Abstracting Concolic Execution for Soft Contract Verification

Bram Vandenbogaerde[1]✉, Quentin Stiévenart[2], and Coen De Roover[1]

[1] Vrije Universiteit Brussel, Belgium
{bram.vandenbogaerde,coen.de.roover}@vub.be
[2] Université du Québec à Montréal, Canada
stievenart.quentin@uqam.ca

Abstract. Design-by-contract programming is a best practice in which a contract is used to specify the expected behavior of program elements such as functions and classes. Although enforcing compliance with these contracts at run time renders a program robust against unexpected failures, they also introduce a substantial contract monitoring overhead. Soft contract verification intends to reduce this overhead by verifying as many contracts as possible through static analysis. Thus far, the static analyses underlying existing soft contract verifiers have been based on adaptations of the *AAM* technique for *CESK* machines. Intended solely to introduce the concept of soft contract verification, these purpose-built analyses lack configurability. In this paper, we propose a novel static analysis for soft contract verification called *abstract concolic execution*. We systematically abstract a concolic execution, which is a form of dynamic symbolic execution, into abstract concolic execution, rendering the technique terminating and sound for any program input. To show that our analysis is more configurable than the state-of-the-art analysis supporting soft contract verification, we propose two variations of the analysis. Finally, we show that our approach is comparable to if not more precise than the state of the art at the cost of performance. We find that in 10 out of the 24 benchmark programs, our approach is more precise than the state-of-the-art approach, while being as precise in 9 of them and less precise in 5.

1 Introduction

Design-by-contract [14] is a programming methodology where program elements (e.g., classes or functions) are annotated with *contracts*. These contracts usually encoded invariants or pre- and post-conditions on the program element. In the case of a function, its pre-conditions are usually about the arguments of the function, while its post-conditions are about its return value and potential side effects. Expanding upon the work of Meyer et al. [14], Felleisen [5] et al. propose a contract language for *higher-order* programming languages. A well-known implementation of their language can be found in *Racket*, where contracts are embedded and implemented in the same language as the elements they annotate. Unfortunately, as these contracts are often highly dynamic (e.g., depend on the function input, or change over time), they require *run-time contract checks*, resulting in a large performance penalty when executing the program.

Multiple approaches, collectively known as *soft contract verifiers* [17,16,24], have been proposed to verify as many contracts as possible before running the program. The first incarnation of this approach by Nguyen et al. [17] relies on *higher-order symbolic execution*, a calculus of opaque or *symbolic* values refined with predicates originating from contracts in the code. Unfortunately, their approach is not suitable for more complex programs that exhibit *side effects*. Thus, in follow-up work [16], Nguyen et al. propose a soft contract verifier that also takes side effects into account. The approach is based on the systematic abstraction of a concrete *CESK* machine, adding machinery for tracking symbolic variables and symbolic path constraints along the way.

More specifically, Nguyen et al. [16] extend the *CESK* machine with a *store cache* and *path constraint*. The store cache tracks *locally-precise* information about the in-scope variables by mapping variables to *post-values* which are combinations of abstract and symbolic values. The resulting abstract machine implements a form of *static symbolic execution* but renders it finite by carefully constructing and updating its store caches. Unfortunately, this renders the resulting analysis less *configurable* and precludes the application of common optimizations such as global store widening. This is because the store cache must be at a specific location in the state space and is governed by rules tailored to the analysed programming language.

Instead, we propose systematic abstraction of a *concolic execution* of the program to determine the reachability of contract violations. Concolic execution is a form of *dynamic symbolic execution* in which the program is executed concretely while keeping a symbolic representation alongside each program value. To this end, the program is instrumented to track each branching point in its execution as well as the conditions leading to that branching point. When the execution has terminated, the satisfiability of the conditions of the non-taken branches is checked, and a *model* is generated representing a mapping of symbolic variables to values that satisfy the selected condition. The program is executed again with the newly-obtained values. This process is repeated until all possible branches have been explored or until a time budget is exceeded. Unfortunately, concolic execution is not suitable for fully automated program verification as it is not guaranteed to terminate. This potentially leads to false negatives as it might fail to discover contract violations in the program. Abstracting a concolic execution engine solves this limitation by computing an over-approximation of the program behaviour, eliminating false negatives at the cost of introducing false positives.

In short, the contributions of this paper are as follows:

- We are the first to explore the idea of *abstract concolic execution* as an abstract interpretation of concolic execution for soft contract verification. We do so by systematically abstracting a $CESK_\varphi$ machine, a new variation on a *CESK* machine augmented with failure continuations and path constraints. We claim that this systematic abstraction results in a more configurable analysis compared to the state-of-the-art soft contract verification approaches. Furthermore, we demonstrate this claim by proposing two variations of the abstract machine.
- We formulate and prove a soundness and termination theorem for the resulting analysis, showing that the analysis terminates for any analysed program, and that its results are guaranteed to over-approximate the actual run-time behaviour of the program.
- Finally, we apply this novel analysis technique to the problem of *soft contract verification* [24,16,17]. Our analysis enables novel configurations of the resulting abstract machine, yielding different trade-offs between performance and precision.

In what follows we recall existing soft contract verification techniques and highlight their shortcomings. Next, we proceed by explaining why concrete concolic execution is not sufficient for the purposes of soft contract verification, and highlight challenges for its abstraction process. Next, we formalize a *concrete* version of a concolic execution engine using a variation on the *CESK* machine called the $CESK_\varphi$ machine. Then, we systematically abstract this machine by applying the *AAM* method [13] to obtain a finite and sound static analysis. We show that the resulting analysis is more configurable than state-of-the-art soft contract verification approaches by formalizing two variations of the abstract machine. We conclude with an evaluation of our approach by applying it to benchmark programs found in other related soft contract verification work.

