Symbolic Execution Game Semantics

Yu-Yang Lin
Queen Mary University of London, UK

Nikos Tzevelekos
Queen Mary University of London, UK

Abstract

We present a framework for symbolically executing and model checking higher-order programs with external (open) methods. We focus on the client-library paradigm and in particular we aim to check libraries with respect to any definable client. We combine traditional symbolic execution techniques with operational game semantics to build a symbolic execution semantics that captures arbitrary external behaviour. We prove the symbolic semantics to be sound and complete. This yields a bounded technique by imposing bounds on the depth of recursion and callbacks. We provide an implementation of our technique in the $\mathbb{K}$ framework and showcase its performance on a custom benchmark based on higher-order coding errors such as reentrancy bugs.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases game semantics, symbolic execution, higher-order open programs

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.27

Supplementary Material The tool and its benchmarks can be found at: `https://github.com/LaifsV1/HOLiK`.

Funding Research funded by EPSRC (EP/P004172/1).

Acknowledgements We would like to thank members of the K framework for their consistent support with K, and anonymous reviewers for their insightful reviews.

1 Introduction

Two important challenges in program verification are state-space explosion and the environment problem. The former refers to the need to investigate infeasibly many states, while the latter concerns cases where the code depends on an environment that is not available for analysis. State-space explosion has been approached with a range of techniques, which have led to verification tools being nowadays routinely used on industrial-scale code (e.g. [10, 5, 7]). The environment problem, however, remains largely unanswered: verification techniques often require the whole code to be present for the analysis and, in particular, cannot analyse components like libraries where parts of the code are missing (e.g. the client using the library). This problem is particularly acute in higher-order programs, where the interaction between a program and its environment can be intricate and e.g. involve callbacks or reentrant calls. In this paper we address this latter problem by combining *game semantics*, a semantics theory for higher-order programs, with *symbolic execution*, a technique that uses *symbolic values* to explore multiple execution paths of a program.

To showcase the importance and challenges of the environment problem, following is a simple example of a library written in a sugared version of HOLi, the vehicle language of this paper. The example is a simplified implementation of "The DAO" smart contract, a failed decentralised autonomous organisation on the Ethereum blockchain platform [12]. As with libraries, the challenge in analysing smart contracts is that the client code is not available. We must thus generate all possible contexts in which the contract can be called. In this case, the error is caused by a reentrant call from the `send()` method, which is provided by the

5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020).
Editor: Zena M. Ariola; Article No. 27; pp. 27:1–27:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

environment. When this method is called, the environment takes control and is allowed to call any method in the library. If a client were to call `withdraw()` within its `send()` method, the recursive call would drain all the funds available, which is simulated in this example by a negative balance. This happens because the method is manipulating a global state, and is updating it after the external call. We can see that an analysis capturing this error would need to be able to predict an intricate environment behaviour. Moreover, such an analysis should ideally only predict realisable environment behaviours.

```
1 import send:(int → unit)
2 int balance := 100;
3
4 public withdraw (m:int) :(unit) =
5   if (not (!balance < m)) then
6        send(m);
7        balance := !balance - m;
8        assert(not(!balance < 0))
9   else ();
```

Symbolic execution [33, 13, 19] explores all paths of a program using symbolic values instead of concrete input values. Each symbolic path holds a path condition (a SAT formula) that is satisfiable if and only if the path can be concretely executed. While the resulting analysis is unbounded in general, by restricting our focus to bounded paths we can soundly catch errors, or affirm the absence thereof up to the used bound. Game semantics [2, 14], on the other hand, models higher-order program phrases in isolation as 2-player games: sequences of computational *moves* (method calls and returns) between the program and its hypothetical environment. The power of the technique lies in its use of combinatorial conditions to precisely allow those game plays that can be realised by including the program in an actual environment. Moreover, the theory can be formulated operationally in terms of a trace semantics for open terms [18, 21, 16] which, in turn, lends itself to a symbolic representation. The latter yields a symbolic execution technique that is *sound and complete* in the following sense: given an open program, its symbolic traces match its concrete traces, which match its realisable traces in some environment.

Returning to the DAO example, we can model the ensuing interaction as a sequence of moves, alternating between the environment and the library. Any finite sequence of moves (that leads to an assertion violation) is a trace defining a counterexample. Running the example in HOLiK, our implementation of the symbolic semantics in the $\mathbb{K}$ Framework [32], the following minimal symbolic trace is automatically found:

$$\begin{aligned}&call\langle withdraw, x_1\rangle \cdot call\langle send, x_1\rangle \cdot call\langle withdraw, x_2\rangle\\&\quad \cdot call\langle send, x_2\rangle \cdot ret\langle send, ()\rangle \cdot ret\langle withdraw, ()\rangle \cdot ret\langle send, ()\rangle\end{aligned}$$

where x_1 is the original call parameter, and x_2 is the parameter for the reentrant call, satisfiable with values $x_1 = 100$ and $x_2 = 1$. A fix would be to swap line 6 and 7, to update internal state before passing control.

In Appendix A we look at a few more examples of libraries that exhibit errors due to high-order behaviours. We provide three examples: a file lock example, a double deallocation example, and an unsafe implementation of flat-combining.

Overall, this paper contributes a novel symbolic execution technique based on game semantics to precisely model the behaviour of higher-order stateful programs. Specifically:

- We present a symbolic trace semantics for higher-order libraries that captures the behaviour of an unknown environment, and prove it sound and complete: i.e. it produces no spurious error traces, and is able to produce the complete execution tree of any library.

$$
\begin{array}{llll}
\textit{Libraries} \;\; L ::= & B \mid \texttt{abstract}\ m; L & \textit{Terms} \;\; M ::= & m \mid i \mid () \mid x \mid \lambda x.M \mid r := M \mid !r \\
\textit{Blocks} \;\; B ::= & \varepsilon \mid \texttt{public}\ m = \lambda x.M; B & & \mid M \oplus M \mid \langle M, M\rangle \mid \pi_1 M \mid \pi_2 M \\
& \mid m = \lambda x.M; B \mid \texttt{global}\ r := i; B & & \mid MM \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \\
& \mid \texttt{global}\ r := \lambda x.M; B & & \mid \texttt{letrec}\ x = \lambda x.M\ \texttt{in}\ M \\
\textit{Clients} \;\; C \;\; ::= & L; \texttt{main} = M & & \mid \texttt{let}\ x = M\ \texttt{in}\ M \mid \texttt{assert}(M)
\end{array}
$$

$$
\frac{}{() : \texttt{unit}} \quad \frac{}{i : \texttt{int}} \quad \frac{x \in \texttt{Vars}_\theta}{x : \theta} \quad \frac{m \in \texttt{Meths}_{\theta,\theta'}}{m : \theta \to \theta'} \quad \frac{M, M' : \texttt{int}}{M \oplus M' : \texttt{int}} \quad \frac{M : \texttt{int} \qquad M_1, M_0 : \theta}{\texttt{if}\ M\ \texttt{then}\ M_1\ \texttt{else}\ M_0 : \theta}
$$

$$
\frac{M : \theta_1 \quad M' : \theta_2}{\langle M, M'\rangle : \theta_1 \times \theta_2} \quad \frac{\langle M, M'\rangle : \theta_1 \times \theta_2}{\pi_i\langle M, M'\rangle : \theta_i} \quad \frac{r \in \texttt{Refs}_\theta}{!r : \theta} \quad \frac{r \in \texttt{Refs}_\theta \quad M : \theta}{r := M : \texttt{unit}} \quad \frac{M' : \theta \to \theta' \quad M : \theta}{M'\,M : \theta'}
$$

$$
\frac{M : \theta' \quad x : \theta}{\lambda x.M : \theta \to \theta'} \quad \frac{x, M : \theta \quad M' : \theta'}{\texttt{let}\ x = M\ \texttt{in}\ M' : \theta'} \quad \frac{x, \lambda y.M : \theta \to \theta'' \quad M' : \theta'}{\texttt{letrec}\ x = \lambda y.M\ \texttt{in}\ M' : \theta'} \quad \frac{M : \texttt{int}}{\texttt{assert}(M) : \texttt{unit}}
$$

Figure 1 Syntax and typing rules of HOLi.

- By bounding the depth of nested calls and the *insistence* of the environment in calling library methods, we derive a sound and bounded-complete technique to check higher-order libraries for errors.
- We implement the latter in the $\mathbb{K}$ semantical framework [32] to produce a sound and bounded-complete tool for higher-order libraries as a proof of concept. We test our implementation with benchmarks adapted from the literature.

Some material has been delegated to an Appendix.

2 A Language for Higher-Order Libraries: HOLi

We introduce HOLi, a language for higher-order libraries which define methods to be used by an external client, and in turn require external methods (provided by the client). We give in HOLi an operational semantics for terms that integrates a counter for the depth of nested calls that a program phrase can make. We then extend this counting semantics to open terms by means of a trace semantics. We show that the trace semantics of libraries is sound and complete for reachability of errors under any external client.

2.1 Syntax and operational semantics

A library in HOLi is a collection of typed higher-order methods. A client is simply a library with a main body. Types are given by the grammar:

$$\theta \;::=\; \texttt{unit} \mid \texttt{int} \mid \theta \times \theta \mid \theta \to \theta$$

We use countably infinite sets $\texttt{Meths}$, $\texttt{Refs}$ and $\texttt{Vars}$ for method, global reference and variable names, ranged over by m, r and x respectively, and variants thereof; while i is for ranging over the integers. We use $\oplus$ to range over a set of binary integer operations, which we leave unspecified. Each set of names is typed, that is, it can be expressed as a disjoint union as follows: $\texttt{Meths} = \biguplus_{\theta,\theta'} \texttt{Meths}_{\theta,\theta'}$, $\texttt{Refs} = \biguplus_{\theta \neq \theta_1 \times \theta_2} \texttt{Refs}_\theta$, $\texttt{Vars} = \biguplus_\theta \texttt{Vars}_\theta$.

