Trace-Guided Synthesis of Effectful Test Generators

ZHE ZHOU, Purdue University, USA
ANKUSH DESAI, Snowflake, USA
BENJAMIN DELAWARE, Purdue University, USA
SURESH JAGANNATHAN, Purdue University, USA

Several recently proposed program logics have incorporated notions of *underapproximation* into their design, enabling them to reason about reachability rather than safety. In this paper, we explore how similar ideas can be integrated into an expressive type and effect system. We use the resulting underapproximate type specifications to guide the synthesis of test generators that probe the behavior of effectful black-box systems. A key novelty of our type language is its ability to capture underapproximate behaviors of *effectful* operations using symbolic traces that expose latent data and control dependencies, constraints that must be preserved by the test sequences the generator outputs. We implement this approach in a tool called Clouseau, and evaluate it on a diverse range of applications by integrating Clouseau's synthesized generators into property-based testing frameworks like QCheck and model-checking tools like P. In both settings, the generators synthesized by Clouseau are significantly more effective than the default testing strategy, and are competitive with state-of-the-art, handwritten solutions.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: property-based testing, synthesis, type systems, underapproximation

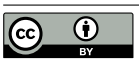
ACM Reference Format:
Zhe Zhou, Ankush Desai, Benjamin Delaware, and Suresh Jagannathan. 2026. Trace-Guided Synthesis of Effectful Test Generators. *Proc. ACM Program. Lang.* 10, PLDI, Article 186 (June 2026), 25 pages. https://doi.org/10.1145/3808264

1 Introduction

Unlike program verifiers, symbolic execution [3, 16], bounded model checking [7, 12, 14], or property-based testing [5, 19, 25, 32] frameworks *underapproximate* a program's behavior, with a goal of generating only true positives (i.e., every reported bug is a real bug), but admitting false negatives (i.e., not all real bugs may be reported). To place such tools on a more formal footing, several new underapproximate program logics have recently been proposed [26, 33, 37, 48] for reasoning about *incorrectness* properties of a program, i.e., a characterization of the conditions under which a particular error state is guaranteed to be reached. As one example, O'Hearn defines [33] incorrectness triples of the form $[P]\ \mathsf{c}\ [Q]$, which assert that for any post-state of a command c satisfying assertion Q there *must* exist a pre-state satisfying P. This interpretation underapproximates c's behavior because it requires a witness pre-state to justify every valid post-state.

While these underapproximate axiomatic proof systems have focused on imperative first-order state-based specifications, other recent efforts have instead explored notions of incorrectness and underapproximation using the language of types [38, 43, 45]. Ramsay and Walpole [38], for example,

Authors' Contact Information: Zhe Zhou, Purdue University, West Lafayette, USA, zhou956@purdue.edu; Ankush Desai, Snowflake, San Mateo, USA, ankushdesai@gmail.com; Benjamin Delaware, Purdue University, West Lafayette, USA, bendy@purdue.edu; Suresh Jagannathan, Purdue University, West Lafayette, USA, suresh@cs.purdue.edu.



ACM 2475-1421/2026/6-ART186
https://doi.org/10.1145/3808264

introduces a two-sided type system that allows for the refutation of typing formulas, guaranteeing that not only do well-typed programs not go wrong, but also that ill-typed programs don't evaluate. Zhou et al. [45], on the other hand, leverages notions of underapproximation in the form of *coverage* types to represent the set of values guaranteed to be produced by a test generator.

These type systems consider compositional underapproximate reasoning techniques for pure functional programs, and thus cannot be directly applied to impure functional programs that, e.g., interact with effectful libraries. To address this important limitation, this paper introduces *Underapproximate Hoare Automata Types*, or uHATs, a type abstraction for *underapproximating effectful computations*. A uHAT type judgement has the form $\vdash e : [H]\ t\ [F]$ where H and F qualify type t with pre- (H) and post- (F) conditions. Both conditions are expressed as symbolic regular expressions (SREs) that capture the trace of effects performed before and by e, similar to the HATs proposed by Zhou et al. [47]. Unlike their overapproximate predecessors, uHATs *underapproximate* program behaviors: the judgement above states that for every trace of effects α accepted by F, there exists a trace accepted by H under which e *must* produce α. Intuitively, a uHAT captures the sufficient conditions (i.e., H) that guarantee a set of effects (i.e., F) can be realized by executing e.

To illustrate, consider a program that interacts with a key-value store that provides **get** and **put** operations. The following uHAT underapproximates the behavior of **get** with respect to a context (i.e., trace) of previously performed **put**s:

$$\textbf{get} : k{:}\texttt{Key.t} \to [\bullet^*\cdot\langle \textbf{put}\ k\ v\rangle\cdot\bullet^*]\{v{:}\texttt{Value.t} \mid v = v\}^u[\langle \textbf{get}\ k\ v\rangle]$$

This uHAT asserts that a **get** operation is guaranteed to produce a trace consisting of a single event, $\langle \textbf{get}\ k\ v\rangle$, provided that some **put** operation binding k to v (i.e., $\bullet^*\cdot\langle \textbf{put}\ k\ v\rangle\cdot\bullet^*$) was executed before. The underapproximate interpretation given to uHATs encodes a form of reachability and data dependence between the effects that have occurred prior to executing an operation, and the effect produced by the operation — in this case, the execution of a **get** operation on a key k depends on the execution of a previous **put** operation on k.[1]

In this paper, we show how uHATs can be used to guide a synthesis procedure for the property-based testing (PBT) domain. PBT is an increasingly popular framework for automated testing of user-specified properties. A key component of PBT frameworks are *test generators*, nondeterministic functions that supply input values to the system under test (SUT). To be effective, a generator needs to produce inputs that are relevant to the property of interest, e.g., they should satisfy the preconditions of the SUT. The need to manually write effective generators is a well-known pain point [19] to wider adoption of PBT, and developing generators for properties that impose deep structural or semantic constraints on inputs remains a publication-worthy exercise [18, 20, 34].

The challenges with writing PBT generators are further exacerbated when the SUT is effectful and includes opaque components, e.g., libraries whose source code may not be available or amenable for inspection. In this setting, tests are typically comprised of *sequences* of inputs that induce effects in a particular order to expose latent data and control dependencies relevant to the property being checked, indirectly manipulating library-managed state. Some examples of scenarios where tests are naturally expressed this way include:

(1) **Library API invocations:** The SUT is a library implementing an effectful abstract data type (ADT) equipped with a set of APIs. A test consists of a sequence of API calls that could lead the implementation to an error state (e.g., a violation of a representation invariant [29, 47]).
(2) **Serialized inputs:** The SUT is a procedure that processes serialized input data whose effects are tokens used to reconstruct an AST which is then evaluated. The test generator creates

[1]Note that an overapproximate interpretation of this judgement is not sensible, as it would require that a **get** operation on k be able to simultaneously observe every value v previously **put** into k. This would mean that after performing $\langle \textbf{put}\ k_1\ 2\rangle$ and $\langle \textbf{put}\ k_1\ 3\rangle$, the result of $\langle \textbf{get}\ k_1\rangle$ would be both *both* 2 and 3, which is obviously impossible.

(serialized) inputs that produce semantic dependencies among nodes in the corresponding AST that could violate a property of interest (e.g., evaluation errors due to incorrect treatment of variable scoping or binding).

(3) **Database transactions:** The SUT is a database supporting transactional read and write operations under various isolation levels. A test consists of a sequence of transaction operations whose interleaved order is chosen to expose potential violations of isolation or consistency guarantees the database is supposed to provide.

(4) **Distributed protocols:** The SUT is a distributed system whose components are expected to adhere to protocol invariants. A test is a sequence of message invocations that can lead the system to a global state in which these invariants are not maintained.

Despite being drawn from diverse domains, all these examples share the requirement that an effective test generator needs to be cognizant of how dependencies between the events in traces are (or are not) meaningful to manifesting a property violation. To capture this information succinctly, we use uHATs as a formal specification language to precisely describe latent dependencies among effectful operations, and to ensure that effectful generators adhere to these specifications by conforming to those dependencies. We develop a custom DSL for writing such generators. For example, using the specification given above, a well-typed program written in our DSL would only generate test sequences in which every **`get`** operation is preceded by at least one **`put`** on the same key, while placing no other constraints on the operations that might occur between the two. While this use of uHATs enables us to check whether a manually-written generator explores traces relevant to a target property, the burden of writing such a generator still remains. To alleviate this overhead, we connect these two elements by defining a type-guided program synthesis technique that automatically constructs DSL generator programs using uHAT specifications. The resulting generators are guaranteed to produce traces that violate a target property when executed angelically [2], and can thus evince a concrete bug assuming the SUT provides responses conforming to provided specifications when performing effects.

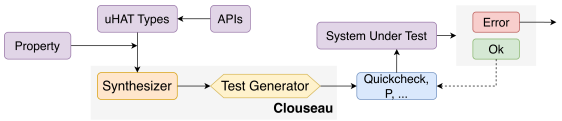


Fig. 1. Clouseau workflow

Fig. 1 depicts the workflow of our system, named Clouseau. The relevant APIs of the SUT are specified using uHATs. Along with the property of interest, these specifications drive the synthesis of a test generator in Clouseau's DSL, which is then transpiled into different testing frameworks; thus far, we have implemented QCheck [35] (OCaml) and P [12] targets.[2] Synthesized test generator programs are guaranteed to produce test sequences that respect the underapproximate interpretation of the uHAT API specifications.

This paper makes the following contributions:

(1) We present a new type abstraction, uHAT, that ascribes an underapproximate interpretation to an effect history trace used to justify the effects produced by an expression.

(2) We develop a DSL for writing test generators typed using uHATs. The DSL, an extension of a typed λ-calculus with support for (asynchronous) effects, nondeterministic choice, and recursion, is sufficiently expressive to be applicable across a diverse range of applications,

(3) We describe a trace-guided synthesis procedure that uses uHAT specifications associated with SUT-provided effectful operators, along with an SRE specification of the property to be tested, to generate well-typed test generators. We evaluate this procedure by comparing the generators synthesized by Clouseau to state-of-the-art baselines on benchmarks whose bugs require deep (effectful) semantic and structural properties of the target SUT to surface.

[2]For P, the generator replaces the default (random) scheduler.

The rest of the paper is organized as follows. Sec. 2 provides additional motivation and examples. Sec. 3 presents the operational semantics and type system for the Clouseau DSL. Sec. 4 describes the synthesis algorithm. A detailed evaluation study over a broad range of diverse benchmarks drawn from the domains given above is given in Sec. 5; these include generating tests for semantically complex data structure properties such as well-typed simply-typed lambda calculus terms using de Bruijn indices, as well as real-world distributed protocols deployed in industry. Sec. 6 presents related work and conclusions.

2 Overview

We propose a trace-based type-guided synthesis framework for generating effectful input test sequences in a PBT system interacting with a SUT comprised of opaque, potentially concurrent and/or distributed, components. Before formalizing our underapproximate type system and detailing our synthesis algorithm, we first present the key features of our approach, followed by illustrative examples from multiple domains.

2.1 Trace-based Specifications

We capture the inputs and observed outputs of an effectful operation (e.g., **`put`** or **`get`** in our earlier example) as an *event* in a *trace*, which is simply a sequence of events. To reason compositionally about dependencies between the effects performed by a generator program, we rely on local specifications of operations in the form of uHats. Our type system uses these specifications to capture the constraints of individual operations in a modular fashion, delegating the burden of managing the event and data dependencies of effectful operations from top-level trace specifications to the underlying type system.

Concretely, we model these local specifications as a collection of uHat-based signatures like the one for **`get`** from the introduction: parameter types constrain the payloads of events, while return types capture relationships between events. We employ symbolic regular expressions (SREs) to qualify trace-based properties within a type, following prior work on trace-based type systems [23, 31, 47]. However, our formulation is distinguished from these other efforts by how it generalizes beyond history-sensitive specifications of effects or state, additionally allowing types to express latent dependencies among arbitrary events.

2.2 The Clouseau DSL

To support test generators across diverse application domains, we introduce a domain-specific language (DSL) that extends a typed lambda calculus core, with recursion, nondeterministic choice ($\oplus$), and effectful operations (**`op`**). This language serves as the basis for writing (and, in our case, synthesizing) effectful test generators. The DSL further allows (angelic) random value generation and property checking via **`assume`** and **`assert`** statements. Concretely, an **`assume`**(ϕ) expression constrains all the program variables at the point at which it appears to satisfy the formula ϕ, while **`assert`**(ϕ) halts program execution if the formula ϕ fails to hold.