2 State of the Art in Soft Contract Verification

In this paper we consider functional design-by-contract languages, more specifically the contract model introduced by Findler et al. [9] as implemented in Racket. Listing 1 depicts the `square` function and its contract. The contract stipulates that if the arguments of the function are numbers, the function's return value will be a positive number. Analysing the function's implementation carefully, one can deduce that this is indeed the case since squaring a number always results in a positive number.

Contracts are not only used in user-defined functions, but also for primitive functions provided by the host programming language. The multiplication operator, for instance, requires that its arguments are numbers. Failing to provide numbers as arguments results in a contract violation at run time. In the example depicted in Listing 1, however, such a contract violation is not possible because

Listing 1 A `square` function annotated with a contract stipulating that if the input is a number, the output will be a positive number.

```
def square(x: number?): positive? =
        return x*x
```

the contract on the argument of the function already requires that the argument is a number. Thus, the program would fail if the `square` function was not called with a number before reaching the multiplication operator. This results in more helpful error messages, as errors are reported at the earliest location where expectations are not met. The example also shows that the multiplication operator can never be called with an invalid argument, as the execution of the program halts before the execution of the body of `square`. Note that all paths to the multiplication operator have a `number?` constraint, highlighting the need for *path-sensitive* reasoning when verifying contract validity.

In general, contracts in functional programs can encode arbitrary predicates on the arguments and return value of a function. The contract system proposed by Findler et al. even allows for contracts on the return value of the function to be specified in terms of the arguments of the function, resulting in a *dependent contract*. This highlights the need for reasoning about arbitrary constraints on unknown (user) input.

Combining these needs naturally leads to *static symbolic execution* which represents unknown (user) input and subsequent operations on this input symbolically, and in tandem keeps track of a *path constraint* encoding the symbolic conditions necessary for a particular program state to be reached. These path constraints are updated whenever execution reaches a *branching point* (e.g., an *if statement*). Upon a branching point, a symbolic condition is added to the path constraint. Unfortunately, static symbolic execution is known to not always terminate such as when programs have an unbounded number of `input` statements, or whose `input` statements are under-constrained. Static symbolic execution is therefore unsuitable for verification purposes.

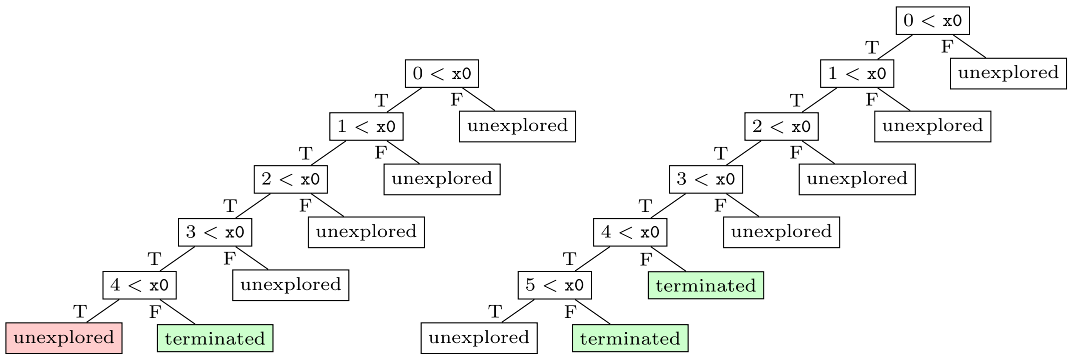
Nguyen et al. [16] instead propose *symbolic verification* which ensures a sound and terminating process for checking the reachability of contract violations. To this end, they abstract a *CESK* machine and extend the abstracted machine with a path constraint and a *store cache*. This store cache keeps track of symbolic information about the variables in scope. To ensure termination, Nguyen et al. only consider looping through recursion (i.e., through function calls) and carefully adapt the store cache to ensure termination. In the case of function calls, function arguments are replaced with symbolic variables corresponding to the names of the parameters of function. This ensures that a symbolic expression tree remains finite, but also discards potentially essential information from the caller of the function.

Summarising the discussion, we identify the following shortcomings of existing soft contract verification approaches:

- **Unnecessary precision loss** Simply discarding symbolic information from the caller of the function and replacing it with a symbolic variable for each parameter results in unnecessary precision loss. The caller of the function no longer controls which symbolic variables it passes to the callee. There is no longer a connection of the path constraint from the caller of the function with that of the callee.
- **Ad-hoc store cache management** Store caches represent *locally precise* information about the variables in scope of the current program state. Their contents and update rules are determined by analysis developers based on factors such as precision and termination. In the case of the state-of-the-art soft contract verification work, store caches are invalidated at function call boundaries, and whenever mutation of variables occurs. This ad-hoc management of the store cache makes common systematic optimizations such as global store widening more difficult to implement and formalize.

We address these shortcomings by proposing *abstract concolic execution*, a novel abstraction interpretation of concolic execution. In this setting, the abstract machine is no longer extended with a store-cache that collects locally precise information. Instead, we guarantee termination by properly abstracting symbolic variables and symbolic expressions. In the remainder of this paper we assume (as demonstrated in section 7), without loss of generality, that all contract checks can be translated to first-class `assert` statements.