The full syntax and typing rules are given in Figure 1. Thus, a library consists of abstract method declarations, followed by blocks of public and private method and reference definitions. A method is considered private unless it is declared `public`. Each public/private method and reference is defined once. Abstract methods are not given definitions: these methods are external to the library. Public, private and abstract methods are all disjoint.

$$(E[\texttt{let}\ x = v\ \texttt{in}\ M], R, S, k) \to (E[M\{v/x\}], R, S, k) \qquad (E[\pi_j\langle v_1, v_2\rangle], R, S, k) \to (E[v_j], R, S, k)$$
$$(E[r := v], R, S, k) \to (E[()], R, S[r \mapsto v], k) \qquad (E[!r], R, S, k) \to (E[S(r)], R, S, k)$$
$$(E[\texttt{if}\ i\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_j], R, S, k)\ \ (1) \qquad (E[i_1 \oplus i_2], R, S, k) \to (E[i], R, S, k)\ \ (2)$$
$$(E[\lambda x.M], R, S, k) \to (E[m], R \uplus \{m \mapsto \lambda x.M\}, S, k) \qquad (E[\texttt{assert}(i)], R, S, k) \to (E[()], R, S, k)\ \ (3)$$
$$(E[mv], R, S, k) \to (E[(\!|M\{v/x\}|\!)], R, S, k+1)\ \ (4) \qquad (E[(\!|v|\!)], R, S, k+1) \to (E[v], R, S, k)$$
$$(E[\texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M'], R, S, k) \to (E[M'\{m/f\}], R \uplus \{m \mapsto \lambda x.M\{m/f\}\}, S, k)$$

Conditions: $(1): j = 1$ iff $i \neq 0$, $\quad (2): i = i_1 \oplus i_2$, $\quad (3): i \neq 0$, $\quad (4): R(m) = \lambda x.M$.

$$\textit{Values}\ \ v\ ::=\ m \mid i \mid () \mid \langle v, v\rangle \qquad \textit{Terms (extended)}\ \ M ::= \cdots \mid (\!|M|\!)$$
$$\textit{Eval. Contexts}\ \ E ::= \bullet \mid \texttt{assert}(E) \mid r := E \mid E \oplus M \mid v \oplus E \mid \langle E, M\rangle \mid \langle v, E\rangle \mid \pi_j E$$
$$EM \mid mE \mid \texttt{let}\ x = E\ \texttt{in}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid (\!|E|\!)$$

$$(\texttt{abstract}\ m; L, R, S, \mathcal{P}, \mathcal{A}) \xrightarrow{bld} (L, R, S, \mathcal{P}, \mathcal{A} \uplus \{m\})$$
$$(\texttt{public}\ m = \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) \xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S, \mathcal{P} \uplus \{m\}, \mathcal{A})$$
$$(m = \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) \xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S, \mathcal{P}, \mathcal{A})$$
$$(\texttt{global}\ r := i; B, R, S, \mathcal{P}, \mathcal{A}) \xrightarrow{bld} (B, R, S \uplus \{r \mapsto i\}, \mathcal{P}, \mathcal{A})$$
$$(\texttt{global}\ r := \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) \xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S \uplus \{r \mapsto m\}, \mathcal{P}, \mathcal{A})$$

Figure 2 Operational semantics (top); values and evaluation contexts (mid); library build (bottom).

Libraries are well typed if all their method and reference definitions are well typed (e.g. `public` $m = \lambda x.M$ is well typed if $m : \theta$ and $\lambda x.M : \theta$ are both valid for the same type θ) and only mention methods and references that are defined or abstract. A client L; `main` $= M$ is well typed if M : `unit` is valid and $L; m = \lambda x.M$ is well typed for some fresh x, m. A library/client is *open* if it contains abstract methods. This is different to open/closed terms: we call a term *open* if it contains free variables.

▶ Remark 1. By typing variable, reference and method names, we do not need to provide a context in typing judgements. Note that the references we use are of non-product type and, more importantly, **global** to the library: a term can use references but not create them locally or pass them as arguments (we discuss how to include such references in Appendix C).

▶ **Example 2.** The DAO-attack example from the Introduction can be written in HOLi as:

$$\texttt{abstract}\ \textit{send};\ \ \texttt{global}\ \textit{bal} := 100;$$
$$\texttt{public}\ \textit{wdraw} =$$
$$\lambda x.\ \texttt{if}\ !\textit{bal} \geq x\ \texttt{then}\ (\textit{send}(x); \textit{bal} := !\textit{bal} - x; \texttt{assert}(!\textit{bal} \geq 0))\ \texttt{else}\ ()$$

where $\textit{send}, \textit{wdraw} \in \texttt{Meths}_{\texttt{int},\texttt{unit}}$, $\textit{bal} \in \texttt{Refs}_{\texttt{int}}$, and $M; M'$ stands for `let` $\_ = M$ `in` M'.

A library contains public methods that can be called by a client. On the other hand, a client contains a main body that can be executed. These two scenarios constitute the operational semantics of HOLi. Both are based on evaluating (closed) terms, which we define next. Term evaluation requires: the closed term being evaluated; method definitions, provided by a method repository; reference values, provided by a store; and a call-depth counter (a natural number). Since method application is the only source of infinite behaviour in HOLi, bounding the depth of nested calls is enough to guarantee termination in program analysis. Hence we provide a mechanism to keep track of call depth.

The operational semantics is given in Figure 2. The evaluation of terms (top part) involves configurations of the form (M, R, S, k), where:

- M is a closed term which may contain *evaluation boxes*, i.e. points inside a term where a method call has been made and has not yet returned, and is taken from the syntax extending the one of Figure 1 with the rule: $M ::= \cdots \mid (\!|M|\!)$
- R is a *method repository*, i.e. a partial map from method names to their bodies
- S is a *store*, i.e. a partial map from reference names to their stored values
- k is a *counter*, i.e. a natural number.

Most of the rules are standard, but it is worth noting that lambdas are not values themselves but, rather, evaluate to method names that are freshly stored in the repository. Moreover, evaluation boxes interplay with the counter k in the semantics: they mark places where the depth has increased because of a nested call. The penultimate line of rules in the operational semantics keeps track of call depth, and illustrates the utility of evaluation boxes: making a call increases the counter and leaves behind an evaluation box; returning form the call removes the box and decreases the counter again.

A library L *builds* into a configuration of the form $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$, which includes its public methods according to the rules in Figure 2 (bottom). More precisely, R and S are as above, while $\mathcal{P}, \mathcal{A} \subseteq \texttt{Meths}$ are (disjoint) sets of *public* and *abstract* method names. We say that (a well typed) L builds to $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ if $(L, \emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{bld}{}^{*} (\varepsilon, R, S, \mathcal{P}, \mathcal{A})$. If L builds to $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ then the client L; $\texttt{main} = M$ builds to $(M, R, S, \mathcal{P}, \mathcal{A})$. Moreover, we can link libraries to clients and evaluate them, as in the following definition.

▶ **Definition 3.**

1. *Library L and client C are* compiatible *if L builds to some $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ and C builds to some $(M, R', S', \mathcal{P}', \mathcal{A}')$ such that: $\mathcal{P} \supseteq \mathcal{A}'$ and $\mathcal{A} \supseteq \mathcal{P}'$ (*complementation*); $\mathrm{dom}(S) \cap \mathrm{dom}(S') = \emptyset$ (*disjoint state*); and $\mathrm{dom}(R) \cap \mathrm{dom}(R') = \emptyset$ (*method ownership*).*
2. *For a library L, we let $\hat{L}$ be L with all its abstract method declarations and* `public` *keywords removed; and similarly for $\hat{C}$. Given compatible library L and client C, we let their* composition *be the client: L;$C = \hat{L}; \hat{C}$.*
3. *Given compatible L, C, the semantics of L;C is:*

$$[\![L;C]\!] = \{\rho \mid L;C \text{ builds to } (M, R, S, \emptyset, \emptyset) \wedge (M, R, S, 0) \to^* \rho\}$$

We say that $[\![L;C]\!]$ fails *if it contains some $(E[\texttt{assert}(0)], \cdots)$.*

▶ **Example 4.** To illustrate how libraries and clients are used, consider the DAO example again as a library $L_{\texttt{DAO}}$. We can define a client $C_{\texttt{atk}}$:

$\texttt{abstract}\ wdraw;\ \texttt{global}\ wlet := 0;$

$\texttt{public}\ send = \lambda x.wlet := !wlet + x;\ \texttt{if}\ !wlet < 100\ \texttt{then}\ wdraw(x)\ \texttt{else}\ ();$

$\texttt{main} = wdraw(1)$

to produce the following linked client $L_{\texttt{DAO}}$;$C_{\texttt{atk}}$ (modulo re-ordering):

$\texttt{global}\ bal := 100;\ \texttt{global}\ wlet := 0;$

$wdraw = \lambda x.\ \texttt{if}\ !bal \geq x\ \texttt{then}\ (send(x); bal := !bal - x; \texttt{assert}(!bal > 0))\ \texttt{else}\ ();$

$\texttt{public}\ send = \lambda x.wlet := !wlet + x;\ \texttt{if}\ !wlet < 100\ \texttt{then}\ wdraw(x)\ \texttt{else}\ ();$

$\texttt{main} = wdraw(1)$

We can see how $L_{\texttt{DAO}}$ is vulnerable to an attacker such as $C_{\texttt{atk}}$ after linking them. The aim is thus to use bounded analysis to find counterexamples that define clients such as this one.

$$\text{(INT)} \quad \frac{(M,R,S,k) \to (M',R',S',k')}{(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p \to (\mathcal{E},M',R',S',\mathcal{P},\mathcal{A},k')_p}$$

$$\text{(PQ)} \quad (\mathcal{E},E[mv],R,S,\mathcal{P},\mathcal{A},k)_p \xrightarrow{\texttt{call}(m,v)} ((m,E) :: \mathcal{E},0,R,S,\mathcal{P}',\mathcal{A},k)_o$$

$$\text{(OQ)} \quad (\mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o \xrightarrow{\texttt{call}(m,v)} ((m,l+1) :: \mathcal{E},mv,R,S,\mathcal{P},\mathcal{A}',k)_p \text{ if } R(m) = \lambda x.M$$

$$\text{(PA)} \quad ((m,l) :: \mathcal{E},v,R,S,\mathcal{P},\mathcal{A},k)_p \xrightarrow{\texttt{ret}(m,v)} (\mathcal{E},l,R,S,\mathcal{P}',\mathcal{A},k)_o$$

$$\text{(OA)} \quad ((m,E) :: \mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o \xrightarrow{\texttt{ret}(m,v)} (\mathcal{E},E[v],R,\mathcal{P},\mathcal{A}',k)_p$$

$$(\textbf{PC}):\ m \in \mathcal{A} \wedge \mathcal{P}' = \mathcal{P} \cup (\texttt{Meths}(v) \cap dom(R)),\ \ (\textbf{OC}):\ m \in \mathcal{P} \wedge \mathcal{A}' = \mathcal{A} \cup (\texttt{Meths}(v) \setminus dom(R)).$$

Figure 3 Trace semantics rules. Rules (PQ), (PA) assume the condition (**PC**), and similarly for (OQ),(OA) and (**OC**). $\texttt{Meths}(v)$ contains all method names appearing in v. INT stands for *internal* transition; PQ for *P-question* (i.e. call); PA for *P-answer* (i.e. return). Similarly for OQ and OA.