2.3 Examples

In order to bias test exploration toward desirable execution paths within a huge search space, test developers typically embed domain knowledge about the SUT into specialized test generators. This knowledge includes not only *basic correctness properties* SUT operations must respect, but also incorporates features of the SUT that can help a test generator focus on interesting configurations. While similar observations guide the design of prior PBT frameworks [20], uHats allow this knowledge to be stated formally and compositionally. These specifications inform our automated synthesis procedure and ensure conformance by the generators it synthesizes, without the need

to *manually craft and tune generation strategies* directly into test generator implementations. The following three examples, drawn from different domains, highlight this point. We show both weak uHats that express only basic correctness properties as well as more precise specifications that are sufficient to guide the synthesizer to produce more effective generators.

Read Atomicity. Database transactions allow concurrent transactions to operate on database state. To mask latency, client-issued read operations are typically implemented as asynchronous request and response event pairs. It is desirable, however, that read operations be logically atomic, i.e., the response to a client-issued read request on a key k always returns the last committed value of k *at the time of the request*, regardless of writes that may have been committed between the request and response. The following trace captures an execution that violates read atomicity[3]:

Client 1: $\langle \mathbf{write}\ 3\rangle$; $\langle \mathbf{write}\ 4\rangle$
Client 2: $\langle \mathbf{readReq}\ \iota_1\rangle; \langle \mathbf{readRsp}\ \iota_1\ 3\rangle; \langle \mathbf{readReq}\ \iota_2\rangle;$ $\langle \mathbf{readRsp}\ \iota_2\ 4\rangle$

Here, Client 2 issues two consecutive asynchronous read operations. A **readReq** event indicates that an asynchronous read request effect has been issued by the client on the single key managed by the database; the asynchronous response from the database is recorded by the **readRsp** event in the trace. Requests are matched to responses using the tag ι. In this trace, however, the second read response witnesses the value of a write performed by another transaction that was issued after its matching request. We are interested in surfacing executions like these from uHat specifications.

The following uHat type on **readRsp** operations admits executions that satisfy read atomicity:

$$\mathbf{readRsp} : [\mathsf{LAST}(\langle \mathbf{write}\ v\rangle) \cdot \langle \mathbf{readReq}\ \iota\rangle \cdot \bullet^*]\{v{:}\mathtt{int} \mid v = v\}^{u}[\langle \mathbf{readRsp}\ \iota\ v\rangle]$$

This type specifies that the response to a previously issued asynchronous **readReq** operation can return a value v, if the last committed write prior to the request wrote the value v. (Here, ι and v are treated as implicitly quantified ghost variables; we make this notion precise in Sec. 3.) Note that this specification also encodes a simple correctness property requiring read responses to correspond to a prior read request with the same tag ι and to reflect a write operation that occurred before the request. A store that enforces read atomicity further constrains the write operations visible to a response. Because this specification is interpreted as an underapproximation of the possible behaviors of the SUT, it establishes a clear dependence between **write** and asynchronous **readReq**/**readRsp** operations that should be respected by any trace which contains these events. This uHat uses $\mathsf{LAST}(\langle \mathbf{op}\ v\rangle)$ as an alias for the following regex:

$$\mathsf{LAST}(\langle \mathbf{op}\ v\rangle) \equiv \bullet^* \cdot \langle \mathbf{op}\ v\rangle \cdot (\bullet \setminus \langle \mathbf{op}\rangle)^* \tag{1}$$

An implementation of **readRsp** that does not satisfy this type could yield a trace in which a response to a request yields a different value than previously seen by the transaction, thus exposing a read atomicity violation. This safety property is captured by the following SRE:[4]

$$\mathsf{LAST}(\langle \mathbf{write}\ v\rangle) \cdot \langle \mathbf{readReq}\ \iota\rangle \cdot \bullet^* \cdot \langle \mathbf{readRsp}\ \iota\ v\rangle \cdot \bullet^* \qquad (A_{\text{atomic}})$$

A violation of this property can be manifested by a test generator that performs sequences of synchronous **write** and asynchronous **read** effects in search of an offending trace similar to the one given above.

Asynchronous effect pairs like **readReq**/**readRsp** allow our DSL to support a form of *interactive* test generation, instructing the SUT to perform effects via **opReq**, and later observing the results via their **opResp** counterparts. For instance, the test generator shown below is automatically synthesized from the above uHat specification and A_{atomic}. The presence of the **readResp** i operation at line 6

[3]For simplicity, we assume the database manages a single key that is elided from event arguments, and that write operations are synchronous and commit their value to the database immediately.
[4]SREs are implicitly universally quantified over any free variables occurring in them.

corresponds to the last effect present in A_{atomic}. The uHAT associated with this operation admits arbitrary operations between it and the matching request, captured by "$\bullet^*$" in the specification, following the $\langle$**readReq** $\iota\rangle$ event in the history trace of **readRsp**. This behavior is subsumed by the recursive call to `transaction_gen` at line 5. The **readReq** effect at line 4 matches the $\langle$**readRsp** $\iota\rangle$ event and the **write** operations on line 3 are synthesized from the $\langle$**write** $v\rangle$ event that precedes the $\langle$**readReq**$\rangle$ in the history trace; since v is unconstrained in the specification, it is instantiated by a call to the `int_gen` primitive generator at line 2. The choice operator at line 2 determines if more recursive calls are required, i.e., if more **readReq** and **writeResp** events are generated. The assertion comparing `y` with `x` determines if a violation (i.e., if the read result is different from the first written value) occurs.

```
1 let rec transaction_gen () =
2   () ⊕ let x = int_gen () in
3     let _ = write x in
4     let i = readReq in
5     transaction_gen ();
6     let y = readResp i in
7     assert (x == y)
```

Alternatively, we can provide a weaker specification that encodes the more basic correctness property that read responses can only return values written before its matching request, making no assumptions about *which* write the response is paired with:

$$\textbf{readRsp} : [\bullet^* \cdot \langle \textbf{write}\ v\rangle \cdot\ \bullet^* \cdot \langle \textbf{readReq}\ \iota\rangle \cdot \bullet^*]\{v{:}\texttt{int} \mid v = v\}^u[\langle \textbf{readRsp}\ \iota\ v\rangle]$$

This specification only requires that the value v returned by the read response is *some* value written before the read was requested: it need not be the *last* value written before the corresponding request. The following generator is also type-safe with respect to this weaker specification: This generator issues two writes followed by a read, and asserts that the read value must be one of the two previously written ones. Notably, although this program is type-safe according to the weaker uHAT, it is not allowed by the original specification, which requires the read value to be the last one. Because this weaker specification does not capture the core property of read atomicity, generators synthesized from this specification may produce benign test cases that would not surface read atomicity violations. Nonetheless, the generator `transaction_gen` can also be derived from this weaker specification. Thus, an imprecise specification simply means that our synthesis algorithm has less guidance on the structure of the generators it produces. In the limit, a specification that places no constraints on the operations of the system under test results in synthesized generators that simply perform random testing.

```
1 let x = int_gen () in let _ = write x in
2 let y = int_gen () in let _ = write y in
3 let i = readReq in let z = readResp i in
4 assert (z == x || z == y)
```

Information-Flow Control. Hritcu et al. [20] applies the PBT methodology to a secure information-flow control (IFC) abstract machine meant to enforce end-to-end noninterference (EENI). This property captures the notion that secret inputs should not influence public outputs: any two executions starting initial states that are indistinguishable (i.e., states in which all "low" security values are the same) should terminate in final states that are also indistinguishable. Hritcu et al. [20] encode the IFC machine input as a sequence of stack machine commands. For instance, the following serialized input corresponds to two IFC programs that both store value 1 from stack into memory, but at different addresses; the only difference is the pointer address which is labeled H and thus not visible to an observer:

$$\textbf{push}(\mathsf{L}, \langle 1, 1\rangle); \textbf{push}(\mathsf{H}, \langle 0, 1\rangle); \textbf{store} \equiv$$

Program 1:	$\textbf{push}_1(\mathsf{L}, 1); \textbf{push}_1(\mathsf{H}, 0); \textbf{store}$	and	Program 2:	$\textbf{push}_2(\mathsf{L}, 1); \textbf{push}_2(\mathsf{H}, 1); \textbf{store}$
Memory 1:	$[(\mathsf{L}, 0); (\mathsf{L}, 0)] \Downarrow [(\mathsf{L}, 1), (\mathsf{L}, 0)]$		Memory 2:	$[(\mathsf{L}, 0); (\mathsf{L}, 0)] \Downarrow [(\mathsf{L}, 0), (\mathsf{L}, 1)]$

Here, L and H are the low (public) and high (secret) security levels, with $\mathsf{L} \sqsubseteq \mathsf{H}$. To ensure indistinguishability, public commands (level L) must use identical argument values in both programs, whereas secret commands (level H) may choose different address values (0 and 1). Since IFC is a

property that holds over pairs of executions, the trace specification **push**(L, ⟨1, 1⟩) identifies two programs performing a **push** operation with the same security label (L) and value (1); in contrast, **push**(H, ⟨0, 1⟩) captures a **push** operation with security label H that pushes value 0 in one execution and 1 in the other. Although both executions start from the same initial memory state ([(L, 0), (L, 0)]), the IFC machine may yield different final states ([(L, 0), (L, 1)] and [(L, 1), (L, 0)]) after performing the **store** which uses the H secret pointer (either 0 or 1) to store the L value (1). The goal of a PBT generator is to generate input sequences like these to determine if an IFC machine implementation will fault prior to the **store** being executed.

To avoid synthesizing generators that produce uninteresting test programs, we would like the executions they induce to: (1) exercise a noninterference correctness property — two input programs must be indistinguishable at the low security level (e.g., **push**(L, 1) and **push**(L, 2) are distinguishable); (2) bias the search to avoid programs that expose uninteresting secrets, for example by storing a high security value at the same address in both executions (e.g., **push**(H, ⟨1, 1⟩)), or pushing values to invalid addresses (e.g., **push**(L, ⟨−1, −1⟩)). The following uHat specification addresses both concerns:

$$\textbf{push} : [\bullet^*]\ \texttt{unit}\ [\langle \textbf{push}\ \mathit{lvl}\ x\ y \mid \mathit{lvl} = \mathsf{L} \iff x = y\rangle]$$

$$\textbf{store} : [\bullet^* \cdot \langle \textbf{push}\ \mathit{lvl}\ x_1\ y_1\rangle \cdot (\bullet \setminus \langle \textbf{store}\rangle)^* \cdot \langle \textbf{push}\ \mathit{lvl}\ x_2\ y_2 \mid \mathit{isAddr}(x_2) \wedge \mathit{isAddr}(y_2)\rangle]\ \texttt{unit}\ [\textbf{store}]$$

The post-condition SRE for **push** enforces that its arguments should only be equal if they are intended to be public, while the pre-condition SRE of a **store** event describes stacks containing at least two elements (i.e., there must be at least two **push** operations with no intervening **store**), with the top element of the stack referencing the memory location to be updated. The desired global property that a generator needs to test can be expressed as $\bullet^* \cdot$ **store**$\cdot\ \bullet^*$ that ensures **store** can be safely performed. The following test generator generates effect sequences consistent with this property and the uHat specifications for **push** and **store** given above:

```
1  let push_gen () =
2    let (x: int) = assume(isAddr x) in let (y: int) = assume(isAddr y) in
3    (assume (x = y); push L ⟨x, y⟩) ⊕ (assume(x != y); push H ⟨x, y⟩)
4  in push_gen (); push_gen (); store;
```

On the other hand, the following specifications only guarantee that tests exercise noninterference, and do not impose any additional constraints on the diversity of secret inputs or address validity:

$$\textbf{push} : [\bullet^*]\ \texttt{unit}\ [\langle \textbf{push}\ \mathit{lvl}\ x\ y \mid \mathit{lvl} = \mathsf{L} \implies x = y\rangle]$$

$$\textbf{store} : [\bullet^* \cdot \langle \textbf{push}\ \mathit{lvl}\ x_1\ y_1\rangle \cdot (\bullet \setminus \langle \textbf{store}\rangle)^* \cdot \langle \textbf{push}\ \mathit{lvl}\ x_2\ y_2 \mid \texttt{true}\rangle]\ \texttt{unit}\ [\textbf{store}]$$

This weaker specification can synthesize generators that produce uninteresting secret values (**push**'s post-condition weakens the bi-implication found in the stronger specification given earlier) or invalid addresses (**store**s no longer constrain the arguments supplied to a previously executed **push** to be addresses); in either case, overall test efficiency is reduced.

