into a suboptimal loop structure, our approach cannot 'recover' into a better structure because we consider only additive edits as briefly explained in §3 (a formal definition of additive edits may be found the Appendix). While this limitation does make LogLoom sensitive to log ordering, considering only additive edits is a deliberate design choice to keep our approach scalable; moreover, the sensitivity only arises for benchmarks that have complex loop structures, which are relatively rare for automation scripts. The evaluation in §6 explores how sensitive our implementation LogLoom is to log ordering, when trying to reconstruct real-world automation scripts.

5 Soundness, Completeness and Optimality

In this section, we provide a theoretical analysis of the properties of Algorithm 1. Full proofs for theorems in this section are available in the Appendix.

Theorem 5.1 (Soundness). *Let L_{prev} be a set of logs, l an individual log, P a program that captures L_{prev}, and P_{ext} a program. Then if $P_{ext} = \textsc{Extend}(P, L_{prev}, l)$, P_{ext} captures $L_{prev} \cup \{l\}$.*

The proof of Theorem 5.1 follows the construction of the solution P_{ext} in Algorithm 1. The key step is to observe that because the external Max-SMT solver and synthesizer are sound, given a path *path*, solutions for the path formula $\phi_{path}(l)$ (as constructed in §4.2) ultimately lead to a modified script P_{ext} capturing $L_{prev} \cup \{l\}$. We refer the reader to the Appendix for the full proof.

Theorem 5.2 (Completeness). *Let L_{prev} be a set of logs, l an individual log, and P a program that captures L_{prev}. Then $\textsc{Extend}(P, L_{prev}, l)$ always returns a valid program P'.*

The proof of Theorem 5.2 relies on the fact that lenient path matching can always, in the worst-case, generate a *trivial* solution that adds all events in a log l to a new path between Start and Exit; in other words, *CandidatePaths* in line 11 of Algorithm 1 always contains at least one candidate path, and thus P_{ext} can never be None when Algorithm 1 terminates. The formal proof can be found in the Appendix.

Theorem 5.1 and Theorem 5.2 guarantee that Algorithm 1 will always produce a solution given a script P and a new log l, but do not say anything about the optimality of that solution. We thus state and prove Theorem 5.3, which states that Algorithm 1 always generates minimal-distance modifications, with two caveats. First, our loop heuristics are incomplete, so Algorithm 1 can only generate minimal-distance modifications if the solution does not involve a new loop. Second, due to the fact that we only allow expression replacements in the inputs to visible functions and branch conditions, Algorithm 1 can only guarantee optimality over the 3-tuple distance $D_{cfg} : (d_{node}, d_{edge}, d_{input})$, instead of the full 4-tuple distance $D_{edit} : (d_{node}, d_{edge}, d_{exp}, d_{input})$ discussed in §3: the number of replaced expressions d_{exp} is dropped. For example, in Figure 3, $D_{cfg}(\texttt{StopInstancesInitial}, \texttt{StopInstancesCond}) = (2, 2, 1)$ (instead of $(2, 2, 2, 1)$ for D_{edit}). We

observe that D_{cfg} is a sufficient metric for our main motivation behind maintaining optimality—scalability of the approach as discussed at the end of §4.3—D_{cfg} still guarantees optimality over CFG structure, the main scaling factor behind lenient path matching.

Theorem 5.3 (Optimality). *Let L_{prev} be a set of logs, l an individual log, P a program that captures L_{prev}, and P_{ext} a program such that $P_{ext} = \textsc{Extend}(P, L_{prev}, l)$. Then there does not exist any P' such that P' captures $\{l\} \cup L_{prev}$ and $D_{cfg}(P, P') < D_{cfg}(P, P_{ext})$, P' can be constructed from P with only additive modifications, and P' does not contain any loops not in P.*

For exact path matching, the proof of Theorem 5.3 is straightforward: d_{node} and d_{edge} remain unchanged, while line 8 of Algorithm 1 ensures that the increase to d_{input} is minimal. The case for lenient path matching follows a similar structure: the sorted for-loop on line 12 of Algorithm 1 ensures that paths are partially sorted according to D_{cfg}; the remaining burden of proof is to show that ExtendProg (line 14) generates a P' such that there does not exist any other P'' such that $D_{cfg}(P, P'') < D_{cfg}(P, P')$, and P'' accepts $L_{prev} \cup \{l\}$. This fact can be shown by considering the nodes and edges that ExtendProg generates, which are a minimal set of additive nodes and edges that are required to accept the new log l by definition. We provide a formal proof of Theorem 5.3, along with a formal definition of the distance metric D_{cfg}, in the Appendix.

Most importantly, applying Theorem 5.3 recursively, it follows that our approach can generate *minimal* programs in terms of CFG size when synthesizing a new program from a set of logs.