2.2 Trace Semantics

The semantics we defined only allows us to evaluate terms, and only so long as their method applications only involve methods that can be found in the repository R. We next extend this semantics to encompass libraries and terms that can also call abstract methods. The approach we follow is based on operational game semantics [18, 21, 16] and in particular the semantics is given by means of traces of method calls and returns (called *moves* in game semantics jargon), between the library and its client. In between such moves, the semantics evolves as the operational semantics we already saw.

To maintain a terminating analysis, we need to keep track of an added source of infinite execution, namely endless consecutive calls from an external component: a library will never terminate if its client keeps calling its methods. This leads us to a semantics with two counters, k and l, where k keeps track of internal nested method calls and l records the number of consecutive calls made from the external component. This counter l is orthogonal to k and is refreshed at every call to the external context.

When computing the semantics of a library, the library and its methods are the *Player (P)* of the computation game, while the (intended) client is the *Opponent (O)*. As the semantics is given in absence of an actual client, O actually represents every possible client. When computing the semantics of a client, the roles are reversed. In both cases, though, the same sets of rules is used and there is no need to specify who is P and O in the semantics.

The trace semantics uses *game configurations*, which are divided into *P-configurations* and *O-configurations* given respectively as:

$$(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p \quad \text{and} \quad (\mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o\,.$$

In a P-configuration, a term M is being evaluated – this is P's role. In an O-configuration, an external call has been made and the semantics waits for O to either return that call, or reply itself with another call. The components $M,R,S,\mathcal{P},\mathcal{A},k,l$ are as above, while $\mathcal{E}$ is an *evaluation stack*:

$$\mathcal{E} \ ::= \ \varepsilon \mid (m,E) :: \mathcal{E} \mid (m,l) :: \mathcal{E}$$

which keeps track of the computations that are on hold due to external calls. The trace semantics is generated by the rules given in Figure 3.

The formulation follows closely the operational game semantics technique. For example, from a P-configuration $(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p$, there are 3 options:

1. If M can make an internal reduction, i.e. in the operational semantics in context (R, S, k), then $(\mathcal{E}, M, R, S, \mathcal{P}, \mathcal{A}, k)_p$ performs this reduction (via (INT)).
2. If M is stuck at a method application for a method that is not in the repository R, then that method must be abstract (i.e. external) and needs to be called externally. This is achieved be issuing a call move and moving to an O-configuration (via (PQ)). The current evaluation context and the called method name are stored, in order to resume once the call is returned (via (OA)).
3. If M is a value and the evaluation stack is non-empty, then P has completed a method call that was issued by O (via (OQ)) and can now return (via (PA)).

On the other hand, from an O-configuration $(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o$, there are 2 options:

1. either return the last open method call (made by P) via (OA), or
2. call one of the public methods (from $\mathcal{P}$) using (OQ).

The role of conditions (PC) and (OC) is to ensure that each player calls the methods owned by the other, or returns their own, and update the sets of public and abstract names according to the method names passed inside v.

▶ Remark 5. The novelty of Figure 3 with respect to previous work on trace semantics for open libraries (e.g. [25]) lies in the use of l in order to bound the ability of O to ask repeated questions for finite analysis. The way rules (OQ) and (PA) are designed is such that any sequence of consecutive O-calls and P-returns has maximum length $2n$ if we bound l to n (i.e. $l \leq n$), as each such pair of moves increases l by 1. On the other hand, each P-call supplies to O a fresh counter ($l = 0$) to be used in contiguous (OQ)-(PA)'s. Thus, l can be seen as keeping track of the *insistence* of O in calling.

Finally, we can define the trace semantics of libraries.

▶ **Definition 6.** *Let L be a library. The semantics of L is :*

$$[\![L]\!] = \{(\tau, \rho) \mid (L, \emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{bld}{}^{*} (\varepsilon, R, S, \mathcal{P}, \mathcal{A}) \wedge (\varepsilon, 0, R, S, \mathcal{P}, \mathcal{A}, 0)_o \xrightarrow{\tau} \rho\}$$

We say that $[\![L]\!]$ fails *if it contains some $(\tau, (\mathcal{E}, E[\mathtt{assert}(0)], \cdots))$.*

▶ **Example 7.** Consider the DAO example as library $L_{\mathtt{DAO}}$ once again. Evaluating the game semantics we know the following sequence is in $[\![L_{\mathtt{DAO}}]\!]$. For economy, we hide $R, \mathcal{P}, \mathcal{A}$ and show only the top of the stack in the configurations. We also use $m(v)?$ and $m(v)!$ for calls and returns. We write S_i for the store $[bal \mapsto i]$.

$$\begin{aligned}
&(\varepsilon, 0, S_{100}, 0)_o \xrightarrow{wdraw(42)?} ((wdraw, 1), wdraw(42), S_{100}, 0)_p \\
&\quad \to^{*} ((wdraw, 1), E[send(42)], S_{100}, 1)_p \xrightarrow{send(42)?} ((send, E), 2, S_{100}, 1)_o \\
&\quad \xrightarrow{wdraw(100)?} ((wdraw, 1), wdraw(100), S_{100}, 1)_p \\
&\quad \to^{*} ((wdraw, 1), E'[send(100)], S_{100}, 2)_p \xrightarrow{send(100)?} ((send, E), 2, S_{100}, 2)_o \\
&\quad \xrightarrow{send(())!} ((wdraw, 1), E'[()], S_{100}, 2)_p \to^{*} ((wdraw, 1), (), S_0, 2)_p \\
&\quad \xrightarrow{wdraw(())!} ((send, E), 1, S_0, 2)_o \xrightarrow{send(())!} ((wdraw, 1), E[()], S_0, 1)_p \\
&\quad \to^{*} ((wdraw, 1), E[\mathtt{assert}(-42 \geq 0)], S_{-42}, 1)_p
\end{aligned}$$

This transition sequence is an instance of the symbolic trace provided in the Introduction. Here, a call is made with parameter 42, and a reentrant call with 100, which leads to the assertion violation $\mathtt{assert}(-42 \geq 0)$. Note that a bound of $k \leq 2$ is sufficient to find this assertion violation.

We next establish two focal properties of the trace semantics: bounding k and l ensures termination (Theorem 8), and that it is sound and complete with respect to library errors (Theorem 9). Notice Theorem 9 captures both soundness and completeness as it states that the game semantics eventually reaches every error that is concretely reachable for any client while finding only errors that can be reached concretely by a definable client.

▶ **Theorem 8** (Boundedness). *For any game configuration ρ, provided an upper bound k_0 and l_0 for call counters k and l, the labelled transition system starting from ρ is strongly normalising.*

Proof. For any transition sequence $\rho = \rho_0 \to \cdots \to \rho_i \to \ldots$ and each $i > 0$, we set the following two classes of configurations:

$$(A) = \{\rho_i \mid |\rho_i| < |\rho_{i-1}|\} \qquad (B) = \{\rho_i \mid \exists j < i-1.\ |\rho_i| < |\rho_j|\}$$

where $|\rho| = (k_0 - k, |M|, l_0 - l)$ is the *size* of ρ, and $|\rho| < |\rho'|$ is defined by the lexicographic ordering of the triple $(k_0 - k, |M|, l_0 - l)$, with bounds k_0 and l_0 such that $k \le k_0$ and $l \le l_0$ for semantic transitions to be applicable. If not present in the configuration, we look at the evaluation stack $\mathcal{E}$ to find the top-most missing component. In other words, opponent configurations will have size $(k_0 - k, |E|, l_0 - l)$ where E is the top-most one in $\mathcal{E}$, whereas proponent configurations will have size $(k_0 - k, |M|, l_0 - l)$ where l is the top-most one in $\mathcal{E}$.

We approach the proof in two steps: (1) we show that, for any transition sequence out of ρ, each reachable configuration belongs to (at least) one of the above classes; and (2) prove that the classes form a terminating sequence. For (1), considering all moves available to ρ, we have the following cases.

1. If $\rho \to \rho'$ is an (Int) move, we have two possibilities.
   a. For a transition $(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k+1)$, where $k+1 \le k_0$, we have a class (B) configuration since there must be a $(E[mv], R, S, k)$ such that $(E[mv], R, S, k) \to^* (E[v], R, S, k)$ which is lexicographically ordered since $|v| < |mv|$.
   b. Every other transition sequence is class (A) since they reduce the size of the term.
2. If $\rho \to \rho'$ is a (Pq) move, we have that ρ' is a class (A) configuration since $(k, |E|, l_0) < (k, |E[mv]|, l_0 - l)$ by lexicographic ordering.
3. If $\rho \to \rho'$ is an (Oa) move, we have a transition

   $$((m, E) :: \mathcal{E}, l, \ldots, k)_o \xrightarrow{ret(m,v)} (\mathcal{E}, E[v], \ldots, k)_p$$

   which must be a result of the prior proponent question, meaning $\mathcal{E}$ holds an l' on top. We thus have the following sequence

   $$(\mathcal{E}, E[mv], \ldots, k)_p \to^* (\mathcal{E}, E[v], \ldots, k)_o$$

   where $(k, |E[v]|, l) < (k, |E[mv]|, l')$, so ρ' is a class (B) configuration.
4. If $\rho \to \rho'$ is an (Oq) move, we have the transition

   $$\begin{aligned}(\mathcal{E}, l, \ldots, k)_o &\xrightarrow{call(m,v)} ((m, l+1) :: \mathcal{E}, mv, \ldots, k)_p \\ &\to ((m, l+1) :: \mathcal{E}, (\!|M\{v/x\}|\!), \ldots, k+1)\end{aligned}$$

   Simplifying the transition, we remove the configuration in between and take

   $$(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \xrightarrow{call(m,v)} ((m, l+1) :: \mathcal{E}, (\!|M\{v/x\}|\!), R, S, \mathcal{P}, \mathcal{A}, k+1)_p$$

   to be our new equivalent transition. We thus have that ρ' is a class (A) configuration since $(k_0 - (k+1), |(\!|M\{v/x\}|\!)|, l_0 - (l+1)) < (k_0 - k, |E|, l_0 - l)$ by lexicographic ordering.