STLC Terms. Our final example applies our approach to the well-studied [18, 25, 45] problem of using PBT to test the correctness of an interpreter for simply-typed lambda calculus (STLC) terms. The SUT in this example is an STLC interpreter, and the synthesized test generator produces a family of *serialized* STLC terms, i.e., sequences of string tokens, using a single effectful operation, **token** : `String` → `unit`. Unlike these earlier efforts, we consider terms in which variables are represented by their de Bruijn index [9]. Consider a SUT that fails to correctly increment the index during substitution under a lambda abstraction, leading to incorrect variable capture.

$$\textsc{Reduction rule:}\quad (\lambda t.e)\ v \hookrightarrow e[0 \mapsto v]$$

$$\textsc{Incorrect substitution:}\quad (\lambda t.e)[n \mapsto v] = \lambda t.(e[n \mapsto v])\quad \text{✗ should be } n+1$$

The following term will trigger a bug in our SUT, as this well-typed term will, in fact, get stuck:

$(\lambda \texttt{int}.(\lambda(\texttt{int} \to \texttt{int}).[0]\ [1])\ (\lambda \texttt{int}.[0]))\ 3$ (e_{STLC})

$\hookrightarrow (\lambda(\texttt{int} \to \texttt{int}).3\ [1])\ (\lambda \texttt{int}.3)$ (wrong substitution "$[0] \mapsto 3$")

$\hookrightarrow 3\ [1]$ (stuck!)

Here, our buggy implementation mistakenly substitutes the bound variable "$[0]$" with the argument value 3 in the two inner abstractions, instead of "$[1]$".

Guiding test generation for interpreters over *serialized* STLC terms is challenging, as the generated token sequences must contain *nested function abstractions* with appropriate de Bruijn indices; otherwise, errors involving incorrect variable substitution with non-zero bound variable indices cannot be detected. To enforce the correct use of de Bruijn indices we need to express the relation between indices and abstraction depth. To do so, we introduce a *ghost* event `depth` that tracks the current abstraction depth; as we describe in the following sections, ghost events carry a payload that enables establishing additional dependencies with other events beyond those involving the effects parameters and result that can still be efficiently checked using our type system. Here, **token** is an effect that appends its argument to a serialized string that is eventually passed to the interpreter being tested. A uHat specification to track depth can now be given as follows:

$$[\mathsf{LAST}(\langle \texttt{depth}\ d \rangle) \cdot \langle \textbf{token}\ \boxed{(\lambda t.}\ \rangle \cdot \langle \texttt{depth}\ d+1 \rangle \cdot \bullet^*]\ \mathit{unit}\ [\langle \textbf{token}\ \boxed{)}\ \rangle \cdot \langle \texttt{depth}\ d \rangle]$$

As shown above, upon entering a function abstraction, we retrieve the last tagged depth (assumed to be initialized to 0), increment it at the start of the body, and restore it upon exiting. Leveraging `depth`, we can specify a correct de Bruijn index safety property using the following SRE:

$$\mathsf{LAST}(\langle \texttt{depth}\ d \mid n < d \rangle) \cdot \textbf{token}\ \boxed{[\widehat{n}]} \cdot \bullet^* \qquad (A_\lambda)$$

The bound variable index n is required to be less than the current depth d; the notation $\widehat{n}$ signifies that n's value must be substituted in the string output produced by **token**. The following code snippet shows a test generator for nested abstractions synthesized by Clouseau based on the above specification and safety property:

```
1  let rec abs_gen (d: depth) =
2    (let (i: int) = assume (0 <= i && i < d) in token [î] ) ⊕
3    (let t = random_stlc_ty () in token (λt. ; abs_gen (d + 1); token ) )
```

Observe that the ghost event used in A_λ and the specification for **token** `)` is not present in the code; instead it acts as a hint to the synthesis procedure to construct a recursive function `abs_gen` parameterized by the current depth d, ensuring correct tracking of abstraction nesting.

In order to facilitate test exploration, token sequences should correspond to STLC terms that are both: (1) *well-formed*: all opened lambda abstractions need to eventually be closed and all variables in a term should be bound, so that terms are not immediately rejected by the parser; and (2) *well-typed*: to avoid terms that are discarded by the type checker. The specification given above encodes only the first property, and thus permit a synthesized test generator to produce well-formed but ill-typed STLC terms. A more precise specification that encodes both conditions is provided in the technical report [44]. Although this weaker specification does not prohibit the generation of ill-typed terms, it nonetheless correctly encodes the structure of Bruijn indices, using the ghost variable d to track abstraction depth.

In the following, we present a procedure to synthesize generators that only produce traces consistent with uHat-based specifications and SRE-given safety properties. We emphasize that because these specifications are used primarily to constrain the search space of terms used to synthesize a test generator, they give developers significant leeway on the level of precision captured: less precise underapproximations simply lead to (potentially) less effective generators.

3 Language

Variables		$x, y, z, v, \ldots$
Base Types	$b ::=$	$\texttt{unit} \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{id}\ldots$
Basic Types	$s ::=$	$b \mid s \to s$
Pure Operations	$op ::=$	$+ \mid - \mid == \mid < \mid \leq \mid \texttt{rand\_int} \ldots$
Constants	$c ::=$	$() \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{I}\ldots$
Qualifiers	$\phi ::=$	$c \mid x \mid op\ \overline{v} \mid \bot \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \implies \phi \mid \forall x{:}b.\, \phi$
Effectful Operations	$\mathbf{op} ::=$	$\texttt{put} \mid \texttt{get} \mid \texttt{readReq} \mid \texttt{readRsp} \mid \texttt{write} \mid \texttt{push} \mid \texttt{pop} \mid \ldots$
Values	$v ::=$	$c \mid x \mid \lambda x{:}s.e \mid \texttt{fix}f{:}s.\lambda x{:}s.e$
Expressions	$e ::=$	$v \mid e \oplus e \mid v\ v \mid op\ \overline{v} \mid \mathbf{op}\ \overline{v} \mid \texttt{let}\ x{:}b = e\ \texttt{in}\ e \mid \texttt{assume}\ \phi \mid \texttt{assert}\ \phi$
Events	$m ::=$	$\langle \mathbf{op}\ \overline{v} \rangle$
Traces	$\alpha ::=$	$[\,] \mid m :: \alpha \mid \alpha \# \alpha$
Notations	$e_1; e_2 \doteq$	$\texttt{let}\ \_ = e_1\ \texttt{in}\ e_2$
	$\texttt{let}\ x{:}t = \texttt{assume}\ \phi(x)\ \texttt{in}\ e \doteq$	$\texttt{let}\ x{:}t = \texttt{rand\_int}()\ \texttt{in}\ \texttt{assume}\ \phi(x); e$

Fig. 2. Syntax of λ^C expressions

We formalize our approach using a core language, λ^C, for expressing *trace generators*– nondeterministic programs that probe the behaviors of a black-box system under test via a set of (effectful) operators that the system provides. This calculus is equipped with a type system that characterizes the set of traces that a program *must* be able to produce, *if* the system under test produces appropriate responses. The syntax of λ^C is shown in Fig. 2; the calculus includes both pure and effectful operations (*op* and **op**), nondeterministic choices ($\oplus$), recursion (**fix**), and **assume** and **assert** statements in the style of solver-aided languages [2]. As seen in Sec. 2, asynchronous effects can be modeled as a pair of **opReq**/**opResp** operations, where the former generates an id (given as ι in the syntax) that can then be subsequently referenced when performing a corresponding response operation.

λ^C is equipped with a small-step operational semantics similar to other calculi for trace-based type systems [47]; selected reduction rules are shown in Fig. 3.[5] The reduction relation $\alpha \vDash e \xhookrightarrow{\alpha'} e'$ constrains the output trace α' produced by executing e based on a *history trace* α. The evaluation of pure operations (rule StPOp) makes no use of trace, either history or output. The result of an effectful operation, in contrast, can depend on the events that preceded it, and it produces an output trace containing an event whose payload contains both the arguments and return value of the operation.

Operator Semantics $\boxed{\vDash op(\overline{c}) \Downarrow c \quad \alpha \vDash \mathbf{op}(\overline{c}) \Downarrow c}$

Operational Semantics $\boxed{\alpha \vDash e \xhookrightarrow{\alpha} e}$

$$\frac{\vDash op(\overline{c}) \Downarrow c'}{\alpha \vDash op\ \overline{c} \xhookrightarrow{[\,]} c'}\ \textsc{StPOp} \qquad \frac{\alpha \vDash \mathbf{op}\overline{c} \Downarrow c'}{\alpha \vDash \mathbf{op}\ \overline{c} \xhookrightarrow{[\mathbf{op}\ \overline{c}\ c']} c'}\ \textsc{StEfOp}$$

Fig. 3. Selected reduction rules

3.1 Type System

The syntax of types in λ^C is shown in Fig. 4. Types include *pure* refinement types, which describe pure computations, and underapproximate Hoare Automata Types (uHAts), which describe effectful computations. Pure refinement types are similar to those found in prior underapproximate refinement type systems [45], and allow base types (e.g., `int`) to be further constrained by a logical

[5] The full reduction rules; complete definitions of the auxiliary typing relations (including well-formedness); the full typing rules beyond Fig. 5; further denotational details; and proofs of the theorems in Sec. 3.3 are all given in the technical report [44].

$$
\begin{array}{rrl}
\textbf{Symbolic Regex} & H, F, A ::= & \emptyset \mid \epsilon \mid \langle \mathbf{op}\ \overline{x} \mid \phi \rangle \mid A \vee A \mid A{\cdot}A \mid A^* \mid A \setminus A \mid A \wedge A \mid \bullet \\
\textbf{Pure Refinement Types} & t ::= & \{v{:}b \mid \phi\}^u \mid x{:}t \to t \mid x{:}t \to \tau \mid x{:}b \dashrightarrow t \mid \mathsf{op}{:}s \to t \\
\textbf{Underapproximate HATs} & \tau ::= & [H]\, x{:}t\, [F] \mid \tau \sqcap \tau \\
\textbf{Type Contexts} & \Gamma ::= & \emptyset \mid x{:}t, \Gamma
\end{array}
$$

$$
\textbf{Type Aliases}\quad \langle \mathbf{op} \mid \phi \rangle \doteq \langle \mathbf{op}\ \overline{x} \mid \phi \rangle \quad \langle \mathbf{op}\ \overline{v} \rangle \doteq \langle \mathbf{op}\ \overline{x} \mid \overline{x = v} \rangle \quad \langle \mathbf{op} \rangle \doteq \langle \mathbf{op}\ \overline{x} \mid \top \rangle \quad A_1 A_2 \doteq A_1{\cdot}A_2 \quad b \doteq \{v{:}b \mid \top\}^u
$$

Fig. 4. λ^C types

formula or qualifier. In contrast to traditional type systems where a pure type qualifier $\{v{:}b\}^o$ constrains every value a pure expression *may* evaluate to; the type qualifier $\{v{:}b \mid \phi\}^u$ constrains a subset of the values it *must* evaluate to. For example, the constant value 1 can be assigned the types $\{v{:}\mathtt{int}\}^o$ and $\{v{:}\mathtt{int}\}^o$ – the value 1 satisfies the constraint in both *overapproximate* qualifiers – but, 1 cannot be assigned the type, $\{v{:}\mathtt{int} \mid v = 1 \vee v = 2\}^u$, since it cannot evaluate to both 1 and 2 as required by this *underapproximate* qualifier. The term $1 \oplus 2 \oplus 3$, in contrast, does have this underapproximate type.

Underapproximate Hoare Automata Types. Similar to other recent trace-based type systems [47], λ^C uses *Symbolic Finite Automata* (SFAs) [8, 17, 42] to qualify the traces generated by effectful computations; in contrast to those prior works, the uHATs in λ^C use SFAs to *underapproximate* the set of effects that a program must produce. A uHAT $[H]\, x{:}t\, [F]$ consists of a pure refinement type t and two SFAs: a *history* SFA H that constrains the history trace under which a term is executed, and a *future* SFA F that describes a subset of the events that it must generate as a result. uHATs represent SFAs using symbolic regular expressions (SREs), but in principle could support any encoding of SFAs, e.g., Symbolic LTL$_f$ [10]. SREs support *symbolic events*, $\langle \mathbf{op}\ \overline{x} \mid \phi \rangle$, atomic predicates that describe an effectful operation **op** whose arguments $\overline{x}$ satisfy the qualifier ϕ. SREs provide a convenient language for writing SFAs, supporting versions of the standard regex operators shown in Fig. 4, including union $A \vee A$, sequence $A{\cdot}A$, and Kleene star A^* operators.

The figure also gives the definition of the type aliases that Sec. 2 used to limit user annotation burden. When the fields of a symbolic event are clear from context, we omit its arguments $\overline{x}$, e.g., $\langle \mathbf{push} \mid v > 0 \rangle$ means $\langle \mathbf{push}\ v \mid v > 0 \rangle$. We omit quantifiers with equality constraints, e.g., writing $\langle \mathbf{push}\ v \mid v = 3 \rangle$, as $\langle \mathbf{push}\ 3 \rangle$, and the top ($\top$) qualifier in symbolic events and types.