Corollary 5.4 (Minimality). *Let L denote a set of logs. Starting from the empty program, our approach generates the* minimal *loop-free program in terms of CFG size that captures L.*

Theorem 5.4 guarantees minimality for loop-free generated scripts assuming the external synthesizer can always give us a solution for the queries that we issue. Following Theorem 5.4, our implementation LogLoom is capable of generating minimal solutions for the majority of the benchmarks we test, including those that contain loops, except for a few cases where the synthesizer fails to return a valid answer or the loop heuristic is imperfect.

6 Evaluation

We implement the incremental trace-guided synthesis approach in LogLoom, a tool written in Scala. LogLoom offloads solving Max-SMT problems to Z3 [16], and utilizes a custom-built synthesizer based on CVC5 for synthesizing expressions that manipulate JSON objects. We treat the synthesizer as an external component to our approach and do not claim it as part of our contribution.

In this section, we evaluate the effectiveness of our incremental approach by answering the following three research questions:

RQ1 How effective is LogLoom at generating the intended solution (hand-written automation scripts by users) from a set of logs?

RQ2 How scalable is LogLoom at generating automation scripts compared to existing approaches (Syren [20] and LLMs)?

RQ3 In cases where LogLoom fails to produce the intended solution or is slow, what are the reasons? How sensitive is LogLoom to log ordering for benchmarks that contain loops?

Benchmarks. We evaluate LogLoom on the task of reconstructing hand-written automation scripts on a suite of 72 benchmarks; 62 represent synthesizing automation scripts from scratch, and 10 represent extending existing automation scripts with new functionality. Of the 62 from-scratch benchmarks, 54 are collected from previous literature that studies similar topics, including benchmarks from Syren [20], ApiPhany [23], and real-world API calling scripts from Blink [9] and AWS Systems Manager Runbooks [6]. We extend this benchmark set by adding 8 problems where we synthesize Blink automation scripts [9] rewritten in our scripting DSL.

All benchmarks have an ideal reference solution to compare the scripts that LogLoom generates against. Reference solutions are implemented by a human and represent minimal, canonical solutions for a task, with 68 of them being collected and examined in previous work on implementing or synthesizing programs that contain API calls [6, 9, 20, 23]. We added 4 reference solutions which represent adaptations of existing automation scripts to display different behavior.

The reference solutions in our benchmarks contain up to 2 loops, 1 to 6 API calls and 2 conditions (the Appendix lists detailed information about our benchmarks). For all Syren benchmarks, we add more example logs to evaluate the scalability of our approach in the presence of many logs. This corresponds to a setting where log collection is automated and the synthesizer is given a large set of logs; our benchmarks contain 10 to 40 logs; the evaluation suite for Syren contains at most 9 logs for each benchmark.

Baselines. We compare LogLoom against two categories of baselines: Syren and various LLMs. Syren is the only existing symbolic approach that can synthesize automation scripts from logs. We compare LogLoom against Syren only in the from-scratch synthesis evaluation, as Syren does not offer functionality to extend an existing script with new logs. Syren also relies on an external synthesizer to solve synthesis queries, so we configure Syren to use the same custom-built synthesizer as LogLoom to ensure a fair comparison.

We evaluate LogLoom against different cloud-based LLM services: Claude Sonnet 4, 4.5, 4.6, and Opus 4. We note that cloud-based LLMs arguably may not be the best choice when trying to synthesize automation scripts from logs, even if they show superior performance, for two reasons: (*i*) logs are often very verbose and large, leading to high token costs, and (*ii*) more importantly, logs often contain private customer data that cloud providers are prohibited from releasing to a third party. Nevertheless, we include this comparison to better illustrate how LogLoom compares against state-of-the-art methods for synthesizing programs.

We evaluate the models under *pass@*1; i.e., the LLM must generate the correct script on the first try. We evaluate pass@1 because to the best of our knowledge, there is no existing tool that LLMs can use to check whether a candidate solution P generated by an LLM captures a given log l (although the verification problem here is tractable in theory). We use the exact path match algorithm portion of LogLoom to check whether the generated solutions are correct, but because LogLoom is our own implementation, we argue that LLMs must get the answer right on the first try in order to represent a valid replacement for LogLoom on our target use case of supplying users with automation scripts.

In addition to the comparison with LLMs under pass@1, we also include a comparison of LogLoom against an agentic approach using Claude Sonnet 4.5 as the base model. We implement an agent in Strands [7] that has access to tools for reading logs, fetching documentation and examples about our DSL, and checking whether a candidate script parses. We evaluate two agentic approaches: (*i*) one where the LLM does not have access to a verification tool, and is instead instructed to be particularly careful about correctness; and (*ii*) one where the LLM *can* call LogLoom as an external verifier to verify whether a candidate solution is correct. The latter scenario does not represent a strict comparison between modern LLMs and LogLoom, but rather serves to illustrate how LogLoom can be used in tandem with an LLM to synthesize automation scripts.