3 From Concrete to Abstract Concolic Execution



(a) First iteration (b) Second iteration

Fig. 1: Execution tree after two concolic execution of the `square` program

In the listing below, we present a program that computes the square of a number through repeated additions of `x` coming from user input. A concolic execution engine executes this program by first generating a *concolic value*, consisting of a program value and symbolic variable, that substitutes the missing user input. In the remainder of this paper, symbolic variables are typeset with a monospace letter (usually `x`) followed by a unique number (e.g., `x0` for the first symbolic variable being generated). Next, the concolic execution engine executes the `square` function which contains a branching point, i.e., to stop the loop if `i >= x` and to continue whenever `i < x`. This is represented in a *symbolic execution tree* as a node created for each branching point, labelled with the conditions leading to the next branching point in the program. The edges of the tree are labelled with truth values for the conditions (T for true, F for false). Assuming that the initial input, named `x0`, has been arbitrarily chosen to be `4`, the symbolic execution tree grows to contain 5 branching points, one for each iteration of the loop. This execution is depicted in Figure 1a. The first execution eventually reaches the node highlighted in green, and then terminates.

```
def square(x) =
   y = 0
   i = 0
   while i < x do
      y = y+x
      i++
   return y

print(square(input()))
```

Next, the concolic execution engine selects an unexplored node in its execution tree (highlighted in red in Figure 1a), collects all constraints alongside the path leading up to that unexplored node and subsequently generates inputs (i.e., a model) satisfying these constraints. These inputs result into a new value for `x` used in the next execution of the program. In this case, the value `5` is chosen as the input for `x` in the next concolic execution (Figure 1b). Normally, this process is repeated until all nodes in the execution tree have been explored. Unfortunately, in this case the execution tree continues to grow indefinitely because different values for `x0` can continue to be computed and the exploration never finishes. Thus, in this example, the execution can only terminate after a set timeout has been reached rendering the program exploration incomplete. To summarize, this example illustrates a number of problems with dynamic symbolic execution:

– **Exploration of redundant states:** A concolic execution engine might explore nodes that are *behaviourally equivalent* to another node in the execution tree but differ in its path condition. In the example program, increasing the number of iterations of the loop does not yield any new interesting behaviour, yet their concolic execution states are considered different.
– **Non-termination:** The concolic execution approach is not guaranteed to terminate. This problem is illustrated in the example above as the execution tree keeps growing with increasing values for `x`. This results in an *unsound* analysis from a static analysis point of view, as it does not consider *all*

possible paths in the program. This is problematic for verification purposes as the program cannot be verified without considering all of its paths.

Abstract interpretation [4] could offer a solution to these problems by defining an abstraction for each component of the concolic execution engine. Abstract interpretation solves the first problem by abstracting concolic execution states, rendering identical those that do not differ in the property of interest. The second problem is solved by abstracting reoccurring subtrees into a graph shared between all parts of the tree in which the subtree occurs.

In this paper, we investigate whether this approach can be applied for abstracting concolic execution. This problem is challenging because a number of concolic execution aspects that need to be abstracted:

- **Symbolic representations:** A symbolic execution engine uses *symbolic representations* of program values in order to constrain them through the program's path condition. These symbolic representations are not necessarily finite (e.g, a loop containing an assignment $x = x + 1$, resulting in repeated suffixes of $+1$ in the symbolic expression). To guarantee termination without complex widening operators, an abstract interpretation of a concolic execution engine needs to abstract these symbolic representation so that a finite number of them is present at all times.
- **Symbolic variables:** In a concrete run of a concolic execution engine, the same input statement may be executed multiple times (e.g., in a loop), resulting in distinct concrete values and symbolic variables for each execution. Unfortunately, this is often a source of non-termination in a concolic execution engine. An abstract interpretation needs to abstract multiple concrete executions into a single abstract execution having a single abstract value with a corresponding abstract symbolic variable.
- **Path constraints:** Since the path constraint consists of first-order logic assertions over a number of symbolic expressions, the interpretation of the truth values of its abstract counterpart needs to account for abstract symbolic expressions too. Moreover, as infinite path constraints often give rise to non-terminating concolic execution engines, they need to be rendered finite for the resulting analysis to terminate without complex widening.
- **Model:** After each concrete symbolic execution run, a *model* is computed which maps symbolic variables to concrete values so that the next iteration follows the intended path to an unexplored node of the execution tree. Since abstract symbolic variables could correspond to multiple input statements, an abstract version of this model needs to map the statement's abstract symbolic variable to an abstract value that subsumes all possible invocations of the same input statement.

In this paper, we propose an abstraction of the $CESK_\varphi$ machine, a new variation of the CESK machine [8], by applying the *abstracting abstract machines* (or *AAM*) recipe [13]. We investigate whether this process results in an efficient and precise static analysis for the analysed program. And if so, what machine configurations work best in terms of precision and performance.

4 Concrete Concolic Execution

In this section we present a concrete version of a concolic execution engine. This concrete version largely follows the standard concolic execution semantics [20]. However, it does not explicitly model the execution as an execution tree. Instead, it relies on failure continuations to model backtracking and program re-execution. We do so in order to make the abstraction of the resulting machine easier. We formalize the concolic execution engine for a language λ_s. The semantics is defined as a small-step relation over a *CESK*φ machine, extending the standard *CESK* machine [8] with constraints and failure continuations.