Example 3.1 (Ghost Events). Consider an effectful stack library implemented as a fixed-size array whose elements grow monotonically. An incorrect implementation that fails to grow the array when the stack is full may cause **push** operations to be dropped. To detect this violation, we need to explore traces of the following form, where the last pop operation raises an "empty stack" exception:

$$\mathbf{push}(1); \mathbf{push}(2); \mathbf{push}(3); \mathbf{pop}(2); \mathbf{pop}(1); \mathbf{pop}(?)$$

Because we would like the uHAT specifications for **push** and **pop** to guide the synthesis of a test generator, they should be interpreted as underapproximations, consisting of nested pairs of **push** and **pop** events. This can be expressed by the following uHAT specifications, using ghost events `pushI` and `popI` as shown below:

$$
\begin{array}{l}
\mathbf{push}: \ n{:}\mathtt{int} \dashrightarrow y{:}\mathtt{int} \dashrightarrow x{:}\{v{:}\mathtt{int} \mid y < v\}^u \to [\mathtt{Last}(\langle \mathtt{pushI}\ n\ y \rangle)]\, \mathtt{unit}\, [\langle \mathbf{push}\ x \rangle{\cdot}\langle \mathtt{pushI}\ n{+}1\ x \rangle] \\
\mathbf{pop}: \ n{:}\mathtt{int} \dashrightarrow m{:}\mathtt{int} \dashrightarrow \\
\quad [\mathtt{Last}(\langle \mathtt{popI}\ m\ \_ \rangle) \wedge \mathtt{Last}(\langle \mathtt{pushI}\ n\ \_ \rangle) \wedge (\bullet^*{\cdot}\langle \mathtt{pushI}\ (n{-}m)\ x \rangle{\cdot}\bullet^*)]\, x{:}\mathtt{int}\, [\langle \mathbf{pop}\ x \rangle{\cdot}\langle \mathtt{popI}\ m{+}1\ x \rangle]
\end{array}
$$

Intuitively, the `pushI` and `popI` ghost events are simply **push** and **pop** events equipped with indices that record the number of push and pop operations. As seen here, ghost events allow uHAT

Auxiliary Typing $\boxed{\Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma \vdash A \subseteq A \quad \Gamma \vdash \tau <: \tau}$ **Typing** $\boxed{\Gamma \vdash \mathbf{op} : t \quad \Gamma \vdash v : t \quad \Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash^{\mathbf{WF}} \tau[x \mapsto v]}{\Gamma \vdash x{:}b \dashrightarrow \tau <: \tau[x \mapsto v]}\ \textsc{SubG} \qquad \frac{\Gamma \vdash t_1 <: t_2 \quad \Gamma, x{:}t_1 \vdash F_2 \subseteq F_1 \quad \Gamma, x{:}t_1 \vdash H_1 \subseteq H_2}{\Gamma \vdash [H_1]\, x{:}t_1\, [F_1] <: [H_2]\, x{:}t_2\, [F_2]}\ \textsc{SubHF}$$

$$\frac{\Gamma \vdash \mathbf{op} : \overline{x{:}t_x} \to [H]\, t\, [F] \quad \Gamma, \overline{x{:}t_x} \vdash H' \subseteq H \quad \Gamma, \overline{x{:}t_x} \nvdash H' \subseteq \emptyset}{\Gamma \vdash \mathbf{op} : \overline{x{:}t_x} \to [H']\, t\, [F]}\ \textsc{TOpHis}$$

$$\frac{\Gamma \vdash v : t}{\Gamma \vdash v : [H]\, t\, [\epsilon]}\ \textsc{TRet} \qquad \frac{\Delta(\mathbf{op}) = t}{\Gamma \vdash \mathbf{op} : t}\ \textsc{TOpCtx} \qquad \frac{\Gamma \vdash \mathbf{op} : \overline{x_i{:}t_i} \to [H]\, t\, [\langle \mathbf{op}\ \overline{x_i} \rangle \cdot F] \quad \forall i.\ \Gamma \vdash v_i : t_i}{\Gamma \vdash \mathbf{op}\ \overline{v_i} : [H]\, t\, [\langle \mathbf{op}\ \overline{v_i} \rangle \cdot F]}\ \textsc{TEffOp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \oplus e_2 : \tau_1 \sqcap \tau_2}\ \textsc{TChoice} \qquad \frac{\Gamma \vdash e_1 : [H]\, x{:}t_x\, [F_1] \quad \Gamma, x{:}t_x \vdash e_2 : [H \cdot F_1]\, t\, [F_2]}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : [H]\, t\, [F_1 \cdot F_2]}\ \textsc{TLet}$$

$$\frac{\Gamma, x{:}t_x \vdash e : \tau}{\Gamma \vdash \lambda x.e : x{:}t_x \to \tau}\ \textsc{TFun} \qquad \frac{}{\Gamma \vdash \mathbf{fix} f.\lambda x.e : \overline{x{:}s} \to [H]\, \{v{:}b \mid \bot\}^u\, [\emptyset]}\ \textsc{TFixBase} \qquad \frac{\Gamma \vdash \mathbf{fix} f.\lambda x.e : t \quad \Gamma, f{:}t \vdash \lambda x.e : t'}{\Gamma \vdash \mathbf{fix} f.\lambda x.e : t'}\ \textsc{TFixInd}$$

Fig. 5. Selected typing rules.

specifications to capture latent data dependencies between effectful operations. The signature for **push** requires that the new pushed value (x) is greater than the last pushed value (y), and adds a new ghost event `pushI` with an incremented index (i.e., $\langle \texttt{pushI}\ n{+}1\ x \rangle$). The signature for **pop** also records the number of pop operations in the same way (i.e., $\langle \texttt{popI}\ m{+}1\ x \rangle$); additionally, it requires the last popped value (x) to be equal to the $(n{-}m)^{th}$ pushed value, reflecting the "last-in-first-out" property of the stack.

3.2 Typing Rules

λ^C is equipped with a pair of typing judgments on pure and effectful expressions, $\Gamma \vdash v : t$ and $\Gamma \vdash e : [H]\, x{:}t\, [F]$, respectively. Both judgments rely on a type context, Γ, that maps from variables to pure refinement types (i.e., t). As in prior work [45], typing contexts are not allowed to contain uHats— doing so breaks several structural properties (e.g., weakening) that are used to prove type safety. Both rules are parameterized over an additional context Δ, which provides types for pure and effectful operators.

Auxiliary typing relations. Our type system also relies on three additional auxiliary relations. The first is a well-formedness relation $\Gamma \vdash^{\mathbf{WF}} \tau$ that ensures, among other things, that all qualifiers appearing in a type τ are closed under the current typing context Γ and that the verification conditions generated by the synthesis algorithm presented in the next section can be encoded as effectively propositional (EPR) sentences [39], which can be efficiently handled by an off-the-shelf theorem prover such as Z3 [11]. We also require the history regex H to not be empty, which guarantees that it represents a meaningful underapproximate precondition.

The type system also uses a semantic subtyping relation tailored for underapproximation; Fig. 5 presents the key subtyping rule for uHats. The rule SubG allows the instantiation of a ghost variable in a uHat into well-formed types. The rule SubHF uses an auxiliary inclusion relation on SREs $\Gamma \vdash A_1 \subseteq A_2$ to relate the history and future regexes of two uHats under the current type context Γ, as well as the return type using the standard subtyping relation on pure refinement types. Notably, because our typing judgements are underapproximate, this subtyping relation is covariant in the history regex and contravariant in the future regex, yielding behavior similar to the reverse rule of consequence found in Incorrectness Logic (IL) [33]. Like that rule, SubHF allows us to strengthen a postcondition (i.e., shrink the future trace), and weaken the precondition (i.e.,

enlarge the history trace). Since the future automata in a uHAT can refer to the pure value it returns, the inclusion check between the future automata extends the typing context with a binding for the return value.

A subset of our typing rules is shown in Fig. 5. All our rules assume any types they use are well-formed, so we elide the corresponding well-formedness judgments from their premises. The TRET rule requires the future regex of a pure term (ϵ) to only accept the empty trace. The operator type is retrieved from the global operator context Δ using the TOPCTX rule. For a given event, there may be multiple reachable history traces (underapproximate preconditions) that can be allowed by an operator that is determined by the context in which the operator executes, and by the operator type alone. We, therefore, expect that the operator type found in the operator context is the most precise one, i.e., it doesn't allow any unreachable history traces, and admits underapproximation specialization. The rule TOPHIS can then freely choose a *non-empty* subset of the reachable history traces to align with the actual calling context. The application rule TEFFOP requires the arguments of an effectful operator to be well-typed against its parameter types; it records the effect as an event in the future trace along with any additional ghost events (F). The nondeterministic choice operator is typed using the TCHOICE rule, which merges the uHATs of the two branches. The TLET rule requires the binding expression e_1 to generate the first part of its future trace F_1, with the rest of trace produced by the body expression e_2 under a new history that includes F_1. The typing rule for functions (TFUN) is standard, while the rule for recursive functions is modeled after the corresponding IL rule, unrolling the recursion as needed. The FIXBASE rule types any recursive function with a uHAT lacking reachability constraints (i.e., $[H]\,\{v{:}b \mid \bot\}^u\,[\emptyset]$). In contrast, the FIXIND rule permits typing a recursive function with type t' by requiring it to have another type t, and by checking that its body can be assigned type t' under the assumption that the function itself has type t.

Example 3.2 (Operator Application Typing). We present a typing derivation for a simple generator that generates **push** and **pop** operations using the signatures from Example 3.1. Consider how we might type check the following expression:

$$\emptyset \vdash \texttt{let}\ x{:}\texttt{int} = \texttt{assume}\ 0 < x\ \texttt{in}\ \textbf{push}\ x; \textbf{pop} :$$
$$[\langle \texttt{pushI}\ 0\ 0\rangle\langle \texttt{popI}\ 0\ 0\rangle]\, x{:}\{v{:}\texttt{int} \mid 0 < v\}^u\, [\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ 1\ x\rangle\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ 1\ x\rangle]$$

The signature of the generator is a uHAT that assumes a history trace in which the stack is empty (i.e., the ghost events `pushI` and `popI` both hold a 0 index and a dummy value). The body of the generator generates a positive number x using **assume**; the underapproximate refinement type of x signifies that it must evaluate to every positive integer. The generator then is expected to pop the same value. We can thus type check its two effectful operation applications using the following judgement:

$$x{:}\{v{:}\texttt{int} \mid 0 < v\}^u \vdash \textbf{push}\ x; \textbf{pop} :$$
$$[\langle \texttt{pushI}\ 0\ 0\rangle\langle \texttt{popI}\ 0\ 0\rangle]\, x{:}\{v{:}\texttt{int} \mid 0 < v\}^u\, [\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ 1\ x\rangle\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ 1\ x\rangle]$$

Using the TEFFOP rule to type the first **push** effect, we retrieve the operator's types from the operator context Δ:

$$\ldots \vdash \textbf{push} : n{:}\texttt{int} \dashrightarrow y{:}\texttt{int} \dashrightarrow$$
$$x{:}\{v{:}\texttt{int} \mid y < v\}^u \to [\texttt{Last}(\langle \texttt{pushI}\ n\ y\rangle)]\ \texttt{unit}\ [\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ n{+}1\ x\rangle] \qquad \text{(by TOPCTX)}$$

We can then apply the subtyping rule (SUBG) to instantiate the signature's ghost variables (e.g., $n \mapsto 0$ and $y \mapsto 0$):

$$\ldots \vdash \textbf{push} : x{:}\{v{:}\texttt{int} \mid 0 < v\}^u \to [\texttt{Last}(\langle \texttt{pushI}\ 0\ 0\rangle)]\ \texttt{unit}\ [\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ 1\ x\rangle] \qquad \text{(by SUBG)}$$

The rule TOpHis can then be used to specialize its history trace to align with that of the actual calling context ($\langle \texttt{pushI}\ 0\ 0\rangle\langle \texttt{popI}\ 0\ 0\rangle$). Note that we do not need to also specialize the future regex, since it is already a prefix of the future trace of the target type:

$$\ldots \vdash \textbf{push} : x{:}\{v{:}\texttt{int} \mid 0 < v\}^{u} \to [\langle \texttt{pushI}\ 0\ 0\rangle\langle \texttt{popI}\ 0\ 0\rangle]\ \texttt{unit}\ [\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ 1\ x\rangle] \qquad \text{(by TOpHis)}$$

The rest of program is now typed via the following judgement:

$$x{:}\{v{:}\texttt{int} \mid 0 < v\}^{u} \vdash \textbf{pop} : [\langle \texttt{pushI}\ 0\ 0\rangle\langle \texttt{popI}\ 0\ 0\rangle\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ 1\ x\rangle]\ x{:}\{v{:}\texttt{int} \mid 0 < v\}^{u}\ [\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ 1\ x\rangle]$$