We also compared LogLoom against locally executable LLMs of smaller size (the instruct versions of Llama-3.1-8B [22], Qwen-2.5-7B [28], and Phi-3.5-mini [1]), but discovered that the smaller LLMs could solve none of our 72 benchmarks, regardless of whether they were used in an agentic setting or not. Small LLMs displayed a variety of failure modes: Phi-3.5-mini failed to produce code in most cases, while Qwen-2.5-7B and Llama-3.1-8B sometimes succeeded in generating parseable scripts, but the generated scripts failed to capture the logs. When used within an agent, both Phi-3.5-mini

Algorithm	INT	VAL	INV	TO
LogLoom	**52**	**9**	**0**	**1**
Sonnet 4	33	0	29	0
Sonnet 4.5	38	0	24	0
Sonnet 4.6	39	0	23	0
Opus 4	29	0	33	0
Agent (no verifier)	45	0	16	1
Agent (with verifier)	**53**	0	6	3
Syren	14	31	0	17

(a) From-Scratch Benchmarks

Algorithm	INT	VAL	INV	TO
LogLoom	**8**	**2**	**0**	**0**
Sonnet 4	5	0	5	0
Sonnet 4.5	4	0	6	0
Sonnet 4.6	2	0	8	0
Opus 4	5	0	5	0
Agent (no verifier)	6	0	4	0
Agent (with verifier)	**10**	0	0	0
Syren	–	–	–	–

(b) Extension Benchmarks

Fig. 6. Comparison of the number of from-scratch (left) and extension (right) benchmarks solved by LogLoom, LLMs, agents, and Syren. The agent uses Sonnet 4.5 as the base model. Results are categorized as **INT**ENDED, **VAL**ID, **INV**ALID, and **T**IME**O**UT. Results for LogLoom and the best-performing approach in each category are bolded.

and Qwen-2.5-7B failed to utilize the given tools and produced invalid outputs; Llama-3.1-8B was capable of calling tools (such as the LogLoom verifier), but ultimately failed to generate valid scripts. We conjecture that small models show especially bad behavior on these benchmarks because our specifications (large log files) lie far outside the distributions the models were originally trained on.

Because the small LLMs could not solve any of our benchmarks, the rest of this section focuses on a comparison with Syren and the larger LLMs. Specific models used, the prompt for the LLMs, and a description of the agent can be found in the Appendix.

Evaluation. We run all experiments on a MacBook Pro with 36GB of RAM and a M3Pro processor. Each benchmark is given a 300-second timeout. For LogLoom, each call to an external synthesizer is timed out at 20 seconds. Having a separate timeout on the external synthesizer allows LogLoom to search for solutions using different candidate paths if the synthesizer hangs on a problem.

We inspect the scripts synthesized by each approach on each benchmark and categorize them into one of four categories: (*i*) INTENDED, when the synthesized script is identical in CFG structure to the human-implemented reference solution;[3] (*ii*) VALID, when the synthesized script is different in CFG structure from the reference solution, but still can capture the given set of logs; (*iii*) INVALID, when the synthesized script is either syntactically invalid or fails to capture the given set of logs; and (*iv*) TIMEOUT, when the approach fails to produce a script within the given time limit. We treat a benchmark as fully solved only when it produces the INTENDED solution.

6.1 Results

Figure 6 summarizes the number of **INT**ENDED, **VAL**ID, **INV**ALID, and **T**IME**O**UT benchmarks generated by each approach.

Comparisons. LogLoom succeeds in generating INTENDED solutions for 52/62 from-scratch benchmarks, compared to only 14 for Syren and 39 at most for LLMs under pass@1 metrics. Extension benchmarks tell a similar story: LogLoom succeeds in generating INTENDED solutions for 8/10 benchmarks, compared to at most 5 for LLMs under pass@1. LogLoom outperforms the agentic approach (using Claude Sonnet 4.5 as the base model) without a verifier, which can only solve 45/62 from-scratch and 6/10 extension benchmarks. Overall, the agent with the LogLoom verifier is the *only* approach that shows comparable performance to LogLoom, generating one more INTENDED solution on the from-scratch benchmarks and two more INTENDED solutions on

[3]We do not require that expressions (arguments to APIs and branch conditions) are identical for two reasons: (*i*) , the efficiency of LogLoom depends greatly on the CFG structure of the solution due to path matching, but not on specific expressions; and (*ii*) the quality of generated expressions depends on the external synthesizer and is orthogonal to LogLoom.

```
λ input_1, input_2, input_3
let api_res_1 = user.GetInput()
let api_res_2 = slack.getConversationsList()
if (input_1 == true) {
  let api_res_3 = slack.GetConvMembers(Channel=input_2)
  for (x) in api_res_3.members {
    api_res_4 = slack.GetUserProfile(User=input_3)
    api_res_5 = output.Log(Result=api_res_4.profile)
  }
}
```

```
λ $_b, x, x_0, x_1
let api_res_9_0 = user.GetInput()
let api_res_9_1 = slack.GetConversationsList()
let s_2 = f?_20(api_res_9_0)
if s_2 {
  let api_res_9_2 = slack.GetConvMembers(Channel="C890IJKL4")
  let api_res_9_3 = ...
    output.Log(
} else {
  if ! (($_b == 8)  ($_b == 7)) {...
```