5. If $\rho \to \rho'$ is a (PA) move, we have the transition

$$((m,l) :: \mathcal{E}, v, \dots, k)_p \xrightarrow{ret(m,v)} (\mathcal{E}, l, \dots, k)_o$$

which must be the result of a prior opponent question

$$(\mathcal{E}, l+1, \dots, k)_o \xrightarrow{call(m,v)} ((m,l) :: \mathcal{E}, (\!|M\{v/x\}|\!), \dots, k+1)_p$$
$$\to^* ((m,l) :: \mathcal{E}, (\!|v|\!), \dots, k+1)_p \to ((m,l) :: \mathcal{E}, v, \dots, k)_p \xrightarrow{ret(m,v)} (\mathcal{E}, l, \dots, k)_o$$

where E' is the topmost evaluation context in $\mathcal{E}$. We thus have that $(k_0 - k, E', l_0 - l) < (k_0 - k, E', l_0 - (l+1))$, so ρ' is a class (B) configuration.

For (2), let us assume there is an infinite sequence

$$\rho_0 \to \dots \to \rho_j \to \dots \to \rho_i \to \dots$$

Since all reachable configurations fall into either (A) or (B) class, we know that the sequence must comprise only (A) and (B) configurations. In this infinite sequence, we know that all sequences of (A) configurations are in descending size, so (A) sequences cannot be infinite. We also observe that (B) configurations are padded with (A) sequences. For instance, if ρ_i is a (B) configuration, and ρ_j is its matching configuration, there may exist nested (B) configurations between ρ_j and ρ_i, as well as (A) sequences padding these.

Additionally, these (B) configurations can only occur as a return to a call, so we know they only occur together with the introduction of evaluation boxes $(\!|\bullet|\!)$. Since these brackets occur in pairs and are introduced in a nested fashion, we know $\mathcal{E}$ can only contain evaluation contexts with well-bracketed evaluation boxes, meaning that there cannot be interleaved sequences of (B) configurations where their target configurations intersect. More specifically, the sequence

$$\rho_0 \to \dots \to \rho_j \to \dots \to \rho'_j \to \dots \to \rho_i \to \dots \to \rho'_i \to \dots$$

where ρ'_i matches ρ'_j and ρ_i matches ρ_j is not possible.

Now, ignoring all (A) and nested (B) sequences, we are left with an infinite stream of top-level (B) sequences which are also in descending order. Since starting size is finite, we cannot have an infinite stream of (B) sequences. Thus, the assumption that the sequence is infinite does not hold, meaning our semantics is strongly normalising. ◀

▶ **Theorem 9** (S and C). *We call a client* good *if it contains no assertions. For any library L, the following are equivalent:*

1. $[\![L]\!]$ *fails (reaches an assertion violation)*

2. *there exists a good client C such that $[\![L;C]\!]$ fails*

Proof. 1 to 2: Suppose now that $(\tau, \rho) \in [\![L]\!]$ for some trace τ and failed ρ. By Theorem 11, we have that there is a good client C realising the trace τ. So then, by Lemma 10, we have that $[\![L;C]\!]$ fails.

2 to 1: Suppose $[\![L;C]\!]$ fails for some good client C. Then, by Lemma 10, there are τ, ρ, ρ' such that $(\tau, \rho) \in [\![L]\!]$, $(\tau, \rho') \in [\![C]\!]$, and ρ is failed (i.e. is of the shape $(\mathcal{E}, E[\texttt{assert}(0)], \cdots)$). ◀

The latter relies on an auxiliary lemma (well-composing of libraries and clients), and a definability result akin to game semantics definability arguments (see Appendix D).

▶ **Lemma 10** (L-C Compositionality). *For any library L and compatible good client C, $[\![L;C]\!]$ fails if and only if there exist $(\tau_1,\rho_1) \in [\![L]\!]$ and $(\tau_2,\rho_2) \in [\![C]\!]$ such that $\tau_1 = \tau_2$ and $\rho_1 = (\mathcal{E}, E[\texttt{assert}(0)], \cdots)$.*

▶ **Theorem 11** (Definability). *Let L be a library and $(\tau,\rho) \in [\![L]\!]$. There is a good client C compatible with L such that $(\tau,\rho') \in [\![C]\!]$ for some ρ'.*

3 Symbolic Semantics

Checking libraries for errors using the semantics of the previous section is infeasible, even when the traces are bounded in length, as ground values are concretely represented. In particular, integer values provided by O as arguments to calls or return values range over all integers. The typical way to mitigate this limitation is to execute the semantics symbolically, using symbolic variables for integers and path conditions to bind these variables to plausible values. We use this technique to devise a symbolic version of the trace semantics, corresponding to a symbolic execution which will enable us in the next sections to introduce a practical method and implementation for checking libraries for errors. The symbolic semantics is fully formal, closely following the developments of the previous section, and allows us to prove a strong form of correspondence between concrete and symbolic semantics (a bisimulation).

Apart from integers, another class of concrete values provided by O are method names. For them, the semantics we defined is symbolic by design: all method names played by O are going to be fresh and therefore picking just one of those fresh choices is sufficient (formally speaking, the semantics lives in nominal sets [31]). The reason why using fresh names for methods played by O is sound is that the effect of O calling a higher-order public method with an argument m (where m is another public method), and with $\lambda x.mx$, is equivalent as far as reachability of an error is concerned. In the latter case, the client semantics would create a fresh name m', bind it to $\lambda x.mx$, and pass m' as an argument. We therefore just focus on this latter case.

The symbolic semantics involves terms that may contain symbolic values for integers. We therefore extend the syntax for values and terms to include such values, and abuse notation by continuing to use M to range over them. We let $\texttt{SInts}$ be a set of symbolic integers ranged over by κ and variants, and define:

$$\begin{aligned} &\mathit{Sym.Values} \quad \tilde{v} \;::=\; m \mid i \mid () \mid \kappa \mid \tilde{v} \oplus \tilde{v} \mid \langle \tilde{v}, \tilde{v} \rangle \\ &\mathit{Sym.Terms} \quad M \;::=\; \cdots \mid \kappa \end{aligned}$$

where, in $\tilde{v} \oplus \tilde{v}$, not both $\tilde{v}$ can be integers. We moreover use a symbolic environment to store symbolic values for references, but also to keep track of arithmetic performed with symbolic integers. More precisely, we let σ be a finite partial map from the set $\texttt{SInts} \cup \texttt{Refs}$ to symbolic values. Finally, we use pc to range over program conditions, which will be quantifier-free first-order formulas with variables taken from $\texttt{SInts}$, and with $\top, \bot$ denoting true and false respectively.

The semantics for closed symbolic terms involves configurations of the form (M, R, σ, pc, k). Its rules include copies of those from Figure 2 (top) where the pc and σ are simply carried over. For example:

$$(E[\lambda x.M], R, \sigma, pc, k) \to_s (E[m], R \uplus \{m \mapsto \lambda x.M\}, \sigma, pc, k)$$

where m is fresh. On the other hand, the following rules directly involve symbolic reasoning:

$$(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \to_s (E[\texttt{assert}(0)], \sigma, pc \wedge (\kappa = 0), k)$$

$$(\widetilde{\text{INT}})\quad \frac{(M,R,\sigma,pc,k)\rightarrow_s (M',R',\sigma,pc',k')}{(\mathcal{E},M,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p\rightarrow_s(\mathcal{E},M',R',\mathcal{P},\mathcal{A},\sigma',pc',k')_p}$$

$$(\widetilde{\text{PQ}})\quad (\mathcal{E},E[m\tilde{v}],R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \xrightarrow{\texttt{call}(m,\tilde{v})}_s ((m,E)::\mathcal{E},0,R,\mathcal{P}',\mathcal{A},\sigma,k)_o$$

$$(\widetilde{\text{OQ}})\quad (\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o \xrightarrow{\texttt{call}(m,\tilde{v})}_s ((m,l+1)::\mathcal{E},m\tilde{v},R,\mathcal{P},\mathcal{A}',\sigma,pc,k)_p$$

$$(\widetilde{\text{PA}})\quad ((m,l)::\mathcal{E},\tilde{v},R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \xrightarrow{\texttt{ret}(m,\tilde{v})}_s (\mathcal{E},l,R,\mathcal{P}',\mathcal{A},\sigma,pc,k)_o$$

$$(\widetilde{\text{OA}})\quad ((m,E)::\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o \xrightarrow{\texttt{ret}(m,\tilde{v})}_s (\mathcal{E},E[\tilde{v}],R,\mathcal{P},\mathcal{A}',\sigma,pc,k)_p$$

($\widetilde{\mathbf{PC}}$) $m\in\mathcal{A}$ and $\mathcal{P}'=\mathcal{P}\cup(\texttt{Meths}(\tilde{v})\cap dom(R))$.