4.1 Syntax

Figure 2 depicts the syntax of our language. The language is an *A-normal form* version of the λ-calculus, extended with support for if expressions, let expressions and input statements.

$$e ::= \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ ae\ e\ e \mid ae\ ae \mid \mathsf{input}$$

$$ae \in \mathit{Atomic} ::= n \mid b \mid x \mid \lambda x.e \qquad x \in \mathit{Identifier} \qquad n \in \mathbb{N} \qquad b \in \mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$$

Fig. 2: Syntax of λ_s

4.2 Semantics

Figure 3 depicts the state space of λ_s's semantics. Concolic values are represented by the *Value* sort. Such concolic values consist of a program value and symbolic expression of the *SymbolicValue* sort. This symbolic expression can be empty (denoted by $\emptyset$) to indicate that the corresponding program value does not have a suitable symbolic representation, such as closures. Addresses are represented by their allocation site and combined with a calling context consisting of a call-site history. The allocation site for a variable x is denoted as $\ell(x)$, assuming that an injective labelling function ℓ is defined for any program expression e. This choice for the address representation automatically yields a valid *address allocation* strategy where the new address is derived from the calling context and the label of the expression at the allocation-site. Addresses generated according to this allocation strategy are *unique* since each allocation site can repeat only when the program contains a loop. However, since the λ_s language does not contain any looping constructs, looping can only occur through function calls, meaning that our calling context provides a sufficient distinction from addresses of previous

$\varsigma \in \Sigma ::= \langle c, \sigma, \kappa, \psi, \mathsf{ctx}, M, \phi, V \rangle$	$c \in \mathit{Control} ::= \mathsf{ev}(e, \rho) \mid \mathsf{ap}(v)$
$\sigma \in \mathit{Store} = \mathit{Address} \mapsto \mathit{Value}$	$\kappa \in \mathit{Continuation} ::= \mathsf{let}(x, e, \rho) :: \kappa \mid \emptyset$
$\psi \in \mathit{FailContinuation} ::= \mathsf{branch}(\phi) :: \psi \mid \emptyset$	$M \in \mathit{Model} ::= \mathit{SymVar} \mapsto \mathit{ProgramVal}$
$\rho \in \mathit{Environment} = \mathit{Identifier} \mapsto \mathit{Address}$	$\alpha \in \mathit{Address} = \mathit{Label} \times \mathit{Context}$
$\mathsf{ctx} \in \mathit{Context} = \overline{\mathit{Label}}$	$v_p \in \mathit{ProgramVal} ::= n \mid b \mid (\lambda x.e, \rho)$
$s \in \mathit{SymbolicVal} ::= b \mid \emptyset \mid \mathtt{x} \mid \mathit{op}(s, \ldots, s)$	$\mathtt{xi} \in \mathit{SymVar} = \mathit{Address}$
$v \in \mathit{Value} = \mathit{ProgramVal} \times \mathit{SymbolicVal}$	$\phi, \varphi_1, \varphi_2 \in \mathit{Formula} ::= \varphi_1 \wedge \varphi_2 \mid s \mid \neg s$
$b \in \mathbb{B} ::= \mathsf{true} \mid \mathsf{false}$	*Label* is a program location

Fig. 3: State space of λ_s. A sequence is denoted by an overline, for instance, the set of sequences of elements $a \in A$ is denoted as $\overline{A}$.

loop iterations. Symbolic variables are treated identically to addresses but are generated by another meta-function called fresh.

The state space is derived from the possible program states which consist of the following components: a control (c), a store (σ), a continuation stack (κ), a failure continuation stack (ψ), a model (M), an unbounded calling context (ctx), a path condition (ϕ), and the set of visited branches. The control component is either an *ev*aluation state, or a continuation *ap*plication state. The former includes an environment ρ in which an expression e is to be evaluated. The latter includes the value that should be sent to the continuation on top of the continuation stack. The model consists of a mapping from addresses to program values. A lookup in this mapping is denoted as $M(x)$ where x denotes the address. If the mapping is undefined for some x a random value is returned instead.

Next, as depicted in below, we define an atomic evaluation function $[\![\cdot]\!]$ and a small-step relation $\rightsquigarrow$ (Figure 4). The atomic evaluation function is defined as expected. Literal numbers and boolean literals map to their respective concolic value, and λ expressions evaluate to closure values which do not have a symbolic counterpart. Variable references are evaluated to their respective concolic values by looking up their address and values from the environment and store.

$$
\begin{aligned}
[\![b]\!](\rho, \sigma) &= (b, b) & [\![n]\!](\rho, \sigma) &= (n, n) \\
[\![\lambda x.e]\!](\rho, \sigma) &= ((\lambda x.e, \rho), \emptyset) & [\![x]\!](\rho, \sigma) &= \sigma(\rho(x))
\end{aligned}
$$

The evaluation of function applications proceeds as usual. Rules [ST-Let1] and [ST-Let2] depict the semantics for introducing new lexical scopes. [ST-Let1] first evaluates the binding expression e_1 to a value by transitioning to an ev control state, and pushing a let continuation on the continuation stack. This continuation is applied in rule [ST-Let2] by binding the value of the evaluated expression to the variable x. Conditional expressions require more attention as they introduce additional constraints in the path condition. This is depicted by rule [ST-IfTrue] and [ST-IfFalse] which first evaluate the condition to a concolic value, check whether the concolic value is true or false, and evaluate the consequent and alternative branch accordingly. While doing so, the failure continuation stack is extended with a branch failure continuation, tracking which branch has not been taken. Metafunction in accepts four arguments: a path constraints, a visited set and two failure continuations. The metafunction returns the third argument whenever the path constraint is not in the visited set and returns the fourth argument otherwise. This ensures that the same branch is not repeatedly executed in subsequent concolic execution runs by checking whether the alternative path constraint is part of the visited set. Finally, rule [Input] reduces an input expression to either a new random value or to a value from the *model* M if one is defined in its mapping.