Fig. 7. VALID solutions synthesized by LogLoom (left) and Syren (right) for a sample benchmark (Literature-ApiPhanyExample.1.1). The reference solution is a nested loop that contains a branch within a for-loop. LogLoom fails to generate an optimal solution due to the incompleteness of its loop heuristics, but still manages to generate a similar solution where the branch inside the for-loop has been lifted to the outside, and the nested for-loop has been condensed into a single for-loop. In contrast, Syren generates a trivial solution in which the input variable `$_b` is used as a selector variable to select and replay a concrete input log.

the extension benchmarks. However, the LLM-based approaches are prone to generating INVALID solutions; even with access to LogLoom as a verifier, the agent ultimately generated 6 INVALID solutions for the from-scratch benchmarks. LogLoom, on the other hand, does not generate any INVALID solutions by construction and times out on only one benchmark.

In particular, LogLoom generates INTENDED solutions even for 17 out of 25 total benchmarks that contain loops, compared to 0 for Syren, and a range of 11 to 17 for the LLM under pass@1. The Appendix contains a detailed table of results, listing each benchmark, whether an approach succeeded in solving the benchmark, and the time consumed when solving, for all approaches.

In the 12 cases where LogLoom fails to generate an INTENDED solution, LogLoom generates VALID scripts for 11 benchmarks (from-scratch and extension combined) and times out on only 1 benchmark. Many VALID but non-intended scripts generated by LogLoom are due to the incompleteness of the underlying synthesizer. For example, 3 out of the 11 VALID scripts are those where the CFG structure is identical to the reference solution, but LogLoom adds an additional input argument as illustrated in §2.3 because the external synthesizer fails to synthesize a solution for a realizable sub-synthesis problem. The rest of the VALID benchmarks are due to the incompleteness of our loop heuristics: e.g., some benchmarks contain nested loops. The loop heuristics of LogLoom (§4.5) only try to generate one loop, causing LogLoom to synthesize scripts that contain single loops with a more complex body, or time out. We observe that LogLoom generates intended solutions for *all* from-scratch benchmarks except for cases where we are affected by the incompleteness of the synthesizer or our loop heuristics: this shows that the core incremental approach of LogLoom is indeed very effective at generating minimal scripts close to handwritten solutions.

In comparison, Syren times out on 17 benchmarks, and generates suboptimal solutions for 31 benchmarks for the from-scratch evaluation. In particular, we observe that many of the VALID but non-intended scripts that Syren generates are actually unreadable, trivial solutions that merely combine logs using an if-then-else statement. For example, Figure 7 illustrates a benchmark with a reference solution that contains nested loops, alongside the script generated by LogLoom and the script generated by Syren. Both LogLoom and Syren generate VALID scripts for this benchmark; however, the script generated by Syren is effectively a big if-then-else statement that simply replays concrete logs following the value of the input variable `_b`. In contrast, LogLoom generates a solution that while imperfect, still correctly captures the iterative structure and control flow hidden within the reference solution. LogLoom solved 40 out of the 48 benchmarks originally from Syren, where Syren only succeeded in solving 11. The large difference in performance is because while the benchmark *solutions* were identical, we added more specification logs; Syren scaled poorly with the number of logs and timed out on many benchmarks.

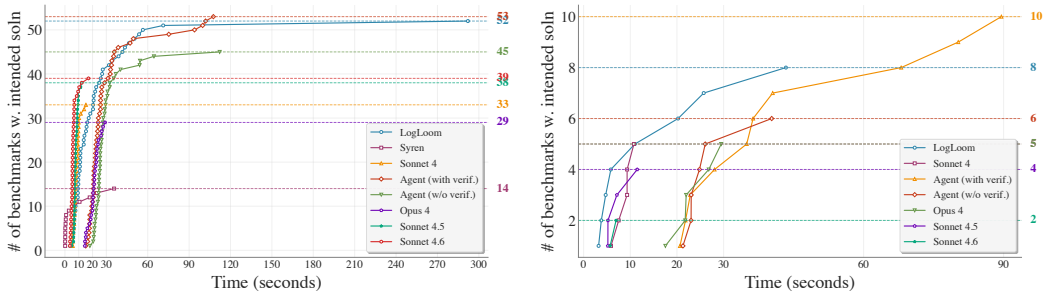


Fig. 8. Cactus plots showing the number of benchmarks solved for a given amount of time for the two experimental settings: (left) synthesis from scratch and (right) script extension with new traces. A point (t, num) in the graph indicates that num benchmarks from the suite can be solved within t seconds. For example, LogLoom can solve 52 out of the 62 from-scratch benchmarks within 280 seconds.