($\widetilde{\mathbf{OC}}$) $m\in\mathcal{P}$ and $(\tilde{v}',\mathcal{A}')\in\mathsf{symval}(\theta,\mathcal{A})$ where θ is the expected type of $\tilde{v}$. Moreover:

$$\texttt{symval}(\theta,\mathcal{A})=\begin{cases}\{((),\mathcal{A})\} & \text{if } \theta=\texttt{unit}\\ \{(\kappa,\mathcal{A}\uplus\{\kappa\})\mid \kappa \text{ is fresh in } dom(\sigma)\uplus\mathcal{A}\} & \text{if } \theta=\texttt{int}\\ \{(m,\mathcal{A}\uplus\{m\})\mid m \text{ is fresh in } dom(R)\uplus\mathcal{A}\} & \text{if } \theta=\theta_1\to\theta_2\\ \{(\langle\tilde{v}_1,\tilde{v}_2\rangle,\mathcal{A}_2)\mid(\tilde{v}_1,\mathcal{A}_1)\in\texttt{symval}(\theta_1,\mathcal{A}) & \text{if } \theta=\theta_1\times\theta_2\\ \qquad\qquad (\tilde{v}_2,\mathcal{A}_2)\in\texttt{symval}(\theta_2,\mathcal{A}_1)\} & \end{cases}$$

Figure 4 Symbolic trace semantics rules. Rules ($\widetilde{\text{PQ}}$), ($\widetilde{\text{PA}}$) assume the condition ($\widetilde{\mathbf{PC}}$), and similarly for ($\widetilde{\text{OQ}}$),($\widetilde{\text{OA}}$) and ($\widetilde{\mathbf{OC}}$). ($\widetilde{\text{OQ}}$),($\widetilde{\text{OA}}$) introduce $\tilde{v}$ and thus are non-deterministic.

$$(E[\texttt{assert}(\kappa)],R,\sigma,pc,k)\rightarrow_s(E[()],R,\sigma,pc\wedge(\kappa\neq 0),k)$$
$$(E[!r],R,\sigma,pc,k)\rightarrow_s(E[\sigma(r)],R,\sigma,pc,k)$$
$$(E[r:=\tilde{v}],R,\sigma,pc,k)\rightarrow_s(E[()],R,\sigma[r\mapsto\tilde{v}],pc,k)$$
$$(E[\tilde{v}_1\oplus\tilde{v}_2],R,\sigma,pc,k)\rightarrow_s(E[\kappa],R,\sigma\uplus\{\kappa\mapsto\tilde{v}_1\oplus\tilde{v}_2\},pc,k)\quad \text{where } \kappa \text{ is fresh}$$
$$(E[\texttt{if } \kappa \texttt{ then } M_1 \texttt{ else } M_0],R,\sigma,pc,k)\rightarrow_s(E[M_0],R,\sigma,pc\wedge(\kappa=0),k)$$
$$(E[\texttt{if } \kappa \texttt{ then } M_1 \texttt{ else } M_0],R,\sigma,pc,k)\rightarrow_s(E[M_1],R,\sigma,pc\wedge(\kappa\neq 0),k)$$

and where $\tilde{v}_1\oplus\tilde{v}_2$ is a symbolic value (for $i_i\oplus i_2$ the rule from Figure 1 applies).

We now extend the symbolic setting to the trace semantics. We define symbolic configurations for P and O respectively as:

$$(\mathcal{E},M,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \qquad\qquad (\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o$$

with evaluation stack $\mathcal{E}$, proponent term M, counters $k,l\in\mathbb{N}$, method repository R, public method name set $\mathcal{P}$, σ and pc as previously. The abstract name set $\mathcal{A}$ is now a finite subset of $\texttt{Meths}\cup\texttt{SInts}$, as we also need to keep track of the symbolic integers introduced by O (in order to be able to introduce fresh such names). The rules for the symbolic trace semantics are given in Figure 4. Note that O always refreshes names it passes. This is a sound overapproximation of all names passed for the sake of analysis.

Similarly to Definition 6, we can define the symbolic semantics of libraries.

▶ **Definition 12.** *Given library L, the symbolic semantics of L is:*

$$[\![L]\!]_s=\{(\tau,\rho)\mid(L,\emptyset,\emptyset,\emptyset,\emptyset)\xrightarrow{bld}{}^{*}(\varepsilon,R,S,\mathcal{P},\mathcal{A})$$
$$\wedge(\varepsilon,0,R,\mathcal{P},\mathcal{A},S,\top,0)_o\xrightarrow{\tau}_s\rho\ \wedge\ \exists\mathcal{M}.\,\mathcal{M}\vDash\rho(\sigma)^\circ\wedge\rho(pc)\}$$

where $\rho(\chi)$ is component χ in configuration ρ, and $\mathcal{M}$ is a model as defined in the next section. We say that $[\![L]\!]_s$ fails *if it contains some $(\tau,(\mathcal{E},E[\texttt{assert}(0)],\cdots))$.*

The symbolic rules follow those of the concrete semantics, the biggest change being the treatment of symbolic values played by O. Condition $(\widetilde{\mathbf{OC}})$ stipulates that O plays distinct fresh symbolic integers as well as fresh method names, in each appropriate position in $\tilde{v}$, and all these names are included in the set $\mathcal{A}$.

▶ **Example 13.** As with Example 7, we consider the DAO attack. Running the symbolic semantics, we find the following minimal class of errors. We write $\sigma_{\tilde{v}}$ for a symbolic environment $[bal \mapsto \tilde{v}]$.

$$
\begin{aligned}
&(\varepsilon, 2, \sigma_{100}, k_0)_o \xrightarrow{wdraw(\kappa_1)?} ((wdraw, 1), wdraw(\kappa_1), \sigma_{100}, 2)_p \\
&\quad \to^* ((wdraw, 1), E[send(\kappa_1)], \sigma_{100}, 1)_p \xrightarrow{send(\kappa_1)?} ((send, E), 2, \sigma_{100}, 1)_o \\
&\quad \xrightarrow{wdraw(\kappa_2)?} ((wdraw, 1), wdraw(\kappa_2), \sigma_{100}, 1)_p \\
&\quad \to^* ((wdraw, 1), E'[send(\kappa_2)], \sigma_{100}, 0)_p \xrightarrow{send(\kappa_2)?} ((send, E), 2, \sigma_{100}, 0)_o \\
&\quad \xrightarrow{send(())!} ((wdraw, 1), E'[()], \sigma_{100}, 0)_p \\
&\quad \to^* ((wdraw, 1), (), \sigma_{100-\kappa_2}, 0)_p \xrightarrow{wdraw(())!} ((send, E), 1, \sigma_{100-\kappa_2}, 0)_o \\
&\quad \xrightarrow{send(())!} ((wdraw, 1), E[()], \sigma_{100-\kappa_2}, 1)_p \\
&\quad \to^* ((wdraw, 1), E[\texttt{assert}(!bal \geq 0)], \sigma_{100-\kappa_2-\kappa_1}, 1)_p
\end{aligned}
$$

For this to be a valid error, we require $(\kappa_1, \kappa_2 \leq 100) \wedge (100 - \kappa_2 - \kappa_1 < 0)$ to be satisfiable. Taking assignment $\{\kappa_1 \mapsto 100, \kappa_2 \mapsto 1\}$, we show the path is valid.

3.1 Soundness

The main result of this section is establishing the soundness of the symbolic semantics: a trace and a specific configuration can be achieved symbolically iff they can be achieved concretely as well. In fact, we will need to quantify this statement as, by construction, the symbolic semantics requires O to always place fresh method names, whereas in the concrete semantics O is given the freedom to play old names as well. What we show is that the symbolic semantics corresponds (via *bisimilarity*) to a restriction of the concrete semantics where O plays fresh names only. This restriction is sound, in the sense that it is sufficient for identifying when a configuration can fail. We make this precise below.

A ***model*** $\mathcal{M}$ is a finite partial map from symbolic integers to concrete integers. Given such an $\mathcal{M}$ and a formula ϕ, we define $\mathcal{M} \models \phi$ using a standard first-order logic interpretation with integers and arithmetic operators (in particular, we require that all symbolic integers in ϕ are in the domain of $\mathcal{M}$). Moreover, for any symbolic term M (or trace, move, etc.), we denote by $M\{\mathcal{M}\}$ the concrete term we obtain by substituting any symbolic integer κ of M with its corresponding concrete integer $\mathcal{M}(\kappa)$. Finally, given a symbolic environment σ, we define its formula representation σ° recursively by:

$$\emptyset^\circ = \top, \quad (\sigma \uplus \{r \mapsto v\})^\circ = \sigma^\circ, \quad (\sigma \uplus \{\kappa \mapsto v\})^\circ = \sigma^\circ \wedge (\kappa = v).$$

We now define notions for equivalence between symbolic and concrete configurations. Let $\mathcal{M}$ be a model. For any concrete configuration $\rho = (\mathcal{E}, \chi, R, S, \mathcal{P}, \mathcal{A}, k)$ and symbolic configuration $\rho_s = (\mathcal{E}', \chi', R', \mathcal{P}', \mathcal{A}', \sigma, pc, k')$, we say they are *equivalent in* $\mathcal{M}$, written $\rho =_{\mathcal{M}} \rho_s$, if:

- $(\mathcal{E}, \chi, R) = (\mathcal{E}', \chi', R')\{\mathcal{M}\}, \mathcal{P} = \mathcal{P}', \mathcal{A} = \mathcal{A}' \cap \texttt{Meths}$ and $S = (\sigma \restriction \texttt{Refs})\{\mathcal{M}\}$;
- $\mathrm{dom}(\mathcal{M}) = (\mathcal{A}' \cup \mathrm{dom}(\sigma)) \cap \texttt{SInts}$ and $\mathcal{M} \vDash pc \wedge \sigma^\circ$.

The notion of equivalence we require between concrete configurations and their symbolic counterparts is behavioural equivalence, modulo O playing fresh names.

More precisely, a transition $\rho \xrightarrow{\chi} \rho'$ is called *O-refreshing* if, when ρ is an O-configuration and $\chi = \texttt{call}/\texttt{ret}(m, v)$ then all names in v are fresh and distinct. A set $\mathcal{R}$ with elements of the form $(\rho, \mathcal{M}, \rho_s)$ is a ***bisimulation*** if, whenever $(\rho, \mathcal{M}, \rho_s) \in \mathcal{R}$, written $\rho\, \mathcal{R}_{\mathcal{M}}\, \rho_s$ then $\rho =_{\mathcal{M}} \rho_s$ and, using χ to range over moves and ε (i.e. no move):

- if $\rho \xrightarrow{\chi} \rho'$ is O-refreshing then there exists $\mathcal{M}' \supseteq \mathcal{M}$ such that $\rho_s \xrightarrow{\chi_s}_s \rho_s'$, with $\chi = \chi_s\{\mathcal{M}'\}$, and $\rho' \mathcal{R}_{\mathcal{M}'} \rho_s'$;
- if $\rho_s \xrightarrow{\chi}_s \rho_s'$ then there exists $\mathcal{M}' \supseteq \mathcal{M}$ such that $\rho \xrightarrow{\chi\{\mathcal{M}'\}}_G \rho'$ and $\rho' \mathcal{R}_{\mathcal{M}'} \rho_s'$.