As before, after looking up the type of **pop** using TOpCtx (①), we can instantiate the ghost variable n as 1 and m as 0 via SubG (②), align the return type via SubHF ((③) and refine the history regex via the rule TOpHis (④):

$$\ldots \vdash \textbf{pop} : n{:}\texttt{int} \dashrightarrow m{:}\texttt{int} \dashrightarrow [\texttt{Last}(\langle \texttt{popI}\ m\ \_\rangle) \wedge \texttt{Last}(\langle \texttt{pushI}\ n\ \_\rangle) \wedge (\bullet^{*}\langle \texttt{pushI}\ (n{-}m)\ x\rangle\bullet^{*})]\ x{:}\texttt{int}\ [\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ m{+}1\ x\rangle] \quad ①$$

$$\ldots \vdash \textbf{pop} : [\texttt{Last}(\langle \texttt{popI}\ 0\ \_\rangle) \wedge \texttt{Last}(\langle \texttt{pushI}\ 1\ \_\rangle) \wedge (\bullet^{*}\langle \texttt{pushI}\ 1\ x\rangle\bullet^{*})]\ x{:}\texttt{int}\ [\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ 1\ x\rangle] \quad ②$$

$$\ldots \vdash \textbf{pop} : [\texttt{Last}(\langle \texttt{popI}\ 0\ \_\rangle) \wedge \texttt{Last}(\langle \texttt{pushI}\ 1\ \_\rangle) \wedge (\bullet^{*}\langle \texttt{pushI}\ 1\ x\rangle\bullet^{*})]\ x{:}\{v{:}\texttt{int} \mid 0 < v\}^{u}\ [\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ 1\ x\rangle] \quad ③$$

$$\ldots \vdash \textbf{pop} : [\langle \texttt{pushI}\ 0\ 0\rangle\langle \texttt{popI}\ 0\ 0\rangle\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ 1\ x\rangle]\ x{:}\{v{:}\texttt{int} \mid 0 < v\}^{u}\ [\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ 1\ x\rangle] \quad ④$$

3.3 Metatheory

Although ghost events constrain uHats, they do not appear in actual traces produced by the operational semantics. We define an erasure function $\lceil\alpha\rceil$ to remove ghost events from a trace α.

Type denotations. Similar to other refinement type systems [22], types in $\boldsymbol{\lambda}^{C}$ are denoted as their inhabitants (i.e., $[\![t]\!]$ and $[\![\tau]\!]$). The type context Γ is denoted as a *substitution* σ (i.e., $[x_1 \mapsto v_1, x_2 \mapsto v_2]$) that provides the assignments for binding variables in Γ and the denotation (the denoted language) of SREs is a set of traces. Then, trace inclusion under a type context is defined as $\Gamma \vdash A \subseteq A' \doteq \forall\sigma \in [\![\Gamma]\!].[\![\sigma(A)]\!] \subseteq [\![\sigma(A')]\!]$. The type denotation of $[H]\ x{:}\{v{:}b \mid \phi\}^{u}\ [F]$ is:

$$\{e \mid \emptyset \vdash_{\mathsf{s}} e : b \wedge \forall v{:}b.\phi[x \mapsto v] \implies \forall\alpha_f \in [\![F[x \mapsto v]]\!].\exists\alpha_h \in [\![H[x \mapsto v]]\!].\lceil\alpha_h\rceil \vDash e \xhookrightarrow{\lceil\alpha_f\rceil}{}^{*}\ v\}$$

A term e inhabits $[H]\ x{:}\{v{:}b \mid \phi\}^{u}\ [F]$ iff, for trace α_f accepted by F, there exists history trace α_h accepted by H, such that the execution of e under α_h produces α_f. Our formulation constitutes a trace-based analogue of IL [33], in which effects are modeled as program state transitions:

$$[P]\ e\ [Q] \doteq \forall s_{\mathsf{post}} \in [\![Q]\!].\exists s_{\mathsf{pre}} \in [\![P]\!].(s_{\mathsf{pre}}, e) \Downarrow s_{\mathsf{post}}$$

A notable distinction between these two presentations is that we characterize F as the set of *newly produced* traces rather than the full post-execution trace, reflecting our trace-based type rules. This formulation additionally avoids redundant prefixes inherited from the execution history.

We expect the uHats provided by the operator typing context Δ to be consistent with the corresponding auxiliary operator semantics:

Definition 3.3 (Well-formed operator typing context). The operator typing context Δ is well-formed iff the semantics of every operator **op** is consistent with the type $\overline{x{:}b_x} \dashrightarrow \overline{y{:}t_y} \to [H]\ z{:}t\ [F]$ provided by Δ, and H does not specify any unreachable traces.

$$\overline{\forall x{:}b_x.}\ \overline{\forall y \in [\![t_y]\!]}.\ \forall H'.H' \neq \emptyset \wedge H' \subseteq H \implies (\textbf{op}\,\overline{y}) \in [\![[H']\ z{:}t\ [F]]\!]$$

Theorem 3.4 (Fundamental Theorem). *A well-typed term, i.e., $\Gamma \vdash e : [H]\ t\ [F]$, generates traces consistent with its uHat: $\forall\sigma \in [\![\Gamma]\!].\sigma(e) \in [\![\sigma([H]\ t\ [F])]\!]$.*

Corollary 3.5 (Type Soundness). *A generator e that satisfies $\emptyset \vdash e : [\epsilon]\ \texttt{unit}\ [F]$ must produce all traces accepted by F, i.e., $\forall\alpha \in [\![F]\!].[] \vDash e \xhookrightarrow{\lceil\alpha\rceil}{}^{*}\ ()$.*

4 Synthesis

Given an SRE A describing some property of a black-box system under test, Corollary 3.5 ensures that a well-typed test generator program $\emptyset \vdash e : [\epsilon]\,\texttt{unit}\,[F]$ *must* be able to produce *every* trace α that is consistent with both F and A, i.e. $\alpha \in F \wedge A$. Thus, the high-level goal of our synthesis procedure is to find a generator that a) uses effectful operators in a way that is consistent with the temporal and data-dependency constraints captured by their specifications in Δ, and b) has a uHat whose future automaton shares a meaningful overlap with A. Our synthesis algorithm simultaneously solves both problems by using a loop, depicted in Fig. 6, to iteratively *refine* the target SRE into a set of more concrete SREs, each of which can be directly mapped to a well-typed generator program. Each iteration of this loop targets a single event in an element of this set and adds events before and after that event so that its dependencies are satisfied, ensuring that the corresponding operator in a synthesized program is well-typed. Our loop also employs a regex non-emptiness check to ensure that each element of this set represents a generator that produces at least one feasible trace. After the refinement loop has finished, a well-typed generator program can be mechanically extracted using the refined property as a template.

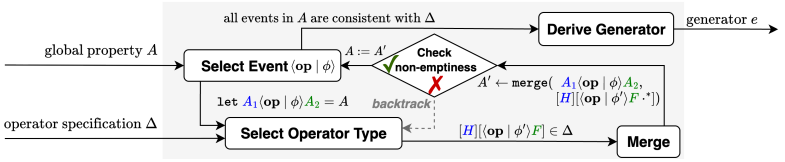


Fig. 6. Clouseau's high-level test generator synthesis algorithm

4.1 Synthesis Algorithm

The main refinement loop in our synthesis algorithm operates over *abstract traces* — sequences of symbolic events $\langle \mathbf{op} \mid \phi \rangle$ plus alternatives defined by the following grammar:

$$\textbf{Abstract Trace} \quad \pi ::= \epsilon \mid \langle \mathbf{op} \mid \phi \rangle^b \mid \pi \cdot \pi \mid [\overline{\langle \mathbf{op} \mid \phi \rangle^b}]^*$$

Each symbolic event $\langle \mathbf{op} \mid \phi \rangle^b$ in an abstract trace is tagged with a boolean b indicating whether the surrounding context satisfies its constraints. Operating over abstract traces helps our synthesizer easily identify events with unresolved dependencies. Every SRE can be normalized into a set of abstract traces via an algorithm that applies a standard minimization algorithm and then decomposes unions into abstract traces: $\langle \mathbf{op_1} \mid \phi_1 \rangle \cup \langle \mathbf{op_2} \mid \phi_2 \rangle$ becomes $\{\langle \mathbf{op_1} \mid \phi_1 \rangle^\perp, \langle \mathbf{op_2} \mid \phi_2 \rangle^\perp\}$, for example.[6]

Example 4.1 (SRE Normalization). The set of abstract traces corresponding to the history regex of **pop** from Example 3.1 includes the following abstract trace:

$$\begin{aligned}
&\text{Original SRE:} && \texttt{Last}(\langle \texttt{popI}\ m\ _\rangle) \wedge \texttt{Last}(\langle \texttt{pushI}\ n\ _\rangle) \wedge (\bullet^* \langle \texttt{pushI}\ (n-m)\ x\rangle \bullet^*) \\
&\text{One normalized trace:} && \Pi^*_{\text{any}} \cdot \langle \texttt{popI}\ m\ _\rangle^\perp \cdot \Pi^*_{\text{noPopI}} \langle \texttt{pushI}\ (n-m)\ x\rangle^\perp \cdot \Pi^*_{\text{noPopI}} \langle \texttt{pushI}\ n\ _\rangle^\perp \cdot \Pi^*_{\text{noI}} && (\pi_{\text{pop1}}) \\
&\text{where} && \Pi_{\text{any}} \equiv [\langle \texttt{pushI}\rangle^\perp \mid \langle \texttt{popI}\rangle^\perp \mid \langle \mathbf{push}\rangle^\perp \mid \langle \mathbf{pop}\rangle^\perp] \\
& && \Pi_{\text{noPopI}} \equiv [\langle \texttt{pushI}\rangle^\perp \mid \langle \mathbf{push}\rangle^\perp \mid \langle \mathbf{pop}\rangle^\perp] \qquad \Pi_{\text{noI}} \equiv [\langle \mathbf{push}\rangle^\perp \mid \langle \mathbf{pop}\rangle^\perp]
\end{aligned}$$

This abstract trace imposes an ordering on the three intersected symbolic events in the original SRE; other traces correspond to different orderings. The alternatives after $\langle \texttt{popI}\ m\ \_\rangle$ and $\langle \texttt{pushI}\ (n-m)\ x\rangle$ in the trace disallow $\langle \texttt{popI}\rangle$ event and $\langle \texttt{pushI}\rangle$ events, respectively. The minimization step also produces traces that merge these intersected events: the procedure automatically considers the case where $m = 0$, which unifies the two `pushI` events, resulting in the inclusion of the following abstract trace in the normalized set:

$$\Pi^*_{\text{any}} \cdot \langle \texttt{popI}\ 0\ \_\rangle^\perp \cdot \Pi^*_{\text{noPopI}} \cdot \langle \texttt{pushI}\ n\ x\rangle^\perp \cdot \Pi^*_{\text{noI}} \qquad (\pi_{\text{pop2}})$$

Algorithm 1: Test Generator Synthesis

Inputs : Operator context Δ, variables $\overline{x{:}b}$, and target trace property A
Output : Generator e, such that $\overline{x{:}\{v{:}b \mid \top\}^u} \vdash e : [\epsilon]\,\mathtt{unit}\,[A'] \wedge \overline{x{:}\{v{:}b \mid \top\}^u} \vdash A' \subseteq A$

1 $\Gamma \leftarrow \overline{x{:}\{v{:}b \mid \top\}^u}$; // initial type context
2 $C \leftarrow \{(\Gamma, \pi) \mid \pi \in \mathbf{Norm}(A)\}$; // initialize set of abstract traces
3 **while** $\exists (\Gamma', \pi) \in C.\ \pi = \pi_h \cdot \langle \mathbf{op} \mid \phi \rangle^{\bot} \cdot \pi_f$ **do** // events still need to be refined
4 $\quad \Pi \leftarrow \mathtt{Refine}(\Delta, \Gamma', \pi_h, \langle \mathbf{op} \mid \phi \rangle^{\bot}, \pi_f) \cup C - \{(\Gamma', \pi)\}$
5 **return** $\mathtt{DeriveTerm}(C)$; // synthesize recursions and derive generator program

Algorithm 2: Abstract Trace Refinement (`Refine`)

Inputs : Operator context Δ, typing context Γ, abstract trace $\pi_h \cdot \langle \mathbf{op} \mid \phi \rangle^{\bot} \cdot \pi_f$
Output : Set of refined abstract traces Π where the dependencies of $\langle \mathbf{op} \mid \phi \rangle$ are satisfied

1 $\overline{z{:}b} \dashrightarrow \overline{y{:}t_y} \to [H]\, x{:}t_x\, [\langle \mathbf{op} \mid \phi' \rangle \cdot F] \leftarrow \Delta(\mathbf{op})$; // retrieve uHat of **op**
2 $\Gamma' \leftarrow \Gamma, \overline{z{:}\{v{:}b \mid \top\}^u}, \overline{y{:}t_y}, x{:}t_x$; // add ghost variables and parameters types to type context
3 $\langle \mathbf{op} \mid \phi'' \rangle \leftarrow \langle \mathbf{op} \mid \phi \wedge \phi' \rangle$; // refine target operation
4 $H' \leftarrow \pi_h \wedge H$; // merge history regex
5 $F' \leftarrow \pi_f \wedge (F \cdot \bullet^*)$; // merge future regex
6 $P \leftarrow \mathbf{Norm}(H' \cdot \langle \mathbf{op} \mid \phi'' \rangle^{\top} \cdot F')$; // normalize refined trace
7 **return** $\{(\Gamma', \pi') \mid \pi' \in P \wedge \Gamma' \vdash \pi' \not\subseteq \emptyset\}$; // filter empty abstract traces

Our top-level synthesis algorithm is shown in Algorithm 1. Given a target trace property A whose qualifiers only contain the variables $\overline{x{:}b}$, this algorithm synthesizes a well-typed $\boldsymbol{\lambda}^C$ generator. The algorithm maintains a set of candidate abstract traces, C, each of which is a refinement of A; this set is initialized directly from A (line 2). Each trace in C is paired with a typing context that the synthesizer uses to filter out refinements that do not correspond to well-typed programs. The algorithm first uses a loop (lines 3 – 4) to refine C until it only contains abstract traces that are consistent with the operator context Δ, and then uses the resulting set of traces to derive the final (recursive) generator (line 5). Each iteration of this loop nondeterministically chooses a target event in one of the current abstract traces (line 2) whose tag indicates its dependencies are not satisfied; it then uses the `Refine` subroutine to insert the required operations before and after the target event.