In general, LLMs are capable of generating scripts that mimic the high-level intent behind the logs, but sometimes fail to precisely capture the target set of logs due to semantic errors in the program logic (e.g., synthesizing wrong loop iterators or branch conditions). This phenomenon is not surprising: the high-level intent behind logs can be captured by looking only at the API names in the events, but correctly synthesizing the precise program logic requires the LLM to identify relevant values in large, noisy logs. Even the agent with access to LogLoom as a verifier ultimately generated a total of 6 INVALID solutions; the agent without access to LogLoom generated 21. As observed in the introduction, generating such invalid scripts is a major threat in using LLMs for our problem, where we lack a separate verifier capable of checking whether a script is correct.

Time Consumed for Benchmarks. We consider the time it takes for LogLoom, Syren, and the LLM to synthesize solutions for the various benchmarks. LogLoom is capable of finding INTENDED solutions for 50 of the 62 from-scratch benchmarks and 8 of the 10 extension benchmarks within 60 seconds. All benchmarks where LogLoom requires more than 60 seconds to find an INTENDED or a VALID solution have loops, with only one exception. The cactus plots in Figure 8 show how many benchmarks can be solved (with the intended solution) within a given amount of time by the three approaches, for from-scratch (the left of Figure 8) and extension (the right of Figure 8) benchmarks. We observe that Syren can solve more benchmarks than LogLoom within 10 seconds; moreover, Syren produced the solution faster than LogLoom for 9 out of the 10 benchmarks solved by both approaches. This difference is because Syren can generate optimal solutions for very simple automation scripts very fast by applying just a few rewrite rules, whereas the core path matching algorithm of LogLoom has a lower bound on the amount of time it takes, due having to construct path matches and issuing Max-SMT solver calls. However, Syren fails to scale and rapidly requires more time for complex benchmarks, eventually timing out on 17 from-scratch benchmarks. LLMs under pass@1 consume an effectively constant amount of time (around $10 \sim 30$ seconds) for all benchmarks, but as discussed, generate many invalid solutions. Agentic LLMs, on the other hand, consume time in a pattern similar to LogLoom, where they quickly solve problems for which they succeed in generating the correct solution on the first try, but begin to consume more time when the initial guess is invalid. The agent timed out on 3 benchmarks with the verifier and 1 without.

Limitations of LogLoom. The benchmarks on which LogLoom takes a long time to synthesize a solution highlight a limitation of our incremental approach: a suboptimal solution when adding one log during the process can impact future log additions. For example, the single benchmark on which LogLoom times out contains a nested loop in the reference solution. Because LogLoom cannot synthesize this nested loop, it ends up initially generating a suboptimal single loop with a very

complex branching structure. Once such a complex program has been generated, we must perform lenient path matching on the complex program for all subsequent logs; non-optimal programs also often result in many spurious queries to the synthesizer.

We observe that LogLoom is more likely to fall into such suboptimal traps when the structure of the optimal solution is complex (e.g., containing multiple loops), which is fortunately relatively rare in the realm of automation scripts. In addition, encountering suboptimal solutions does *not always* mean that LogLoom will fail to produce a solution: for example, Figure 7 illustrates a benchmark in which LogLoom created a suboptimal solution mid-generation, but still managed to synthesize a valid automation script, taking 67 seconds. LogLoom, by virtue of its optimality (Theorem 5.3), is also *guaranteed* not to generate suboptimal solutions when no new loops are created, regardless of how large the input logs or the reference solution may be.

Sensitivity to Log Ordering. Finally, we investigate how sensitive LogLoom and our loop heuristics are to the order of supplied logs by separately running the 25 loop benchmarks for 20 times with logs shuffled in random order, in addition to the initial run reported in the rest of this section. Our initial run reads logs sorted by their filenames, and we did not impose a particular order on the logs; many of our benchmarks, including the filenames of the logs, are from Syren or other existing work [23]. We count the number of times LogLoom manages to synthesize a solution (either INTENDED or VALID) across the different runs, and measure the standard deviation of consumed time as well.

Our experiment allows us to quantify the impact of varying log orderings: LogLoom displays an average standard deviation of 74.5s across the 25 benchmarks for the runs that succeeded without a TIMEOUT. There was no clear correlation between the number of loops and the standard deviation: benchmarks with 1 loop displayed standard deviations between 1.2s and 85.4s, while 2-loop benchmarks displayed deviations between 16.2s and 96.2s. Across all 21 runs, LogLoom produced 0 timeouts for 9 out of 25 benchmarks, 1 timeout for 8 benchmarks, and over 3 timeouts for the remaining 8 benchmarks. The Appendix contains statistics for each benchmark.