We let $\sim$ be the largest bisimulation relation: $\rho \sim_{\mathcal{M}} \rho_s$ iff there is bisimulation $\mathcal{R}$ such that $\rho \mathcal{R}_{\mathcal{M}} \rho_s$.

We can show that concrete and symbolic configurations are bisimilar.

▶ **Lemma 14.** *Given ρ, ρ_s a concrete and symbolic configuration respectively, and $\mathcal{M}$ a model such that $\rho =_{\mathcal{M}} (\rho')$, we have $\rho \sim_{\mathcal{M}} \rho_s$.*

Proof (sketch). We show that $\{(\rho, \mathcal{M}, \rho') \mid \rho =_{\mathcal{M}} \rho'\}$ is a bisimulation. ◀

Next, we argue that O-refreshing transitions suffice for examining failure of concrete configurations. Indeed, suppose τ is a trace leading to fail, and where O plays an old name m in argument position in a given move. Then, τ can be simulated by a trace τ' that uses a fresh m' in place of m. If m is an O-name, we obtain τ' from τ by following exactly the same transitions, only that some P-calls to m are replaced by calls to m' (and accordingly for returns). If, on the other hand, m is a P-name, then the simulation performed by τ' is somewhat more elaborate: some internal calls to m will be replaced by P-calls to m', immediately followed by the required calls to m (and dually for returns).

▶ **Lemma 15** (O-Refreshing)**.** *Let ρ be a concrete configuration. Then, ρ fails iff it fails using only O-refreshing transitions.*

With the above, we can prove soundness.

▶ **Theorem 16** (Soundness)**.** *For any L, $[\![L]\!]$ fails iff $[\![L]\!]_s$ fails.*

Proof. Lemma 14 implies that $[\![L]\!]_s$ fails iff $[\![L]\!]$ fails with O-refreshing transitions, which in turns occurs iff $[\![L]\!]$ fails, by Lemma 15. ◀

3.2 Bounded Analysis for Libraries

Definition 12 states how the symbolic trace semantics can be used to independently check libraries for errors. As with the trace semantics in Definition 6, this is strongly normalising when given an upper limit to the call counters. As such, $[\![L]\!]_s$ with counter bounds $k_0, l_0 \in \mathbb{N}$, for k, l respectively, defines a finite set (modulo selecting of fresh names) of reachable valid configurations within $k \leq k_0, l \leq l_0$, where validity is defined by the satisfiability of the symbolic environment σ and the path condition pc of the configuration reached. By virtue of Theorems 9 and 16, every valid reachable configuration that is failed (evaluates an invalid assertion) is realisable by some client. And viceversa.

Given a library L, taking $\mathcal{F}[\![L]\!]_s$ to be all reachable final configurations, we have the exhaustive set of paths L can reach. In $\mathcal{F}[\![L]\!]_s$, every failed configuration (τ, ρ), i.e. such that ρ holds a term $E[\texttt{assert}(0)]$, defines a reachable assertion violation, where τ is a true counterexample. Hence, to check L for assertion violations it suffices to produce a finite representation of the set $\mathcal{F}[\![L]\!]_s$. One approach is to bound the depth of analysis by setting an

Table 1 Table recording performance of HOLiK on our benchmarks.

	$l \leq 1$	$l \leq 2$	$l \leq 3$
$k \leq 2$	226/70/45 (555s)	5708/60/44 (4710s)	9656/3/23 (12471s)
$k \leq 3$	1254/67/51 (1475s)	4092/27/18 (13482s)	4187/17/12 (16649s)
$k \leq 4$	3392/63/48 (3180s)	3069/19/14 (15903s)	1335/12/10 (17765s)
$k \leq 5$	3659/57/45 (4787s)	895/15/10 (16757s)	215/11/9 (17796s)

$a/b/c$ (d) for a traces found in b successful runs taking d seconds in total where c out of 59 unsafe files were found to have bugs, per bound.
59 of 59 unsafe files found to have bugs over the various bounds checked

upper bound to the call counters, using a name generator to make deterministic the creation of fresh names, and then exhaustively search all final configurations for failed elements. In the following section we implement this routine and test it.

4 Implementation and Experiments

We implemented the syntax and symbolic trace semantics (symbolic games) for HOLi in the $\mathbb{K}$ semantic framework [32] as a proof of concept, and tested it on 70 sample libraries.[1] Using $\mathbb{K}$'s option to exhaustively expand all transitions, $\mathbb{K}$ is able to build a closure of all applicable rules. By providing a bound on the call counters, we produce a finite set of all reachable valid symbolic configurations up to the given depth (equivalent to finding every valid $\rho \in \mathcal{F}[\![L]\!]_s$) which thus implements our bounded symbolic execution.

We wrote and adapted examples of coding errors into a set of 70 sample libraries written in HOLi, totalling 6,510 lines of code (LoC). Examples adapted from literature include: reentrancy bugs from smart contracts [3, 23]; variations of the "awkward example" [30]; various programs from the MoCHi benchmark [35]; and simple implementations related to concurrent programming (e.g. flat combining and race conditions) where errors may occur in a single thread due to higher-order behaviour. We also combined several libraries, by concatenating refactored method and reference definitions, to generate larger libraries that are harder to solve. Combined files range from 150 to 520 LoC.

We ran HOLiK on all sample libraries, lexicographically increasing the bounds from $k \leq 2, l \leq 1$ to $k \leq 5, l \leq 3$ (totalling 78,120 LoC checked), with a timeout set to five minutes per library. We start from $k \leq 2$ because it provides the minimum nesting needed to observe higher-order semantics. All experiments ran on an Ubuntu 19.04 machine with 16GB RAM, Intel Core i7 3.40GHz CPU, with intermediate calls to Z3 to prune invalid configurations. Per bound, the number of counterexamples found, the time taken in seconds, and the execution status, i.e. whether it terminated or not, are recorded in Table 1.

We can observe that independently increasing the bounds for k and l causes exponential growth in the total time taken, which is expected from symbolic execution. Note that the time tends towards 21000 seconds because of the timeout set to 5 minutes for 70 programs. In particular, while the number of errors found grows exponentially with respect to the increase in bounds – which is due to the exponential growth in paths – this trend does not continue indefinitely because programs start timing out without reporting any errors as the bounds

[1] The tool and its benchmarks can be found at: `https://github.com/LaifsV1/HOLiK`.

grow. With bounds $k \leq 2$ and $l \leq 1$, all 70 programs in our benchmark were successfully analysed, though not all minimal errors were found until the bounds were increased further. Cumulatively, all unsafe programs in our benchmark were correctly identified.

While the table may suggest that increasing bound for l is more beneficial than that for k, the number of errors reported does not imply every trace is useful. For instance, increasing the bound for l can lead to errors re-merging in a higher-order version, which suggests potential gain from a partial order reduction. Overall, the k and l counters are incomparable as they keep track of different behaviours. Finally, since HOLiK was able to handle every file and correctly identified all unsafe files in the benchmark, we conclude that HOLiK, as a proof of concept, captures the full range of behaviours in higher-order libraries. Results suggest that the tool scales up to at least medium-sized programs (<1000 LoC), which is promising because real-world medium-size higher-order programs have been proven infeasible to check with standard techniques (e.g. the DAO withdraw contract was approximately 100 LoC).

5 Related Work

Game semantics techniques have been applied to program equivalence verification by reducing program equivalence to language equivalence in a decidable automata class [15, 1]. Equivalence tools can be used for reachability but, as they perform full verification, they can only cover lower-order recursion-free language fragments. For example, the Coneqct [24] tool can verify the simplified DAO attack, but cannot check higher-order or recursive functions (e.g. the "file lock" and "flat combiner" examples), and operates on integers concretely. Close to our approach is also Symbolic GameChecker [11], which performs symbolic model checking by using a representation of games based on symbolic finite-state automata. The tool works on recursion-free Idealized Algol with first-order functions, which supports only integer references. On the other hand, it is complete (not bounded) on the fragment that it covers.

Besides games techniques, a recent line of work on verification of contracts in Racket [27, 26] is the work closest to ours. Racket contracts exist in a higher-order setting similar to ours, and generalise higher-order pre and post conditions, and thus specify safety. To verify these, [27] defines a symbolic execution based on what they call "demonic context" in prior work [38]. This either returns a symbolic value to a call, or performs a call to a known method within some unknown context, thus approximating all the possible higher-order behaviours, and is equivalent to the role the opponent plays in our games. In [26], the technique is extended to handle state, and finitised for total verification. The approaches are notionally similar to ours, since both amount to Symbolic Execution for an unknown environment. In substance, the techniques are very different and in particular ours is based on a semantics theory which allows us to obtain compositionality and definability results, which are not proven for [26] and proven for [27] only in a stateless setting. On the other hand, Racket contracts can be used for richer verification questions than assertion violations. In terms of tool performance, we provide a comparison of the techniques in Appendix B.

Another relevant line of work is that of verifying programs in the Ethereum Platform. Smart contracts call for techniques that handle the environment, with a focus on reentrancy. Tools like Oyente [23] and Majan [28] use pre-defined patterns to find bugs in the transaction order, but are not sound or complete. ReGuard [22] finds sound reentrancy bugs using a fuzzing engine to generate random transactions to check with a reentrancy automaton. In principle, it may detect reentrancy faster than symbolic execution (native execution is faster [40]), but, is incomplete even in a bounded setting. More closely related to our approach,

[17] considers the possibility of an *unknown* contract $c?$ calling a *known* contract $c*$ at each higher call level. This can be generalised in our game semantics as *abstract* and *public* names calling each other, but their focus is on modelling reentrancy, while we handle the full range of higher-order behaviours.

Like KLEE [4] and jCUTE [36], our implementation is a symbolic execution tool. These are generally able to find first-order counterexamples, but are unable to produce higher-order traces involving unknown code. Particularly, KLEE and jCUTE only handle symbolic calls provided these can be concretised. This partially models the environment, but calls are often impossible to concretise with libraries. The CBMC [6, 20] bounded model checking approach, which also bounds function application to a fixed depth, partially handle calls to unknown code by returning a non-deterministic value to such calls. This is equivalent to a game where only move available to the opponent is to answer questions. This restriction allows CBMC to find some bugs caused by interaction with the environment, but misses errors that arise from transferring flow of control (e.g. reentrancy). The typical BMC approach also misses bugs involving disclosure of names.