$$\langle \mathsf{ev}(ae, \rho), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \rightsquigarrow \langle \mathsf{ap}([\![ae]\!](\rho, \sigma), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \ \text{ST-Atomic}$$

$$\frac{\begin{array}{c}[\![ae]\!](\rho,\sigma) = (true, s) \qquad \varphi_t = \varphi \wedge s \\ \varphi_f = \varphi \wedge \neg s \qquad \psi' = \mathsf{in}(\varphi_f, V, \mathsf{branch}(\varphi_f) :: \psi, \psi)\end{array}}{\begin{array}{c}\langle \mathsf{ev}(\mathsf{if}\ ae\ e_1\ e_2, \rho), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \\ \rightsquigarrow \langle \mathsf{ev}(e_1, \rho), \sigma, \kappa, \psi', \mathsf{ctx}, M, \varphi_t, \{\varphi_f\} \cup V \rangle\end{array}} \ \text{ST-IfTrue}$$

$$\frac{\begin{array}{c}[\![ae]\!](\rho,\sigma) = (false, s) \qquad \varphi_t = \varphi \wedge s \\ \varphi_f = \varphi \wedge \neg s \qquad \psi' = \mathsf{in}(\varphi_t, V, \mathsf{branch}(\varphi_t) :: \psi, \psi)\end{array}}{\begin{array}{c}\langle \mathsf{ev}(\mathsf{if}\ ae\ e_1\ e_2, \rho), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \\ \rightsquigarrow \langle \mathsf{ev}(e_2, \rho), \sigma, \kappa, \psi', \mathsf{ctx}, M, \varphi_f, \{\varphi_t\} \cup V \rangle\end{array}} \ \text{ST-IfFalse}$$

$$\begin{array}{c}\langle \mathsf{ev}(\mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2, \rho), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \\ \rightsquigarrow \langle \mathsf{ev}(e_1, \rho), \sigma, \mathsf{let}(x, e_2, \rho) :: \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle\end{array} \ \text{ST-Let1}$$

$$\frac{\alpha = \mathsf{alloc}(\ell(x), \mathsf{ctx})}{\begin{array}{c}\langle \mathsf{ap}(v), \sigma, \mathsf{let}(x, e_2, \rho) :: \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \\ \rightsquigarrow \langle \mathsf{ev}(e_2, \rho[x \mapsto \alpha]), \sigma[\alpha \mapsto v], \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle\end{array}} \ \text{ST-Let2}$$

$$\frac{[\![ae_1]\!]\rho, \sigma = (\lambda x.e, \rho') \qquad [\![ae_2]\!](\rho, \sigma) = v \qquad \alpha = \mathsf{alloc}(\ell(x), \mathsf{ctx})}{\begin{array}{c}\langle \mathsf{ev}(ae_1\, ae_2, \rho), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \\ \rightsquigarrow \langle \mathsf{ev}(e, \rho'\ [x \mapsto \alpha]), \sigma[\alpha \mapsto v], \kappa, \psi, \ell(ae_1 ae_2) :: \mathsf{ctx}, M, \varphi, V \rangle\end{array}} \ \text{ST-App}$$

$$\frac{(v, \overline{v'}) = M(\mathtt{x}) \qquad \mathtt{x} = \mathsf{fresh}(\ell(\mathsf{input}), \mathsf{ctx})}{\begin{array}{c}\langle \mathsf{ev}(\mathsf{input}, \rho), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle \\ \rightsquigarrow \langle \mathsf{ap}((v, \mathtt{x})), \sigma, \kappa, \psi, \mathsf{ctx}, M, \varphi, V \rangle\end{array}} \ \text{Input}$$

$$\frac{M' = \mathsf{model}(\varphi)}{\langle \mathsf{ap}(v), \sigma, \emptyset, \mathsf{branch}(\varphi) :: \psi, \mathsf{ctx}, M, \varphi', V \rangle \rightsquigarrow \langle \mathsf{ev}(e_0, \rho_0), \sigma_0, \emptyset, \psi, \emptyset, M', \emptyset, V \rangle} \ \text{ST-Backtrack}$$

Fig. 4: Small-step semantics of the concolic execution for λ_s

Rule [ST-BACKTRACK] enables backtracking to program states that were not executed in the previous concolic execution. It does so by applying the failure continuation whenever the program terminates, and by restarting the program execution with an empty calling context and a model M' generated by meta-function model giving an assignment of symbolic variables to program values that satisfies the path constraint stored in the failure continuation. This corresponds to a depth-first search (*DFS*) of the concolic execution tree.

The complete concolic execution is defined as the least-fixed point of the small-step evaluation relation, as depicted by *Eval* and *Run*. Variables ρ_0 and σ_0 denote the initial environment and store respectively.

$$Eval(\Sigma) = \Sigma \cup \bigcup_{\substack{\varsigma \in \Sigma \\ \varsigma \rightsquigarrow \varsigma'}} \{\varsigma'\} \qquad Run(e_0) = lfp\ Eval\ \{\langle \mathsf{ev}(e_0, \rho_0), \sigma_0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}$$

5 Abstracting Concolic Execution

In this section we develop a sound over-approximation of the concolic execution semantics from the previous section. We do so by following the *AAM* [13] approach which proposes to abstract a concrete machine and its operational semantics in a component-wise manner. This results in a sound over-approximation of the concrete machine's semantics. While doing so, we establish a Galois connection between the concrete operational semantics and the abstract ones which we use for formulating and proving soundness of the resulting static analysis.

5.1 Preliminaries

We first present some standard notions of abstract interpretation required for the remainder of this section.

Definition 5.1.1. *A join-semilattice, denoted $\langle A, \leq, \sqcup \rangle$ is a partially ordered set $\langle A, \leq \rangle$ with a least-upper bound $\sqcup$ defined for every two elements in the set.*

Definition 5.1.2. *A* ***Galois connection*** *between two partially ordered sets A and B is a pair of functions α and γ, denoted as $\langle A, \leq_A \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle B, \leq_B \rangle$, such that:*

$$\forall a \in A : a \leq_A \gamma(\alpha(a))$$

Definition 5.1.3. *An abstract function $\hat{f}$ is said to be sound with respect to its concrete counterpart f iff:*

$$\forall a \in \hat{A} : \alpha_{\hat{B}}(f(a)) \leq_{\hat{B}} \hat{f}(\alpha_{\hat{A}}(a))$$

where α_x is the abstraction function corresponding to the partially ordered set x.