Interestingly, LogLoom *succeeds* in solving the benchmark that produced a TIMEOUT in the initial run for 4 out of the 21 runs, consuming a minimum of 83.7 seconds (this benchmark is still reported as a TIMEOUT in the rest of our evaluation). In 23 out of the 25 benchmarks, the mean time to solve the benchmarks was closer to the minimum than the maximum time; in 17 out of 25, the mean time is within the lower third of the min-to-max time range, suggesting that the distribution is skewed strongly towards the minimum consumed time. The run with the least amount of TIMEOUTs triggered 1 TIMEOUT, while the run with the most amount of TIMEOUTS triggered 9 TIMEOUTs across the 25 benchmarks. These results suggest that while log order is indeed important when loops need to be synthesized, LogLoom can effectively synthesize scripts for most log orders, while infrequently consuming more time for some log orders.

Summary. To summarize our findings in the evaluation, we answer our three research questions:

- **RQ1 (Efficacy)**: LogLoom is very effective in synthesizing minimal automation scripts close to reference implementations for both from-scratch and extension scenarios, generating intended solutions for 60 out of 72 total benchmarks, including 17 out of 25 for benchmarks with loops. Even if LogLoom fails to generate an intended solution, it often generates scripts that are close to optimal and generalize well.
  LogLoom outperforms all other existing solutions and, through its verification capability, also significantly boosts the performance of LLM-based agentic methods.
- **RQ2 (Scalability)**: LogLoom is much more scalable than existing symbolic approaches for synthesizing automation scripts from logs, timing out on only one benchmark compared to

17 for Syren and at most 3 for agentic LLMs. LogLoom shows comparable runtimes for most benchmarks with LLMs; unlike LLMs, LogLoom will never generate invalid solutions.
- **RQ3 (Limitations)**: LogLoom sometimes fails to generate intended solutions due to incomplete loop heuristics or the limitations of the external synthesizer. The greatest weakness of LogLoom is that generating suboptimal solutions often leads LogLoom into a trap where adding subsequent logs takes longer and longer. This weakness is further illustrated by the sensitivity of LogLoom to log ordering. Nevertheless, LogLoom can often avoid this scenario except for complex automation scripts involving several loops.

7 Discussion and Related Work

Although LogLoom specifically targets synthesizing automation scripts from cloud API logs, our technique generally applies towards synthesis problems where the specification is a set of sequences of events or actions the desired program should take. This makes our approach capable of solving synthesis problems that partially specify behavior that the target program should exhibit during execution (e.g., constraining the number of times a certain function can be called [26]).

One limitation of LogLoom is that it assumes that event logs are clean specifications, in the sense that if a certain event does not appear in the log, then the synthesized program must not trigger that event. This limitation can become problematic for scenarios where logging is not precise enough to capture all desired events. LogLoom will also struggle to synthesize programs where value dependencies between events or branch conditions are very complex, due to the fact that it relies on an external synthesizer for expression synthesis.

LogLoom also builds on a rich foundation of prior work on related problems.

Symbolic Program Synthesis. It is difficult to use existing symbolic synthesis tools (e.g., [3, 8, 30, 47]) directly to synthesize automation scripts from sets of logs. As discussed in §1, this is because logs do not record which input to the script should be used to replay the log, whereas the majority of symbolic synthesis tools are input-output based. In theory, general synthesis frameworks such as Syntax- or Semantics-guided Synthesis [3, 30] could use sets of logs as specifications by existentially quantifying over the set of possible inputs to a script, but we are unaware of any existing solver that is capable of supporting such complex specifications. The path matching algorithm and Max-SMT solving step can be seen as adapting existing synthesis tools for generating automation scripts, by inferring the extra information (input and output examples) that existing synthesizers need.

Programming-by-demonstration (PBD). The premise of our problem is similar to many problems in programming by demonstration (PBD). However, the main difference is that our demonstrations, the logs, are only partial observations of what the task needs to accomplish. In Konure [45], DemoMatch [50], PUMICE [33], and Sketch-n-Sketch [13], all non control-flow operations performed by the program to be synthesized can be observed in the traces; in contrast, our work assumes that we only have access to the input-outputs of visible functions. Another differentiating point is the requirement on interactive demonstrations: works such as WebRobot [17], Arborist [34], and Konure [45] rely on interacting with a user or an existing program (Chisel [36]). Our approach requires only a fixed set of logs. The interested reader may refer to Syren [20] for a more detailed comparison of the problem of synthesizing from partial logs compared to related work in PBD.