Higher-order model checking tools like MoCHi [35] are also related. MoCHi model checks a pure subset of OCaml and is based on predicate abstraction and CEGAR and higher-order recursion scheme model checkers. The modular approach [34] further extends this idea with modular analysis that guesses refinement intersection types for each top-level function. Although generally incomparable, HOLiK covers program features that MoCHi does not: MoCHi does not handle references and support for open code is limited (from experiments, and private communication with the authors).

6 Future Directions

Observing errors resurface deeper in the trace suggests the possibility of defining a partial order for our semantics to obtain equivalence classes for configurations and thus eliminate paths that involve known errors [29, 39]. Additionally, while k and l successfully bound infinite behaviour, a notion of bounding can be arbitrarily chosen. In fact, while we chose to directly bound the sources of infinite behaviour in method calls for simplicity of proofs and implementation, the theory does not prevent the generalisation of k and l as a monotonic cost function that bounds the semantics. It may also be worth considering the elimination of bounds entirely for the sake of unbounded verification. For this, one direction is abstract interpretation [9, 8], which amounts to defining overapproximations for values in our language to then attempt to compute a fixpoint for the range of values that assertions may take. However, defining and using abstract domains that maintain enough precision to check higher-order behaviours, such as reentrancy, is not a simple extension of the theory. Another direction, similar to Coneqct [24], is to define a push-down system for our semantics. Particularly, the approach in [24] is based on the decidability of reachability in fresh-register pushdown automata, and would require overapproximations for methods and integers. As with abstract interpretation, this would require defining abstract domains for methods and integers. While methods could be approximated using a finite set of names, as with k-CFA [37], an extension using integer abstract domains would need refinement to tackle reentrancy attacks. Finally, MoCHi [35] shows that it is possible to use CEGAR and higher-order recursion schemes for unbounded verification of higher-order programs. However, an extension of the MoCHi approach to include references and open code is not obvious.

References

1 S. Abramsky, D. R. Ghica, L. Ong, and A. Murawski. Algorithmic game semantics and component-based verification. In *Proceedings of SAVBCS 2003: Specification and Verification of Component-Based Systems, Workshop at ESEC/FASE 2003*, pages 66–74, 2003. published as Technical Report 03-11, Department of Computer Science, Iowa State University. URL: http://www.cs.iastate.edu/~leavens/SAVBCS/2003/papers/SAVCBS03.pdf.

2 Samson Abramsky and Guy McCusker. Game semantics. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 1–55, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

3 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, New York, NY, USA, 2017. Springer-Verlag New York, Inc. doi:10.1007/978-3-662-54455-6_8.

4 Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

5 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

6 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.

7 Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from AWS data centers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 467–486. Springer, 2018.

8 Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011. doi:10.1016/j.cl.2010.09.001.

9 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

10 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

11 Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014. doi:10.1016/j.tcs.2014.01.016.

12 Quinn Dupont. *Experiments in Algorithmic Governance: A history and ethnography of "The DAO," a failed Decentralized Autonomous Organization*, chapter 8, pages 157–177. Routledge, 2017.

13 William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *Software Engineering, IEEE Transactions on*, SE-3:266–278, August 1977. doi:10.1109/TSE.1977.231144.

14 Dan R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 17–26. IEEE Computer Society, 2009. doi:10.1109/LICS.2009.26.

15 Dan R. Ghica and Guy McCusker. Reasoning about idealized ALGOL using regular languages. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 2000. doi:10.1007/3-540-45022-X_10.

16 Dan R. Ghica and Nikos Tzevelekos. A system-level game semantics. *Electr. Notes Theor. Comput. Sci.*, 286:191–211, 2012. doi:10.1016/j.entcs.2012.08.013.

17 Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 51–78. Springer, 2018. doi:10.1007/978-3-319-96145-3_4.

18 A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 101–112, July 2002. doi:10.1109/LICS.2002.1029820.

19 James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975. doi:10.1145/390016.808444.

20 Daniel Kroening. The CBMC homepage. http://www.cprover.org/cbmc/, 2017. [Online; accessed 13-Jun-2017].

21 James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007. doi:10.1007/978-3-540-73420-8_58.

22 C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, May 2018.

23 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM. doi:10.1145/2976749.2978309.

24 Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. A contextual equivalence checker for IMJ*. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis*, pages 234–240, Cham, 2015. Springer International Publishing.

25 Andrzej S. Murawski and Nikos Tzevelekos. Higher-order linearisability. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, pages 34:1–34:18, 2017. doi:10.4230/LIPIcs.CONCUR.2017.34.

26 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *PACMPL*, 2(POPL):51:1–51:30, 2018. doi:10.1145/3158139.

27 Phuc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM

SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pages 446–456. ACM, 2015. `doi:10.1145/2737924.2737971`.

28 Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 653–663, New York, NY, USA, 2018. ACM. `doi:10.1145/3274694.3274743`.

29 Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. `doi:10.1007/3-540-56922-7_34`.

30 Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

31 Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013.

32 Grigore Roşu and Traian Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79:397–434, August 2010. `doi:10.1016/j.jlap.2010.03.012`.

33 Robert S. Boyer, Bernard Elspas, and Karl Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10:234–245, June 1975. `doi:10.1145/390016.808445`.

34 Ryosuke Sato and Naoki Kobayashi. Modular verification of higher-order functional programs. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 831–854. Springer, 2017. `doi:10.1007/978-3-662-54434-1_31`.

35 Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a scalable software model checker for higher-order programs. In Elvira Albert and Shin-Cheng Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*, pages 53–62. ACM, 2013. `doi:10.1145/2426890.2426900`.

36 Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

37 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991.

38 Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 537–554. ACM, 2012. `doi:10.1145/2384616.2384655`.

39 Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. `doi:10.1007/3-540-53863-1_36`.

40 Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761, Berkeley, CA, USA, 2018. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=3277203.3277260`.

A Motivating examples

Our file lock example provides a scenario where the library makes it possible for the client to update a file without first reacquiring the lock for it. The library contains an empty private method `updateFile` that simulates file access. The library also provides a public method `openFile`, which locks the file, allows the user to update the file indirectly, and then releases the lock.

```
import userExec :((unit → unit) → unit)
int lock := 0;
private updateFile(x:unit) :(unit) = { () };
public openFile (u:unit) :(unit) = {
  if (!lock) then ()
  else (lock := 1;
      let write = fun(x:unit):(unit) → (assert(!lock);updateFile())
      in userExec(write); lock := 0) };
```

The bug here is that `openFile` creates a `write` method, which it then passes to the client, via `userExec(write)`, to use whenever they want. This provides the client indirect access to the private method `updateFile`, which it can call without first acquiring the lock. Running this example in HOLiK we obtain the following minimal trace:

$$\begin{aligned}&call\langle openFile, ()\rangle \cdot call\langle userExec, m_2\rangle \cdot ret\langle userExec, ()\rangle\\&\quad \cdot ret\langle openFile, ()\rangle \cdot call\langle m_2, ()\rangle\end{aligned}$$

where m_2 is the method *name* generated by the library and bound to the variable `write`. This example serves as a representative of a class of bugs caused by revealing methods to the environment, a higher-order problem, in this case involving the second-order method `userExec` revealing m_2.

Next, we simulate double deallocation using a global reference `addr` as the memory address. The library defines private methods `alloc` and `free` to simulate allocation and freeing. The empty private method `doSthing` serves as a placeholder for internal computation that does not free memory.

```
1 import getInput :(unit → int)
2 int addr := 0; // 0 means address is free
3 private alloc (u:unit) :(unit) = {
4   if not(!addr) then addr := 1 else () };
5 private free (u:unit) :(unit) = {
6   assert(!addr); addr := 0 };
7 private doSthing (i:int) :(unit) = { () };
8 public run (u:unit) :(unit) = {
9   alloc(); doSthing(getInput ()); free() };
```

The error occurs in line 9, which calls the client method `getInput`. This passes control to the client, who can now call `run` again, thus causing `free` to be called twice. Executing the example on HOLiK, we obtain the following trace:

$$\begin{aligned}&call\langle run, ()\rangle \cdot call\langle getInput, ()\rangle \cdot call\langle run, ()\rangle \cdot call\langle getInput, ()\rangle\\&\quad \cdot ret\langle getInput, x_1\rangle \cdot ret\langle run, ()\rangle \cdot ret\langle getInput, x_2\rangle\end{aligned}$$

As with the DAO attack, this is a reentrancy bug.

Finally, we have an unsafe implementation of a flat combiner. The library defines two public methods: `enlist`, which allows the client to add procedures to be executed by the library, and `run`, which lets the client run all procedures added so far. The higher-order global reference `list` implements a list of methods.

```
1  private empty(x:int) : (unit) = { () };
2  fun list := empty;
3  int cnt := 0; int running := 0;
4  public enlist(f:(unit → unit)) :(unit) = {
5    if (!running) then ()
6    else
7      cnt := !cnt + 1;
8      (let c = !cnt in let l = !list in
9       list := (fun(z:int):(unit) → if (z == c) then f() else l(z)))};
10 public run(x:unit) :(unit) = {
11   running := 1;
12   if (0 < !cnt) then
13     (!list)(!cnt);
14     cnt := !cnt - 1; assert(not (!cnt < 0)); run()
15   else (list := empty; running := 0) };
```

The bug here is also due to a reentrant call in line 13. However, this is a much tougher example as it involves a higher-order reference `list`, a recursive method `run`, and a second-order method `enlist` that reveals client names to the library. With HOLiK, we obtain the following minimal counterexample:

$$\begin{aligned}&\mathit{call}\langle \mathit{enlist}, m_1\rangle \cdot \mathit{ret}\langle \mathit{enlist}, ()\rangle \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle m_1, ()\rangle \\ &\qquad \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle m_1, ()\rangle \cdot \mathit{ret}\langle m_1, ()\rangle \cdot \mathit{ret}\langle \mathit{run}, ()\rangle \cdot \mathit{ret}\langle m_1, ()\rangle\end{aligned}$$

where m_1 is a client name revealed to the library. In the trace above, `enlist` reveals the method m_1 to the library. This name is then added to the list of procedures to execute. In `run`, the library passes control to the client by calling m_1. At this point, the client is allowed to call `run` again before the list is updated.