Abstract trace refinement. The abstract trace refinement subroutine, `Refine`, is shown in Algorithm 2. It first retrieves the uHat of the target operation **op** from Δ (line 1), and adds its parameters to the type context (line 3). `Refine` then merges the selected uHat with the current abstract trace (lines 4 – 6) and normalizes the result to produce a set of refinements of the input trace that are consistent with **op** (line 7). Line 6 effectively applies the SubHF rule to drop any states that are unreachable after the target operation; while line 4 effectively applies TOpHis to ensure the trace that precedes the target operation satisfies its specification. Before returning the refined trace, `Refine` checks it corresponds to a meaningful test generator by filtering out empty traces (line 7).

Example 4.2 (Abstract Trace Refinement). We illustrate the refinement process by showing how it derives a trace corresponding to the generator from in Example 3.2. Assume the refinement loop begins with the following (singleton) set of abstract traces:

$$\Pi^*_{\mathsf{any}} \cdot \langle \mathbf{pop} \mid \top \rangle \cdot \Pi^*_{\mathsf{any}}$$

[6]The full definition of our SRE normalization procedure is provided in the technical report [44].

where the user requires that the test generator performs at least one **pop** operation. In the first iteration, the symbolic event $\langle \textbf{pop} \mid \top \rangle$ is selected for refinement using `Refine`. The algorithm first retrieves the uHat of **pop** from Δ:

$$\Delta(\textbf{pop}) \equiv n{:}\texttt{int} \dashrightarrow m{:}\texttt{int} \dashrightarrow [\pi_{\text{pop1}} \cup \pi_{\text{pop2}} \cup \ldots]\, x{:}\texttt{int}\, [\langle \textbf{pop}\ x\rangle \cdot \langle \texttt{popI}\ m{+}1\ x\rangle] \cdot \Pi^*_{\text{any}}$$

We have normalized the history regex to illustrate that merging this uHat with the current abstract trace and normalizing can result in multiple new abstract traces. For example, property π_{pop2} in the history regex can yield the following trace:

$$(m{:}\{\nu{:}\texttt{int} \mid \nu = 0\}^u, n{:}\{\nu{:}\texttt{int} \mid \top\}^u, y{:}\{\nu{:}\texttt{int} \mid \top\}^u, x{:}\{\nu{:}\texttt{int} \mid \top\}^u,$$
$$\Pi^*_{\text{any}} \langle \texttt{popI}\ m\ y\rangle^{\bot} \Pi^*_{\text{noPopI}} \langle \texttt{pushI}\ n\ x\rangle^{\bot} \Pi^*_{\text{noI}} \langle \textbf{pop}\ x\rangle^{\top} \langle \texttt{popI}\ m\ x\rangle^{\bot})$$

The new variables in the type context require $m = 0$, as mentioned in Example 4.1. Although the uHat of **pop** does not explicitly add any new **push** or **pop** events to the abstract trace, it does require multiple ghost events, which will recover corresponding concrete events during further refinement. For example, the uHat of `pushI` is given below; it requires that a **pushI** event always occurs after a **push** event, unless it is an initialization event (i.e., $i = 0$):

$$\Delta(\texttt{pushI}) \equiv i{:}\texttt{int} \to x{:}\texttt{int} \to [\bullet^* \langle \textbf{push}\rangle]\, \texttt{unit}\, [\langle \texttt{pushI}\ i\ x \mid i > 0\rangle] \sqcap [(\bullet \setminus \langle \texttt{pushI}\rangle)^*]\, \texttt{unit}\, [\langle \texttt{pushI}\ i\ x \mid i = 0\rangle]$$
$$\Delta(\texttt{popI}) \equiv i{:}\texttt{int} \to x{:}\texttt{int} \to [\bullet^* \langle \textbf{pop}\rangle]\, \texttt{unit}\, [\langle \texttt{popI}\ i\ x \mid i > 0\rangle] \sqcap [(\bullet \setminus \langle \texttt{popI}\rangle)^*]\, \texttt{unit}\, [\langle \texttt{popI}\ i\ x \mid i = 0\rangle]$$

On the other hand, the $\langle \texttt{popI}\ m\ y\rangle$ event in the abstract trace has index $m = 0$, and the algorithm can smartly determine that there is no previous **pop** event in this case. It is because the merge result of the first intersected type of `popI` fails the non-emptiness check, where $m > 0 \wedge m = 0$ is unsatisfiable. Finally, this trace will be refined to one consistent with the program shown in Example 3.2:

$$(m{:}\{\nu{:}\texttt{int} \mid \nu = 0\}^u, n{:}\{\nu{:}\texttt{int} \mid \nu = m{+}1\}^u, y{:}\{\nu{:}\texttt{int} \mid \top\}^u, x{:}\{\nu{:}\texttt{int} \mid \nu > y\}^u,$$
$$\Pi^*_{\text{noI}} \cdot \langle \texttt{pushI}\ m\ y\rangle^{\bot} \cdot \langle \texttt{popI}\ m\ y\rangle^{\bot} \cdot \Pi^*_{\text{noI}} \cdot \langle \textbf{push}\ x\rangle^{\bot} \cdot \langle \texttt{pushI}\ n\ x\rangle^{\bot} \cdot \Pi^*_{\text{noI}} \cdot \langle \textbf{pop}\ x\rangle^{\top} \cdot \langle \texttt{popI}\ n\ x\rangle^{\bot} \cdot \Pi^*_{\text{any}})\ \ (\pi_{\texttt{stack}})$$

Deriving a Generator. The second component of our synthesis algorithm, `DeriveTerm`, derives generators for each abstract trace in the set produced the refinement loop and then combines them using $\oplus$ to construct the final generator. This process relies on a subroutine, `DeriveTrace`, that generates straightline programs without recursion by dropping by all ghost events and variables and then mapping each symbolic event to a corresponding effectful operation with appropriate arguments. So, given the following context and abstract trace:

$$(m{:}\{\nu{:}\texttt{int} \mid \nu = 0\}^u, n{:}\{\nu{:}\texttt{int} \mid \nu = m{+}1\}^u, y{:}\{\nu{:}\texttt{int} \mid \top\}^u, x{:}\{\nu{:}\texttt{int} \mid \nu > y\}^u, \langle \textbf{push}\ x\rangle \cdot \langle \textbf{pop}\ x\rangle)$$

`DeriveTrace` will synthesize the following program:

```
1    let (x: int) = assume (∃m n y.m = 0 ∧ n = m+1 ∧ x > y) in push x; (* equivalent to assume true *)
2    let (z: int) = pop () in assert (x == z)
```

`DeriveTerm` applies a sketch-style methodology to synthesize a recursive generator, it iteratively uses `DeriveTrace` to fill in the holes of a template recursive function definition. It does so by associating each of the holes in the template with a subsequence of the abstract trace, treating each occurence of a Kleene star in an abstract trace as a candidate placeholder for a recursive call. In order to ensure a particular partitioning is sensible, the algorithm simulates a bounded number of loop iterations to ensure a completed sketch is well-typed.

Example 4.3 (Recursion Template). Consider the following template which defines and uses a recursive function f that calls itself once: $T \doteq \textbf{let}\ f = \textbf{fix}\ f.\lambda().\ () \oplus \boxed{e_1}; f\ (); \boxed{e_2}\ \textbf{in}\ (\boxed{e_3}; f\ (); \boxed{e_4})$. This template includes a pair of holes before and after the recursive call in the definition of f, and another pair around the top-level call to f. We can build the following two program sketches by mapping portions of $\pi_{\texttt{stack}}$ from Example 4.2 to the four holes in T:

$$\underbrace{\Pi^*_{\texttt{noI}}\langle \texttt{pushI}\ m\ y\rangle\langle \texttt{popI}\ m\ y\rangle\Pi^*_{\texttt{noI}}}_{e_3}\ \underbrace{\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ n\ x\rangle}_{e_1}\ \underbrace{\Pi^*_{\texttt{noI}}}_{f\ ()}\ \underbrace{\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ n\ x\rangle}_{e_2}\ \underbrace{\Pi^*_{\texttt{noI}}}_{e_4}$$

$$\underbrace{\Pi^*_{\texttt{noI}}\langle \texttt{pushI}\ m\ y\rangle\langle \texttt{popI}\ m\ y\rangle\Pi^*_{\texttt{noI}}\langle \textbf{push}\ x\rangle\langle \texttt{pushI}\ n\ x\rangle}_{e_3}\ \underbrace{\Pi^*_{\texttt{noI}}}_{e_1}\ \underbrace{\Pi^*_{\texttt{noI}}}_{f\ ()}\ \underbrace{\langle \textbf{pop}\ x\rangle\langle \texttt{popI}\ n\ x\rangle}_{e_2}\ \underbrace{\Pi^*_{\texttt{noI}}}_{e_4}$$

We can then simulate unrolling the recursive call in a solution to each sketch by duplicating the subtraces labeled $e_1\ f\ ()\ e_2$ a bounded number of times and checking whether they are consistent with $\pi_{\texttt{stack}}$: doing so reveals that the second sketch would not yield a valid solution, as the recursive call would repeatedly perform **pop** events without any **push** events. The first sketch, in contrast, is consistent with $\pi_{\texttt{stack}}$; using `DeriveTrace` to synthesize generators for each e yields the following:

```
let rec f () = () ⊕
    let (x: int) = int_gen () in push x; (* e1 *)
    f ();
    let (z: int) = pop () in assert (x == z) (* e2 *)
in (* e3 *) f () (* e4 *)
```

Theorem 4.4 (Algorithm 1 is Sound). *Every generator synthesized by Algorithm 1 is type-safe with respect to our declarative typing system.*

5 Implementation And Evaluation

We have implemented a tool based on the above approach, called Clouseau, that synthesizes effectful test generators from user-provided uHat specifications and property. The output program in our DSL can be transpiled to OCaml (QCheck) or P. Our experiments use the OCaml transpilation to test effectful abstract datatype libraries, serializers, transformers, and interpreters, concurrent data structures and databases; we use the P target to test reactive distributed system models (i.e., message-passing systems defined as a collection of actors) by using the synthesized generator as a property-directed scheduler that replaces P's default random one. Notably, the same specification and synthesis strategy is used in both cases. Clouseau consists of approximately 14K lines of OCaml code and uses Z3 [11] as its backend SMT solver.

Evaluation setting. Clouseau takes two inputs: a target property, expressed as an SRE, and an operator context that contains uHat specifications for the operators used in the SUT. During synthesis, Clouseau decomposes the target property to build a generator consistent with the traces that the property accepts, but constrained to respect the history and future traces of any effects it references. Because operator types are interpreted underapproximately, there are potentially many valid candidate programs that are consistent with the uHat specifications of the operators they use. Our implementation considers a union of well-typed synthesis candidates, the number of which is bounded by the amount of time allotted for synthesis (3 minutes in our experiments). Synthesized generators use **`assume`** statements to ensure that data dependencies among effectful operations are respected and use **`assert`** statements to flag property violations. Like other PBT setups that rely on precondition-style filtering, our evaluation adopts an *angelic* reading of **`assume`**: when the SUT or runtime rejects a candidate trace (e.g., when the interpreter aborts on an ill-typed expression), we treat that run as uninformative and *retry* with a fresh sample rather than counting it toward a found violation. This matches how generators are typically used in practice and avoids conflating specification imprecision with genuine safety failures.