In what follows, we will overload the notation by using the same α and γ functions for different value types whenever no ambiguities can arise. In ambiguous contexts, a subscript will be used to differentiate the intended function from other potential candidates.

5.2 Standard Component-Wise Abstractions

Before abstracting the components of our abstract machine, we apply a transformation that store-allocates continuations instead of keeping them in a stack. The details of this transformation are available in the online appendix[3]. We continue the abstraction process on this version of the concrete machine.

Figure 5 depicts an abstraction of the concolic machine. The control component of the abstract machine is adapted so that each continuation is applied to an *abstract value* instead of a concrete one. The machine's stores are also abstracted, so that they form mappings from abstract addresses to abstract values or to sets of abstract continuations. The abstraction of a continuation is defined as a straightforward point-wise abstraction of its components. More specifically, the program continuation for implementing let is adapted to include abstract environments and abstract addresses instead of concrete ones. Furthermore, the failure continuation is adapted to include an abstraction of the path constraint. Both continuations are adapted so that they refer to an abstract continuation address as their next continuation. Abstractions for these addresses are omitted from this paper, as any sound abstraction will do (e.g., one that limits the number of calling contexts).

Finally, the model component (M) and the path constraint component (φ) are abstracted to obtain a fully abstracted concolic machine. To do so, we render symbolic formulae, symbolic expressions, and symbolic variables abstract, as discussed in the following sections.

$$\begin{array}{ll}
\widehat{\varsigma} \in \widehat{\Sigma} ::= \langle \hat{c}, \hat{\sigma}, \hat{a_\kappa}, \hat{\kappa}, \hat{a_\psi}, \hat{\psi}, \hat{M}, \widehat{\mathsf{ctx}}, \hat{\varphi}, \hat{V} \rangle & \widehat{c} \in \widehat{Control} ::= \mathsf{ev}(e, \widehat{\rho}) \mid \mathsf{ap}(\widehat{v}) \\
\widehat{a_\kappa} \in \widehat{KAdr} ::= \ldots \mid \mathsf{Hlt} \quad \widehat{a_\psi} \in \widehat{FAdr} ::= \ldots \mid \mathsf{Hlt} & \mathsf{ctx} \text{ is a finite abstraction of the context} \\
\widehat{\varphi} \in \widehat{Formula} \quad \widehat{M} \in \widehat{Model} & \widehat{\kappa} \in \widehat{KontSto} ::= \widehat{KAdr} \to \mathcal{P}(Continuation) \\
\widehat{\psi} \in \widehat{FailSto} ::= \widehat{FAdr} \to \mathcal{P}(FailureContinuation) & \widehat{v} \in \widehat{Value} = \widehat{ProgramValue} \times \widehat{SymbolicValue} \\
\rho \in \widehat{Environment} = Identifier \to \widehat{Value} & \hat{V} \in \widehat{Visited}
\end{array}$$

Fig. 5: Abstraction of the state space as a component-wise abstraction of the concrete state space

5.3 Abstracting Symbolic Representations

As concrete symbolic variables have the same representation as concrete addresses, any suitable address abstraction can also be used for their abstraction.

[3] `https://doi.org/10.5281/zenodo.16568308`

However, since the abstraction of symbolic variables is central to our approach we define their representation explicitly below.

Definition 5.3.1. *Abstract symbolic variables are constructed from a program label and a finite abstraction of a calling context.*

$$\widehat{\mathtt{x}} \in \widehat{\mathit{SymbolicVariable}} = \mathit{Label} \times \widehat{\mathit{Context}}$$

The power set of these abstract symbolic variables trivially forms a join-semilattice with the subset relation as its partial order, and the union as its least upper bound. Having defined this lattice, we define a Galois connection between the concrete and abstract representations of symbolic variables.

Definition 5.3.2. *$\mathcal{P}(\mathit{SymbolicVariable})$ and $\mathcal{P}(\widehat{\mathit{SymbolicVariable}})$ form a Galois connection $\langle \mathcal{P}(\mathit{SymbolicVariable}), \subseteq \rangle \underset{\alpha_x}{\overset{\gamma_x}{\leftrightarrows}} \langle \mathcal{P}(\widehat{\mathit{SymbolicVariable}}), \subseteq \rangle$, where α and γ are defined as follows:*

$$\alpha_x(SV) = \bigcup_{(\ell,\mathsf{ctx}) \in SV} \{(\ell, \alpha(\mathsf{ctx}))\} \qquad \gamma_x(\widehat{SV}) = \bigcup_{(\ell,\widehat{\mathsf{ctx}}) \in \widehat{SV}} \{(\ell, \mathsf{ctx}) \mid \widehat{\mathsf{ctx}} \subseteq \mathsf{ctx}\}$$

It is important to note that a single abstract symbolic variable could correspond to multiple concrete symbolic variables. This is because the abstraction of the context makes a possibly infinite concrete context finite, resulting in a potentially infinite number of corresponding concrete symbolic variables. The same does not hold for the reverse case: abstractions of concrete symbolic variables only result in a single abstract symbolic variable as witnessed by the usage of singleton sets in the definition of the abstraction function.

Next, we define a lattice of symbolic expressions ($\widehat{e} \in \widehat{\mathit{SymbolicValue}}$). This lattice is defined as a flat lattice that consists of $\mathsf{sym}(e)$ as its elements, and fresh as its top element with the usual partial-ordering and least-upper bound.