B Comparison with Racket Contract Verification

We shall consider the latest version of the tool [26] since it handles state, which we refer to as SCV (Software Contract Verifier). A small benchmark (19 programs) based on HOLiK and SCV benchmarks was used for testing. Programs were manually translated between HOLi and Racket. Care was taken to translate programs whilst maintaining their semantics: contracts enforcing an input-output relation were translated into HOLi using wrapper functions that define the relation through an if statement. In the other direction, since contracts do not directly access references inside a term, stateful functions were translated from HOLi to return any references we wish to reason about.

Table 2 records the comparison. On one hand, HOLiK only found real errors, whereas SCV reported several spurious errors–a third of all errors were spurious. On the other hand, SCV was able to prove total correctness of 3 of the 7 safe files present. SCV also scales much better than HOLiK with respect to program size, which is in exchange of precision. The difference in time for small programs is mainly due to initialisation time. Subtle differences in the nature of each tool can also be observed. e.g., HOLiK reports 1 real error for `ack-simple-e`, whereas SCV reports 2 errors. The difference is because SCV takes into account constraints for integers (e.g. > 0 and $= 0$). More interestingly, for `various`,

Table 2 Comparison of HOLiK (left) and SCV (right). N/A is recorded for `ack` as in our attempts SCV crashed due to unknown reasons.

Program	LoC	Traces	Time (s)	LoC	Errors	Time (s)	False Errors
ack	17	0	6.0	9	N/A	2.4	N/A
ack-simple	13	0	6.5	9	0	2.4	0
ack-simple-e	13	1	6.5	9	2	2.5	0
dao	10	0	5.0	15	1	2.6	1
dao-e	16	1	5.5	15	1	2.7	0
dao-various	85	5	22.5	122	10	3.0	5
dao2-e	85	10	23.5	122	10	2.9	0
escape	9	0	5.0	9	0	2.6	0
escape-e	9	2	5.0	10	1	2.7	0
escape2-e	10	14	6.0	10	1	2.7	0
factorial	10	0	5.0	9	0	2.2	0
mc91	12	0	5.0	9	1	2.2	1
mc91-e	12	1	5.0	8	1	2.4	0
mult	14	0	5.0	11	2	2.7	2
mult-e	14	1	5.0	11	2	2.4	0
succ	7	0	5.0	7	1	2.5	1
succ-e	7	1	5.0	7	1	2.8	0
various	116	19	14.0	108	11	6.2	5
total	459	55	140.5	500	45	49.8	15

HOLiK reports 19 ways to reach assertion violations, whereas SCV reports only 6 real ways to violate contracts. The difference is because HOLiK reports paths through the execution tree that reach errors, whereas SCV reports a set of terms that may violate the contracts. For instance, independently safe methods A and B that may call an unsafe method C would be, from testing, reported as three valid traces ($call\langle A\rangle \cdot call\langle C\rangle$, $call\langle B\rangle \cdot call\langle C\rangle$ and $call\langle C\rangle$) by HOLiK. In contrast, SCV reports a single contract violation blaming C. Finally, `ack` failed to run on SCV due to unknown errors; Racket reported an error internal to the tool. Further testing proved the file is a valid Racket program that can be executed manually.

C ML-like References

HOLi has global higher-order references. These are enough for coding all of our examples and, moreover, allow us to prove completeness (every error has a realising client). We here present a sketch of how games can be extended with (locally created, scope extruding) ML-like references, following e.g. [21, 16]. First, the following extension to types and terms are required.

$$\theta ::= \cdots \mid \mathtt{ref}\ \theta \qquad\qquad M ::= \cdots \mid !M \mid \mathtt{ref}\ M \mid M = M \qquad\qquad v ::= \cdots \mid r$$

The term $!M$ allows dereferencing terms M which evaluate to references, while $\mathtt{ref}\ v$ creates dynamically a fresh name $r \in \mathtt{Refs}_\theta$ (if $v : \theta$), and the semantic purpose is to update the store $S \uplus \{r \mapsto v\}$ when evaluating $\mathtt{ref}\ v$. Note that this allows us to store references to references, etc. Finally, the construct $M = M$ is for comparing references for name equality.

With terms handling general references concretely and symbolically, we extend game configurations with sets $\mathcal{L}_p, \mathcal{L}_o \subseteq \mathtt{Refs}$ that keep track of reference names disclosed by the proponent and opponent respectively. References being passed as values means that the client can update the references belonging to the client, and viceversa. When making a move, for each reference r they own that is passed, the proponent adds r to $\mathcal{L}_p$. Passing of names in a move can be done either by method argument and return value, but also via the common part of the store (i.e. via the references known to both players). Similarly, opponent passes names in their moves, which are added to $\mathcal{L}_o$. Concretely, when the opponent passes control, all references in $\mathcal{L}_p$ are updated with opponent values. Symbolically, the references r are updated with distinct fresh symbolic integers κ if $r \in \mathtt{Refs}_{\mathtt{Int}}$, distinct fresh method names if $r \in \mathtt{Refs}_{\theta_1\to\theta_2}$, or to arbitrary reference names if $r \in \mathtt{Refs}_{\mathtt{Refs}_\theta}$.

D Definability

In this section we show that every trace τ in the semantics of a library L has a corresponding good client that realises the same trace in its semantics.

Let L be a library with public names $\mathcal{P}$ and abstract names $\mathcal{A}$. Given a trace τ produced by L, with $\mathcal{P}'$ and $\mathcal{A}'$ respectively the public and abstract names introduced in τ, we set:

$$\begin{aligned}
\mathcal{N} &= \mathcal{P} \cup \mathcal{P}' \cup \mathcal{A} \cup \mathcal{A}' \\
\Theta_v &= \{\theta \mid \exists m \in \mathcal{N}.\ m : \theta' \wedge \theta \text{ a syntactic subtype of } \theta'\} \\
\Theta_m &= \{\theta \in \Theta \mid \theta \text{ a method type}\}
\end{aligned}$$

Note that the above sets are finite, since $\tau, \mathcal{P}, \mathcal{A}$ are finite. We assume a fixed enumeration of $\mathcal{N} = \{m_1, m_2, \cdots, m_n\}$. Moreover, for each type θ, we let $\mathbf{defval}_\theta$ be a default value, and $\mathbf{diverge}_\theta$ a term that on evaluation diverges by infinite recursion. We then construct a client $C_{\tau,\mathcal{P},\mathcal{A}}$ as in Figure 5.

The code is structured as follows.

1. We start off by defining global references:
   - cnt counts the number of P (Library) moves played so far;
   - $meth$ stores an index that records the move made by P: if the move was a return then $meth$ stores 0; if it was call to m_i then $meth$ stores i;
   - each ref_i will store the method $m_i \in \mathcal{P} \cup \mathcal{P}'$, either since the beginning (if $m_i \in \mathcal{P}$), or once P plays it (if $m_i \in \mathcal{P}'$);
   - each val_θ will be used for storing the value played by P in their last move.

   In the latter case above, there is a light abuse of syntax as θ can be a product type, of which HOLi does not have references. But we can in fact simulate references of arbitrary type by several HOLi references.
2. For each $m_i : \theta_1 \to \theta_2 \in \mathcal{A}$, we define a public method m_i that simulates the behaviour of O whenever m_i is called in τ:
   - it starts by increasing cnt, as a call to m_i corresponds to a P-move being played;
   - it continues by storing i and x in $meth$ and val_{θ_1} respectively;
   - it calls the private method $oracle$, which is tasked with simulating the rest of τ and storing the value that m_i will return in val_{θ_2};
   - it returns the value in val_{θ_2}.
3. For each $m_i : \theta_1 \to \theta_2 \in \mathcal{A}'$ we produce a method just like above, but keep it private (for the time being).
4. The method $oracle$ performs the bulk of the computations, by checking that the last move played by P was the expected one and selecting the next move to play (and playing it if is a call).
   - The oracle is called after each P-move is played, so it starts with increasing cnt.

```
global cnt := 0
global meth := 0
global ref_i := m_i              # for each m_i ∈ 𝒫
global ref_i := defval           # for each m_i ∈ 𝒫'
global val_θ := defval           # for each θ ∈ Θ_v
public m_i = λx.                 # for each m_i ∈ 𝒜
    cnt++; meth:=i; val_θ1:=x; oracle()
m_i = λx.                        # for each m_i ∈ 𝒜'
    cnt++; meth:=i; val_θ1:=x; oracle()
oracle = λ().
    match (!cnt) with       # number of P-moves played so far (max |τ|/2)
    | i →
      # if i > 0 and i-th P-move of τ is cr m_j(v), with m_j : θ1 → θ2, then
      # - if cr = ret then d = 0 and θ = θ2
      # - if cr = call then d = j and θ = θ1
      # diverge if the last P-move played is different from cr m_j(v)
      if not (!meth = d and !val_θ ≜_θ v) then diverge
      else for m_i in fresh(!val_θ) do ref_i := m_i
      # if (i+1)-th O-move of τ is cr' m_k(u), with m_k : θ1 → θ2, then
      # - if cr' = ret then c = 0
      # - if cr' = call then c = k
      if c then let x = (!ref_k)u in     # call m_k(u)
                cnt++; meth:=0; val_θ2:=x; oracle(); !val_θ2
      else val_θ2:=u                     # return u
main = oracle()
```

Figure 5 The client $C_{\tau,\mathcal{P},\mathcal{A}}$.

- It then performs a case analysis on the value of *cnt*, which above we denote collectively by assuming the value is i – this notation hides the fact that we have one case for each of the finitely many values of i.
  For each such i, the oracle first checks if the previous P-move (if there was one), was the expected one. If the move was a call, it checks whether the called method was the expected one (via an appropriate value of d), and also whether the value was the expected one. Value comparisons ($\triangleq_\theta$) only compare the integer components of θ, since we cannot compare method names. If this check is successful, the oracle extracts from u any method names played fresh by P and stores them in the corresponding ref_i.
  Next, the oracle prepares the next move. If, for the given i, the next move is a call, then the oracle issues the call, stores the return value of that call, increases *cnt* and recurs to itself – when the issued call returns, it would be through a P-move. If, on the other hand, the next move is a return, the oracle simply stores the value to be returned in the respective *val* reference – this would allow to the respective m_i to return that value.

5. The **main** method simply calls the oracle.

We can then show the following. For any library L and $(\tau, \rho) \in [\![L]\!]$, C_τ is such that $(\tau, \rho') \in [\![C_\tau]\!]$ for some ρ'.