Our experimental evaluation addresses four research questions:

Q1: Is Clouseau *expressive*? Can it synthesize generators for a diverse set of benchmarks with interesting non-trivial structural and semantic safety properties?

Q2: Is Clouseau *effective*? Do synthesized generators enable more targeted exploration of the state space to witness violations of provided safety properties than existing techniques?
Q3: Is Clouseau *efficient*? Is it able to synthesize meaningful generators reasonably quickly?
Q4: Is Clouseau *robust*? Is it able to synthesize meaningful generators for benchmarks even when provided with weaker specifications than necessary?

We have evaluated Clouseau on a diverse corpus of programs[7] drawn from a variety of sources (described in the captions of Table 1 and Table 2) (**Q1**). Our benchmarks are grouped into four categories. The first are data structure benchmarks (Stack, Set, Filesystem) where the test generator probes for violations of structural invariants. The second are evaluators - programs that input serialized data, transform them into an internal AST representation, and then interpret them; we consider evaluators for data structures like graphs and automata (Graph, NFA), information flow control machines, and simply-typed lambda-calculus, the latter two both described in Sec. 2. The third are concurrent and database benchmarks (Transaction, Shopping, Courseware, Twitter, Smallbank) that test properties related to consistency and synchronization. Property violations in any of these categories only manifest in specific configurations (e.g., a violating test case for NFA involves constructing an automata containing multiple nodes with more than two outgoing edges). We evaluate these three classes of benchmarks by transpiling synthesized generators into OCaml, leveraging its QCheck library as needed. The SUT for each of these benchmarks is known to be buggy; in all cases we use implementations either available from prior sources directly (if the OCaml source was available), or by directly translating them into OCaml. The fourth group of benchmarks shown in Table 2 considers applications involving various kinds of distributed protocols; these are executed using the P runtime environment, by transpiling synthesized generators written in our DSL into P's state machine modeling language. Here as well, the protocol being tested is known to be faulty from the sources they were adopted from. The $\mathsf{IFCLoad}_{\mathsf{W}}$, $\mathsf{DeBruijnHO}_{\mathsf{W}}$, and $\mathsf{Transaction}_{\mathsf{W}}$ benchmarks are variants of other benchmarks that use the weaker specifications described in Sec. 2; the end of this section discusses these results in more detail (**Q4**).

Table 1 divides the results of our experiments into four categories, separated by double bars. The first measures the complexity of benchmarks with respect to the number of distinct effectful operators (#**op**) and the number of qualifiers (i.e., symbolic events) used in uHat specifications and the property being tested. For the benchmarks in Table 1, our results show that writing underapproximate specifications require anywhere from 6 - 22 different operators and include over 100 symbolic events in our most complex examples. The second group of columns characterizes Clouseau's ability to discover a violation compared to the baseline (**Q2**). To make the comparison objective, we execute generators in the two systems 10K times, and count the number of runs in which a fault was uncovered. For example, for Stack, using QCheck to generate random sequences of **push** and **pop** effects, resulted in an erroneous execution every 31.9 times on average; in contrast, every execution of the generator(s) synthesized by Clouseau manifested the bug. Notably, even for benchmarks in which there is a sophisticated customized generator available[8] (e.g., IFC or de Bruijn), uHat-guided synthesis leads to significantly better outcomes. We use MonkeyDB [1], a mock storage system that can be used to find violations of database isolation properties, as the baseline for a number of the concurrency benchmarks. Although it is specialized for this particular application class, and Clouseau is not, evaluation results are comparable, with Clouseau yielding generators that identify weak isolation violations modestly more effectively than the baseline. Similar conclusions apply to the P benchmarks shown in Table 2. For simple benchmarks, P's default scheduler, which performs enumerative state exploration to construct schedules, independently of the behaviors of

[7]The technical report [44] includes additional details about our benchmarks.
[8]When these custom generators are available, we use them as the baseline in our experiments.

Table 1. Using Clouseau to synthesize effectful test generators in OCaml. Benchmarks from prior work are annotated with their source: HAT [47], the OCaml multicore PBT framework [28], QuickCheck on IFC machines [20] and well-typed De Bruijn STLC programs [45]; as well as OLTP-Bench [15], an extensible testbed for benchmarking relational databases, and MonkeyDB [1], a mock storage system for testing weak isolation levels. The baselines used for comparison are taken from the original sources when they exist (these are labeled with †), otherwise we use QCheck's default random generator. Clouseau synthesized correct-by-construction test generators for all benchmarks within 3 minutes; the number of generator variants synthesized is bounded at 3. We set a 30-minute bound for the baselines to find a property violation.

Benchmark		#**op**	#qualifier		# Num. Executions		t_{total}(s)	#evt	#refine	#SMT
Name	Property	uHAT	goal		Clouseau	Baseline				
Stack[47]	Pushes and pops are correctly paired.	6	12	3	1.0	31.9	0.60	15	4	98
Set[47]	Membership holds for every element inserted into the set.	7	22	3	1.0	22.1	1.28	12	12	255
Filesystem[47]	A valid file path only contains non-deleted entries.	7	67	4	1.3	2812.1	6.02	13	14	853
Graph[47]	A serialized stream of nodes and edges is restored to form a fully-connected graph.	6	24	4	1.0	Timeout	16.61	16	16	2114
NFA[47]	An NFA reaches a final state for every string in the language it accepts.	8	50	4	1.0	Timeout	21.85	26	6	5080
IFCStore[20]	A well-behaved IFC program containing a `Store` command never leaks a secret.	8	37	2	2.8	4811.7†	1.02	8	8	160
IFCAdd[20]	A well-behaved IFC program containing an `Add` command never leaks a secret.	8	37	2	1.6	3383.4†	0.96	8	8	154
IFCLoad[20]	A well-behaved IFC program containing a `Load` command never leaks a secret.	8	37	2	15.4	11293.7†	5.76	16	18	930
$\text{IFCLoad}_{\text{W}}$[20]	IFCLoad without search bias (cf. Sec. 2).	8	33	2	115.4	11293.7†	4.83	16	18	754
DeBruijnFO[45]	An STLC interpreter correctly evaluates a *well-typed first-order* STLC program that uses a de Bruijn representation.	10	117	4	1.5	634.0†	40.38	18	26	4765
DeBruijnHO[45]	An STLC interpreter correctly evaluates a *well-typed higher-order* STLC program that uses a de Bruijn representation.	10	118	4	1.4	Timeout†	91.73	18	31	9034
$\text{DeBruijnHO}_{\text{W}}$[45]	DeBruijnHO with *untyped higher-order* programs (cf. Sec. 2).	10	107	4	Timeout	Timeout†	3.23	7	22	957
Shopping[1]	All items added to a cart can be checked-out.	10	60	3	1.0	20.0†	22.10	17	30	1269
HashTable[28]	No updates to a concurrent hashtable are ever lost.	13	38	4	1.0	2.5†	0.76	6	6	57
Transaction	Asynchronous read operations are logically atomic.	8	45	3	1.2	Timeout	2.89	15	16	423
$\text{Transaction}_{\text{W}}$	Transaction without read atomicity enforcement (cf. Sec. 2).	8	40	3	Timeout	Timeout	2.24	15	15	372
Courseware[1]	Every student enrolled in a course exists in the enrollment database for that course.	16	106	3	1.0	57.5†	27.71	18	33	1479
Twitter[1]	Posted tweets are visible to all followers.	16	99	3	1.0	6.3†	61.96	24	24	2339
Smallbank[15]	Account updates are strongly consistent.	22	162	3	1.9	Timeout	163.55	28	29	10263

the actors under test or the target property, or handwritten ones which inject additional actors to control input message generation and prevent uninteresting message orderings (e.g. Simplified2PC or Heartbeat), are effective and Clouseau's performance is comparable to these. As in the previous

Table 2. Experimental results of using Clouseau to synthesize property-directed schedulers for reactive distributed systems. Benchmarks from prior work are annotated with their source: P [12](†), ModP [13](◦) an extension of P with support for modules, and MessageChain [30](⋆), an automated verification tool for P. We also include a real-world model of a two-phase commit protocol (AnonReadAtomicity□) used by a major cloud vendor. The components under test are written in P, and handler specifications for the SUTs components (message-passing actors) are given as uHATs. Clouseau synthesizes a set of schedulers, each of which specifies a distinct scheduling order for messages, all consistent with provided specifications. We set a 3 minute time bound for the synthesis procedure (t_{total} is the average time to find a single controller.) We set a bound of 30 minute time bound for the P baselines to generate a faulty execution.

Benchmark		#op	#qualifier		# Num. Executions		t_{total}(s)	#evt	#refine	#SMT
Name	Property		uHAT	goal	Clouseau	Baseline				
Database	The database maintains a Read-Your-Writes policy.	4	15	3	1.0	5.6	0.32	6	6	49
Firewall[30]	Requests generated inside the firewall are eventually propagated to the outside.	5	26	4	1.0	10.0	0.59	10	10	222
RingLeaderElection[30]	There is always a single unique leader.	3	14	2	1.0	17.7	0.83	8	8	100
BankServer[12]	Withdrawals in excess of the available balance are never allowed.	6	42	1	1.0	2.2†	0.17	5	5	27
Simplified2PC[12]	Transactions are atomic.	9	32	5	2.1	5.8†	1.03	8	8	88
HeartBeat[12]	All available nodes are identified by a detector.	7	31	3	1.0	6.0†	1.10	14	14	145
ChainReplication[13]	Concurrent updates are never lost.	7	72	4	1.0	500.0†	19.24	12	169	2054
Paxos[13]	Logs are correctly replicated.	10	110	2	1.0	667.7†	23.70	14	77	1763
Raft	Leader election is robust to faults.	9	39	6	1.0	Timeout	26.84	12	78	1262
AnonReadAtomicity	Read Atomicity is preserved.	17	113	3	1.0	53.3†	18.35	16	16	1909

set of experiments, however, for more complex benchmarks (e.g., ChainReplication, Paxos), even manual specialization to generate interesting test sequences is significantly less effective than Clouseau's trace-guided synthesis approach. Moreover, as indicated by the Raft numbers, a random exploration procedure, without manually engineered guidance, is too naïve to find a violating execution. It is notable that here too, Clouseau's performance on these P benchmarks is competitive, and in several cases, superior to handcrafted P schedulers, even though Clouseau has no built-in knowledge of P's underlying programming model.

The last group of columns in both tables provide details on the cost of our synthesis procedure. The first column presents total synthesis time (t_{total}), which takes anywhere from .6 to 165 seconds (**Q3**); synthesis time is roughly proportional to benchmark complexity, as reflected in the #**op** and #qualifier columns. The last three columns additionally analyze the characteristics of Clouseau's synthesis procedure with respect to the number of events in the abstract refined trace (#evt), the number of SMT queries (#SMT), as well as the number of refinement steps (#refine); recall that the synthesis procedure uses a refinement loop to construct a more specialized underapproximation from uHAT specifications that are consistent with the provided safety property. In general, these numbers correlate with each other, and serve as rough proxies for benchmark complexity. They indicate that even for the most challenging benchmarks, Clouseau is able to synthesize an effective generator in a reasonable amount of time.

Case study. To demonstrate that Clouseau can be effective in real-world scenarios, we have applied it to AnonReadAtomicity, a distributed transaction system in use at a major cloud provider, where asynchronous reads are expected to execute atomically, as discussed in Sec. 2. The property to

be checked requires that if there exists a key k updated within an active transaction, any successful read response asking its value should return the value last written to k made by that transaction. Generating a fault-inducing scenario requires (a) initiating a new transaction with transaction id `tid`, (b) successfully performing a write within that transaction, and then (c) subsequently performing a read within `tid` that yields a different value than the one last written. As the example in 2 implies, using uHat to express the behavior of **`readRsp`** enables the synthesis of a test generator that can strategically request a new transaction to initiate triggering the intended violation. A version of the benchmark in which this sequence structure is enforced manually discovers the violation in 53 executions, while Clouseau's version can uncover the violation on all of its executions.

Robustness to Specification Quality. As discussed in Sec. 2, Clouseau's effectiveness depends on the quality of user-supplied uHat specifications. To quantify Clouseau's robustness to specification quality, we conducted an additional study using variants of the benchmarks in Tables 1 and 2 that keep the SUT and the global property fixed while weakening the uHat specifications of individual operators provided by the SUT. Specifications were weakened in a similar manner to the examples described in Sec. 2: the variant of the IFC benchmark ($\mathsf{IFCLoad_W}$), for example, eliminates the "valid address" constraint (i.e., *isAddr*) used in the original specification. Broadly speaking, the weakened specifications can be categorized as either relaxing basic correctness properties of the SUT or removing search biases. Both categories of specifications are *lower-quality* because they provide less guidance to Clouseau and thus increase the risk that it will synthesize generators misaligned with the SUT's intended behavior.