Program Repair. Our incremental approach has similarities with automated program repair [2, 12, 15, 27, 32, 35, 37, 38, 41]: the incremental step can be viewed as a fix for the program to accept the new log. Semantic approaches to program repair [2, 15, 37, 38, 41] also use MaxSMT solvers to identify minimal repair locations. However, those approaches do not apply in our case because of the non-deterministic, uninterpreted semantics of API calls. Verifix [2] combines MaxSMT solving

and synthesis in one inductive loop, whereas we fully discharge the synthesis problems to another input-output synthesizer. The problem of matching paths in our approach is similar to program alignment problems in repair [27] and program equivalence checking [14]. However, in our setting, we only seek to extend programs, and the alignment is one-sided (a log to a program). This allows us to use an effective formalism (§4.3) with no reliance on heuristics or program rewrites, as in [27].

Synthesizing scripts with API calls. SyPet [19], TYGAR [24], RbSyn [25] and ApiPhany [23] focus on type-guided component-based synthesis for API call sequences. These systems differ from LogLoom in two key aspects. First, they require input/output type specifications, while LogLoom synthesizes programs from event logs. Second, their main challenge is navigating large API search spaces due to growing library complexity, which TYGAR and ApiPhany address through improved API representation techniques. In contrast, LogLoom already knows which API calls to make from the traces, so API library size does not affect our synthesis complexity. Instead, we focus on synthesizing complex API interactions and data operations for non-observable trace components.

Decision Trees. Another body of related work is on decision trees, which are highly efficient if-then-else structures that map inputs to outputs, commonly learned from labeled examples by recursively generating predicates that partition the learning set into increasingly uniform subsets [11, 43]. Decision-tree learning has been used to combine expressions that are correct on different subsets of examples into a single conditional expression [4]. LogLoom similarly learns a CFG from a set of example logs, but unlike decision trees, synthesizes an entire script complete with input arguments, branch conditions, and API input expressions, that may also contain loops.

Process Mining. A highly related area of previous work is *process mining* [18, 44], which at a high level takes as input a set of logs and creates a 'process model', which, in terms of LogLoom, is essentially a CFG where arguments to events (API calls) and conditions on edges are abstracted away. Thus, process mining does not require synthesizing specific expressions when incrementally extending an existing process model, and can use search algorithms such as A* to deduce a *single* minimum edit. LogLoom, on the other hand, maintains multiple candidate edits in order to satisfy our optimality desideratum (carrying only a single min-edit may result in paths with unrealizable expression synthesis problems, which would violate our optimality property (Theorem 5.3)). One can understand LogLoom as a generalization of process models that creates precise programs instead of abstract process models.

One motivation of process mining is to summarize the information hidden inside a large set of logs as an analyzable artifact. LogLoom can be used for the same purpose; e.g., one could use LogLoom to materialize where users often make mistakes to improve an inefficient user interface.

8 Conclusion

This paper develops a novel, incremental approach to synthesizing automation scripts from sets of logs that (*i*) is more scalable than existing approaches, (*ii*) also allows one to extend existing scripts with new behavior, and (*iii*) optimal, in terms of script size, loop-free scripts.

One interesting extension of LogLoom is as a *repair tool* for scripts generated by a different entity. For example, our evaluation shows that LLMs are fast but generate many unsound solutions: then, could we use the ideas in LogLoom to *repair* the initially unsound solution produced by the LLM? This question leads to a promising line of future work that marries neural and symbolic approaches to synthesis, where we use a symbolic tool such as LogLoom to repair initially faulty solutions produced by an LLM, to achieve both scalability and correctness.

Acknowledgements

This work was partially done during an internship at Amazon. Jinwoo Kim is supported, in part, by NSF under grant 2211968 and a grant from the Korea Foundation for Advanced Studies. Opinions, findings, or conclusions expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

Data Availability Statement

Our artifact and benchmarks, which are currently undergoing internal review for release, will be released on Github under a repository of the same name (LogLoom).

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[2] Umair Z Ahmed, Zhiyu Fan, Jooyong Yi, Omar I Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified repair of programming assignments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–31. doi:10.1145/3510418

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, Portland, OR, USA, 1–8. doi:10.1109/FMCAD.2013.6679385

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 319–336. doi:10.1007/978-3-662-54577-5_18

[5] Andrew W Appel. 2011. Verified Software Toolchain: (Invited Talk). In *European Symposium on Programming*. Springer, 1–17. doi:10.1007/978-3-642-19718-5_1

[6] AWS. 2023. AWS automation runbooks reference. https://docs.aws.amazon.com/systems-manager-automation-runbooks/latest/userguide/automation-runbook-reference.html.

[7] AWS. 2026. Strands Agents. https://strandsagents.com/.

[8] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Munich, Germany, 415–442. doi:10.1007/978-3-030-99524-9_24

[9] Blink. 2023. Blink | The security automation copilot. https://www.blinkops.com/.