Having defined the partial ordering relation, and having established that this representation forms a join-semilattice, we define a Galois connection between the power set of concrete symbolic expressions $\mathcal{P}(\mathit{SymbolicValue})$ and the abstract symbolic expressions $\widehat{\mathit{SymbolicValue}}$ defined earlier.

Definition 5.3.3. *The sets $\mathcal{P}(\mathit{SymbolicValue})$ and $\widehat{\mathit{SymbolicValue}}$ form a Galois connection $\langle \mathcal{P}(\mathit{SymbolicValue}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \widehat{\mathit{SymbolicValue}}, \leq \rangle$ where α and γ are defined as follows:*

$$\begin{array}{ll}
\alpha(S) = \bigsqcup_{e \in S} \alpha_e(e) & \gamma(\bot) = \{\} \\
\alpha_e(n) = \mathsf{sym}(n), n \in \mathbb{N} & \gamma(\mathsf{sym}(n)) = \{n\}, n \in \mathbb{N} \\
\alpha_e(b) = \mathsf{sym}(b), b \in \mathbb{B} & \gamma(\mathsf{sym}(b)) = \{b\}, b \in \mathbb{B} \\
\alpha_e(\mathtt{x}) = \mathsf{sym}(\alpha_x(\mathtt{x})) & \gamma(\mathsf{sym}(\mathtt{x})) = \gamma(\mathtt{x}) \\
 & \gamma(\mathsf{fresh}) = \mathit{SymbolicValue} \\
\alpha_e(op(e_1, \ldots e_n)) = op(\alpha(e_1, \ldots, \alpha(e_n)) & \gamma(\mathsf{sym}(op(\widehat{e}_1, \ldots, \widehat{e}_n))) = \{op(e_1, \ldots, e_n) \mid e_i \in \gamma(\widehat{e}_i)\}
\end{array}$$

Intuitively, the partial ordering is guided by the concrete symbolic expressions corresponding to an abstract symbolic expression. For instance, an abstract expression e corresponds to the singleton set of concrete expression e, whereas `fresh` corresponds to all possible symbolic expressions. The choice for representing this abstract value using a fresh symbolic variable is not a coincidence. For example, a symbolic variable `y0` would subsume a symbolic expression `x0` > 0 as the former evaluates to any value while the latter evaluates to a boolean value after computing the inequality with zero.

Other, more precise abstractions can also be defined for symbolic expressions. However, for simplicity of our presentation, we chose to represent abstract symbolic expressions as a flat lattice consisting of a bottom element, a concrete symbolic expression, and a fresh symbolic variable.

5.4 Path Constraints Abstractions

In the previous section, we defined abstractions for symbolic expressions. These abstractions are used for abstracting the store and its concolic values as well as the symbolic expressions in the path constraint. In this section, we discuss how abstract symbolic expressions are integrated with a symbolic path constraint in order to obtain an abstract path constraint.

Definition 5.4.1. *An abstract path constraint is formed by an abstract formula* $\widehat{Formula}$ *which is defined as follows:*

$$\widehat{\varphi_1}, \widehat{\varphi_2} \in \widehat{Formula} ::= \widehat{\varphi_1} \wedge \widehat{\varphi_2} \mid \neg\widehat{e} \mid \widehat{e} \mid \emptyset$$

Thus, abstract formulae share the same structure as their concrete counterpart but use abstract symbolic expressions as their constraints. A natural partial order on both concrete and abstract formulae can be defined using the logical implication ($\Rightarrow$). Consequently, the least-upper bound can be defined as the least generalization such that if $\varphi_1 \sqcup \varphi_2 = \varphi_3$ then $\varphi_1 \Rightarrow \varphi_3$ and $\varphi_2 \Rightarrow \varphi_3$.

Definition 5.4.2. *Concrete formulae from Formula form a Galois connection with abstract formulae* $\widehat{Formula}$ *denoted by* $\langle Formula, \Rightarrow\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \widehat{Formula}, \Rightarrow\rangle$*, where* α *and* γ *are defined as follows:*

$$\begin{array}{ll}
\alpha(\varphi_1 \wedge \varphi_2) = \alpha(\varphi_1) \wedge \alpha(\varphi_2) & \gamma(\widehat{\varphi_1} \wedge \widehat{\varphi_2}) = \displaystyle\bigsqcup_{\substack{\varphi_1 \in \widehat{\varphi_1} \\ \varphi_2 \in \widehat{\varphi_2}}} \varphi_1 \wedge \varphi_2 \\
\alpha(\neg e) = \neg\alpha(e) & \gamma(\neg\widehat{e}) = \bigsqcup\{\neg e \mid e \in \gamma_e(\widehat{e})\} \\
\alpha(e) = \alpha_s(e) & \gamma(\widehat{e}) = \bigsqcup\{e \mid e \in \gamma_e(\widehat{e})\}
\end{array}$$

5.5 Satisfiability Checking and Abstract Counting

Since an abstract formula corresponds to one or more concrete formulae, *satisfiability checking* of these formulae also need to be adapted. To this end, we define a *translation* function that translates abstract formulae into *SMT* formulae that can be used in any SMT solver.