Table 1 includes the results of this experiment for the three examples from Sec. 2 ($\mathsf{Transaction_W}$, $\mathsf{IFCLoad_W}$, and $\mathsf{DeBruijnHO_W}$).[9] Under these weaker specifications, average synthesis time (t_{total}) improves by 18.18%, and by 78.07% in the extreme case (ChainReplication). This comes at the cost of decreased testing effectiveness, however: 26.9% (7 out of 26) of the benchmarks now time out; the remaining benchmarks show an elevated retry rate (#Num. Executions), 1.9× on average, and 7.5× in the worst case ($\mathsf{IFCLoad_W}$). In general, weakening basic correctness properties significantly impacts Clouseau's effectiveness, as the resulting generators regularly yield executions that do not align with the intended behavior of the SUT: $\mathsf{DeBruijnHO_W}$, for example, generates a large number of "useless" untyped STLC programs that are immediately discarded by the type-checker. Obtaining an informative witness from such a generator may thus require generating many more inputs, and can quickly exhaust the testing time budget. In contrast, weakening the search bias was comparatively benign in our experiments (e.g., the synthesized generator for $\mathsf{IFCLoad_W}$ still finds EENI violations relatively quickly), and primarily manifested in an increased number of retries before a bug was found. Of course, since weaker specifications can lead to shorter synthesis times, the loss of testing efficiency may sometimes be an acceptable trade-off.

Discussion. We have anecdotal evidence that the cost to write uHats is relatively low; the uHats for the HashTable and Courseware benchmarks in Table 1 were written by an undergraduate student, and those for Smallbank and Twitter by two first-year PhD students. None of the students had prior exposure to the uHat specification language or Clouseau's synthesis algorithm. Nonetheless, each was able to independently encode their knowledge about the SUT as uHats, requiring only a few person-hours to complete all the benchmarks.

One potential area for future work is to provide better developer-facing feedback when synthesis fails. Because our algorithm consists of a refinement loop that maintains a set of candidate generators (see Fig. 6), integrating a refinement and debugging phase as part of the synthesizer and test execution pipeline would be a natural extension.

[9]The experimental results for the full set of benchmarks are provided in the technical report [44].

6 Related Work and Conclusions

Underapproximate Reasoning. A number of recently proposed logical frameworks provide program reasoning principles based on under-, rather than over-, approximate abstractions. Incorrectness Logic [33] and related variants [26, 36, 48] present compositional proof rules that enable verification of reachability (aka incorrectness) properties of first-order imperative programs; their primary motivation is to enable formal reasoning of (unsound) program analysis tools such as Infer [16]. Ideas related to these notions have also been incorporated into type systems [21, 27, 38, 45] for functional languages, but none of these efforts have considered their application to the synthesis of effectful test generators of the kind proposed in this paper. For example [45] uses underapproximation to typecheck that a PBT generator is *complete* with respect to the functional inputs it generates to the SUT, while [24] considers an enumerative repair strategy to patch incomplete generators so that they become complete. In contrast, in this paper, we show how interpreting type specifications from the lens of underapproximation can guide a synthesis procedure to construct bespoke effectful test generators tailored to the property of interest, informed by the behavior of the operators provided by the SUT. As a mechanism to characterize underapproximate behavior, types identify *sufficient* effectful constraints that any test input sequence produced by the generator *must* satisfy based on these specifications.

Specializing PBT Generators. There has been significant prior work to make PBT implementations more effective, by providing tools that enable reasoning and customization of effect traces produced by a test generator. For example, Claessen et al. [6] describes a customizable user-level scheduler that uses Quviq QuickCheck's support for state machine models to implement deterministic replayable test inputs to an Erlang SUT. Concuerror [4] systematically explores process interleavings in a concurrent Erlang program via program instrumentation and preemption bounding. Quickstrom [32] uses specifications of possible actions expressed in its Specstrom specification language, a dialect of LTL, to dynamically choose the next input that the generator should provide for interactive applications. In contrast, Clouseau's type-guided synthesis strategy yields generators influenced by both the global property of interest and the local uHAT specifications of the operations used by the SUT. Importantly, the synthesis procedure is guided by the history and future SREs of provided specifications to determine a semantically meaningful ordering of effectful actions.

Type and Effect Systems. Type and effect systems that target *temporal* properties on the sequences of effects that a program may *produce* is a well-studied subject. For example, Skalka and Smith [41] presents a type and effect system for reasoning about the shape of histories (i.e., finite traces) of events embedded in a program. Koskinen and Terauchi [23] present a type and effect system that additionally supports verification properties of infinite traces, specified as Büchi automata. More recently, Sekiyama and Unno [40] have considered how to support richer control flow structures, e.g., delimited continuations, in such an effect system. Closest to our work are *Hoare Automata Types* (HATs) [47], which integrate symbolic finite automata into a refinement type system. HATs enable reasoning about stateful sequential programs structured as a functional core interacting with opaque effectful libraries. uHATs revise HATs by viewing them as underapproximate specifications — every trace accepted by the future regex in a uHAT is justified by a trace accepted by its history.

Conclusions. This paper proposes Clouseau, a trace-guided synthesizer of effectful PBT generators. Its key innovations are (i) the adaptation of SREs, expressed as types, that define traces of effectful actions that occur before and after the execution of an expression, as underapproximate specifications, and (ii) the synthesis of bespoke test generators derived from these specifications. Experimental results on a wide range of applications show that Clouseau is significantly more effective in identifying test input sequences that falsify a property than the existing state-of-the-art.

Acknowledgements

We thank the anonymous reviewers for their detailed comments and suggestions. We are grateful to Aadi Rave, John Rose, Rebekah Sowards (Purdue), and Zhendong Ang (NUS) for their help with our experiments. We also thank Prasita Mukherjee (Purdue) and Tingxuan Xia (PKU) for their helpful feedback on the artifact. This material is based upon work supported by the National Science Foundation under CCF-SHF 2321680.

Data Availability

An artifact containing our implementation, benchmark suite, and results is publicly available on Zenodo [46].

References

[1] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness Under Weak Isolation Levels. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 132 (Oct. 2021), 27 pages. doi:10.1145/3485546

[2] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with angelic nondeterminism. *SIGPLAN Not.* 45, 1 (Jan. 2010), 339–352. doi:10.1145/1707801.1706339

[3] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 459–465. doi:10.1007/978-3-642-20398-5_33

[4] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 154–163. doi:10.1109/ICST.2013.50

[5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/351240.351266

[6] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. 2009. Finding race conditions in Erlang with QuickCheck and PULSE. *SIGPLAN Not.* 44, 9 (Aug. 2009), 149–160. doi:10.1145/1631687.1596574

[7] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176. doi:10.1007/978-3-540-24730-2_15

[8] Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. *SIGPLAN Not.* 49, 1 (Jan 2014), 541–553. doi:10.1145/2578855.2535849

[9] N.G de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, A Tool For Automatic Formula Manipulation, With Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. doi:10.1016/1385-7258(72)90034-0

[10] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (Beijing, China) *(IJCAI '13)*. AAAI Press, 854–860. https://dl.acm.org/doi/10.5555/2540128.2540252

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[12] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *SIGPLAN Not.* 48, 6 (June 2013), 321–332. doi:10.1145/2499370.2462184

[13] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional Programming and Testing of Dynamic Distributed Systems. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 159 (Oct. 2018), 30 pages. doi:10.1145/3276529

[14] Ankush Desai and Shaz Qadeer. 2017. P: Modular and Safe Asynchronous Programming. In *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10548)*, Shuvendu K. Lahiri and Giles Reger (Eds.). Springer, 3–7. doi:10.1007/978-3-319-67531-2_1

[15] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. doi:10.14778/2732240.2732246

[16] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. doi:10.1145/3338112

[17] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification*. Springer, 47–67. doi:10.1007/978-3-319-63387-9_3

[18] Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proc. ACM Program. Lang.* 8, POPL (2024), 2318–2339. doi:10.1145/3632919

[19] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 187:1–187:13. doi:10.1145/3597503.3639581

[20] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 455–468. doi:10.1145/2500365.2500574

[21] Robert Jakob and Peter Thiemann. 2015. A Falsification View of Success Typing. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 234–247. doi:10.1007/978-3-319-17524-9_17

[22] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. doi:10.1561/2500000032

[23] Eric Koskinen and Tachio Terauchi. 2014. Local Temporal Reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. Association for Computing Machinery, New York, NY, USA, Article 59, 10 pages. doi:10.1145/2603088.2603138

[24] Patrick LaFontaine, Zhe Zhou, Ashish Mishra, Suresh Jagannathan, and Benjamin Delaware. 2025. We've Got You Covered: Type-Guided Repair of Incomplete Input Generators. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 380 (Oct. 2025), 29 pages. doi:10.1145/3763158

[25] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's luck: A Language for Property-Based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 114–129. doi:10.1145/3009837.3009868

[26] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–27. doi:10.1145/3527325

[27] Celia Mengyue Li, Sophie Pull, and Steven Ramsay. 2026. A Complementary Approach to Incorrectness Typing. *Proc. ACM Program. Lang.* 10, POPL, Article 82 (Jan. 2026), 29 pages. doi:10.1145/3776724

[28] Jan Midtgaard. 2025. Property-Based Testing of OCaml 5's Runtime System: Fun and Segfaults with Interpreters and State Transition Functions. In *OlivierFest'25* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, 180–193. doi:10.1145/3759427.3760378

[29] Anders Miltner, Saswat Padhi, Todd D. Millstein, and David Walker. 2020. Data-driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*. ACM, 1–15. doi:10.1145/3385412.3385967

[30] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. 2023. Message Chains for Distributed System Verification. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 300 (Oct. 2023), 27 pages. doi:10.1145/3622876

[31] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 759–768. doi:10.1145/3209108.3209204

[32] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1025–1038. doi:10.1145/3519939.3523728

[33] Peter W. O'Hearn. 2019. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. doi:10.1145/3371078

[34] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. Association for Computing Machinery, New York, NY, USA, 91–97. doi:10.1145/1982595.1982615

[35] QCheck 2024. *QuickCheck Inspired Property-Based Testing for OCaml*. https://c-cube.github.io/qcheck/

[36] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 225–252. doi:10.1007/978-3-030-53291-8_14

[37] Azalea Raad, Julien Vanegue, Josh Berdine, and Peter O'Hearn. 2023. A General Approach to Under-Approximate Reasoning About Concurrent Programs. In *34th International Conference on Concurrency Theory (CONCUR 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 279)*, Guillermo A. Pérez and Jean-François Raskin (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:17. doi:10.4230/LIPIcs.CONCUR.2023.25

[38] Steven Ramsay and Charlie Walpole. 2024. Ill-Typed Programs Don't Evaluate. *Proc. ACM Program. Lang.* 8, POPL, Article 67 (Jan. 2024), 31 pages. doi:10.1145/3632909

[39] F. P. Ramsey. 1987. *On a Problem of Formal Logic.* Birkhäuser Boston, Boston, MA, 1–24. doi:10.1007/978-0-8176-4842-8_1

[40] Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL, Article 71 (jan 2023), 32 pages. doi:10.1145/3571264

[41] Christian Skalka and Scott Smith. 2004. History Effects and Verification. In *Programming Languages and Systems*, Wei-Ngan Chin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–128. doi:10.1007/978-3-540-30477-7_8

[42] Margus Veanes. 2013. Applications of Symbolic Finite Automata. In *Implementation and Application of Automata*, Stavros Konstantinidis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–23. doi:10.1007/978-3-642-39274-0_3

[43] Robin Webbers, Klaus von Gleissenthall, and Ranjit Jhala. 2024. Refinement Type Refutations. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 305 (Oct. 2024), 26 pages. doi:10.1145/3689745

[44] Zhe Zhou, Ankush Desai, Benjamin Delaware, and Suresh Jagannathan. 2026. Trace-Guided Synthesis of Effectful Test Generators. arXiv:2604.04345 [cs.PL] doi:10.48550/arXiv.2604.04345

[45] Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 2023. Covering All the Bases: Type-Based Verification of Test Input Generators. *Proc. ACM Program. Lang.* 7, PLDI, Article 157 (June 2023), 24 pages. doi:10.1145/3591271

[46] Zhe Zhou, John Rose, RebekahSowards, and Aadi Rave. 2026. *PLDI26 Artifact: Trace-Guided Synthesis of Effectful Test Generators.* doi:10.5281/zenodo.19076511

[47] Zhe Zhou, Qianchuan Ye, Benjamin Delaware, and Suresh Jagannathan. 2024. A HAT Trick: Automatically Verifying Representation Invariants using Symbolic Finite Automata. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1387–1411. doi:10.1145/3656433

[48] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (April 2023), 29 pages. doi:10.1145/3586045

Received 2025-11-13; accepted 2026-04-03