[10] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. 339–352. doi:10.1145/1706299.1706339

[11] Leo Breiman, Jerome Friedman, Richard A Olshen, and Charles J Stone. 2017. *Classification and regression trees*. Chapman and Hall/CRC. doi:10.1201/9781315139470

[12] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 121–130. doi:10.1145/1985793.1985811

[13] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Santa Barbara, CA, USA, 341–354. doi:10.1145/2908080.2908103

[14] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1027–1040. doi:10.1145/3314221.3314596

[15] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)*. Springer, 383–401. doi:10.1007/978-3-319-41540-6_21

[16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 4963)*. Springer, Budapest, Hungary, 337–340. doi:10.1007/978-3-540-78800-3_24

[17] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: Web robotic process automation using interactive programming-by-demonstration. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, San Diego, CA, USA, 152–167. doi:10.1145/3519939.3523711

[18] Dirk Fahland and Wil MP Van Der Aalst. 2015. Model repair—aligning process models to reality. *Information Systems* 47 (2015), 220–243. doi:10.1016/j.is.2013.12.007

[19] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Paris, France, 599–612. doi:10.1145/3009837.3009851

[20] Margarida Ferreira, Victor Nicolet, Joey Dodds, and Daniel Kroening. 2025. Program synthesis from partial traces. *Proc. ACM Program. Lang.* 9, PLDI, Article 213 (June 2025), 24 pages. doi:10.1145/3729316

[21] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 229–239. doi:10.1145/2813885.2737977

[22] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024). doi:10.48550/arXiv.2407.21783

[23] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. 2022. Type-directed program synthesis for RESTful APIs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, San Diego, CA, USA, 122–136. doi:10.1145/3519939.3523450

[24] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, New Orleans, LA, United States, 12:1–12:28. doi:10.1145/3371080

[25] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: Type- and effect-guided program synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Virtual, Canada, 344–358. doi:10.1145/3453483.3454048

[26] Qinheping Hu and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification*. Springer, 386–403. doi:10.1007/978-3-319-96145-3_21

[27] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based program repair applied to programming assignments. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 388–398. doi:10.1109/ASE.2019.00044

[28] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[29] Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–28. doi:10.1145/3485544

[30] Keith JC Johnson, Andrew Reynolds, Thomas Reps, and Loris D'Antoni. 2024. The SemGuS toolkit. In *International Conference on Computer Aided Verification (CAV)*. Springer, 27–40. doi:10.1007/978-3-031-65633-0_2

[31] Jinwoo Kim, Loris D'Antoni, and Thomas Reps. 2023. Unrealizability logic. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 659–688. doi:10.1145/3571216

[32] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 593–604. doi:10.1145/3106237.3106309

[33] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. ACM, New Orleans, LA, USA, 577–589.

doi:10.1145/3332165.3347899

[34] Xiang Li, Xiangyu Zhou, Rui Dong, Yihong Zhang, and Xinyu Wang. 2024. Efficient bottom-up synthesis for programs with local variables. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, London, UK, 1540–1568. doi:10.1145/3632894

[35] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 298–312. doi:10.1145/2914770.2837617

[36] Benjamin Mariano, Ziteng Wang, Shankara Pailoor, Christian Collberg, and Işil Dillig. 2024. Control-flow deobfuscation using trace-informed compositional program synthesis. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Pasadena, CA, USA, 2211–2241. doi:10.1145/3689789

[37] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *IEEE International Conference on Software Engineering (ICSE)*, Vol. 1. IEEE, 448–458. doi:10.1109/ICSE.2015.63

[38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 691–701. doi:10.1145/2884781.2884807

[39] Shaan Nagy, Jinwoo Kim, Thomas Reps, and Loris D'Antoni. 2024. Automating unrealizability logic: Hoare-style proof synthesis for infinite sets of programs. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 113–139. doi:10.1145/3689715

[40] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–44. doi:10.1145/3385412.3386012

[41] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*. IEEE, 772–781. doi:10.1109/ICSE.2013.6606623

[42] Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing (SAT)*, Armin Biere and Carla P. Gomes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–169. doi:10.1007/11814948_18

[43] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106. doi:10.1023/A:1022643204877

[44] Daniel Schuster, Sebastiaan J van Zelst, and Wil MP van der Aalst. 2020. Incremental discovery of hierarchical process models. In *International Conference on Research Challenges in Information Science*. Springer, 417–433. doi:10.1007/978-3-030-50316-1_25

[45] Jiasi Shen and Martin C. Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Phoenix, AZ, USA, 269–285. doi:10.1145/3314221.3314591

[46] Zachary D Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop rerolling for hardware decompilation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 420–442. doi:10.1145/3591237

[47] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 475–495. doi:10.1007/s10009-012-0249-7

[48] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Edinburgh, UK, 530–541. doi:10.1145/2594291.2594340

[49] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32. doi:10.1145/3371119

[50] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. 2017. DemoMatch: API discovery from demonstrations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Barcelona, Spain, 64–78. doi:10.1145/3062341.3062386

Received 2026-03-17; accepted 2026-08-06