Imprecise Translations The definition of an *SMT* formula is depicted below. The structure of an *SMTFormula* largely corresponds to the structure of concrete and abstract formulae, except that symbolic variables are generated from the set of natural numbers ($\mathbb{N}$) instead of program locations and their context.

$$\varphi'_1, \varphi'_2 \in \mathit{SMTFormula} ::= \varphi'_1 \wedge \varphi'_2 \mid \mathtt{x}' \mid \varphi'_1 \mid \neg\varphi'_1 \qquad \mathtt{x}' \in \mathit{SMTVariable} ::= \mathtt{x}n, n \in \mathbb{N}$$

Below we define the translation function from abstract symbolic formulae to *SMT* formulae. We assume an infinite pool of natural numbers shared between all invocations of translate. A selection from this pool is denoted by $n \in \mathbb{N}$.

$$\begin{array}{ll}
\mathsf{translate} :: \widehat{\mathit{Formula}} \to \mathit{SMTFormula} & \mathsf{translate}(\widehat{\varphi_1} \wedge \widehat{\varphi_2}) = \mathsf{translate}(\widehat{\varphi_1}) \wedge \mathsf{translate}(\widehat{\varphi_2}) \\
\mathsf{translate}(\mathsf{sym}(\neg e)) = \neg\mathsf{translate}(e) & \mathsf{translate}(\mathsf{sym}(n)) = n, n \in \mathbb{N} \\
\mathsf{translate}(\mathsf{sym}(b)) = b, b \in \mathbb{B} & \mathsf{translate}(\mathsf{sym}(\mathtt{x})) = \mathtt{x}n, n \in \mathbb{N} \\
\mathsf{translate}(\bot) = \emptyset & \mathsf{translate}(\mathsf{fresh}) = \mathtt{x}n, n \in \mathbb{N}
\end{array}$$

Note that the translation of abstract symbolic variables to SMT variables requires a unique SMT variable for every occurrence of an abstract symbolic variable, even if these abstract symbolic variables are identical. This is because abstract symbolic variables originate from input statements, and are allocated by combining the location of the input statement in the source program with an abstraction of the current calling context of the program execution. Thus, if such input statements repeat in the same execution, multiple concrete input statements could result in the same abstract symbolic variable, while getting a unique symbolic variable in the concrete execution. Therefore, each symbolic variable needs to be translated to a unique SMT variable.

Precise Translations with Abstract Counting The above translation is suboptimal as every occurrence of an abstract symbolic variable is translated to a distinct *SMT* variable. The code example depicted below illustrates in which situations this suboptimal translation is warranted.

```
y = 0
while True:
   y = x
   x = input()
   if x > y: error
```

In this example, the error is reachable as the user input from the previous loop iteration could be smaller than the current one. To detect is error, a concrete concolic execution engine generates an infinite number of symbolic variables for the input statement on line 4. To render the abstract execution finite, the abstracted version of the concolic execution engine associates a *single* abstract variable with that input statement, yielding the following path constraint at the error statement after two iterations of the `while` loop (where `x0` is an abstract symbolic variable associated with the input statement on line 4).

$$\mathtt{x0} > \mathtt{x0} \wedge \mathtt{x0} > \mathtt{x0}$$

In this case, the abstract symbolic variable `x0` cannot refer to the same concrete symbolic variable as the path constraint would become unsatisfiable. Therefore, the imprecise translation is warranted when abstract symbolic variables refer to multiple concrete symbolic variables.

However, abstract symbolic variables can usually occur more than once in a symbolic path constraint, as illustrated by the code listing depicted below. An abstract concolic execution of this program yields to following path constraint at the error statement: $\mathtt{x0} > 5 \wedge \mathtt{x0} < 4$, which is clearly unsatisfiable. However, since all occurrences of abstract symbolic variables are replaced by fresh onces, the formula becomes satisfiable again, resulting in the imprecise conclusion that the error statement is reachable.

```
x = input()
if x < 4:
   if x > 5: error
```

This problem arises from the absence of information regarding the *cardinality* of a symbolic variable. To render the translation more precise, we propose to conservatively approximate this cardinality by counting the number of concrete symbolic variables corresponding to an abstract symbolic variable. This approach is commonly referred to as *abstract counting* [15], and has been used for soundly implementing *strong updates*. We apply this same concept for determining whether a symbolic variable corresponds to zero, one or more concrete symbolic variables.

Definition 5.5.1. *An* abstract count *is defined as follows:*

$$c \in \mathit{AbstractCount} ::= 0 \mid 1 \mid \infty$$

The elements of *AbstractCount* form a join-semilattice, where the partial order is defined as $0 \leq 1 \leq \infty$ and the least-upper bound as $0 \sqcup x = x$ (idem for the symmetric case), $1 \sqcup 1 = 1$ and $x \sqcup \infty = \infty$ (idem for the symmetric case). Having defined a lattice structure, we can define an abstract operation called inc which increases the abstract count by one.

$$\mathsf{inc}(0) = 1 \quad \mathsf{inc}(1) = \infty \quad \mathsf{inc}(\infty) = \infty$$

To determine the abstract count for each symbolic variable, we introduce an abstract count *mapping* C which maps symbolic variables to their abstract count. The mapping is initially set to zero for every abstract symbolic variable. The definition of this mapping is depicted below:

$$C \in \mathit{AbstractCountMap} = \widehat{\mathit{SymbolicVariable}} \rightarrow \mathit{AbstractCount}$$

Mapping C is used for rendering the translation function more precise. In the definition below, we assume that a mapping $\mathcal{N} : \widehat{\mathit{SymbolicVariable}} \rightarrow \mathit{SMTVariable}$

exists and is updated accordingly when SMT variables are generated from abstract symbolic variables. We denote a successful lookup from this mapping using $\mathcal{N}(\mathtt{x})$. The definition below only depicts the translation of symbolic variables, as the translation of other types of formulae remains identical, except for passing the abstract count mapping alongside the formula that is being translated.

$$\mathsf{translate}(\mathtt{x}, C) = \begin{cases} \mathcal{N}(\mathtt{x}) & \text{if } \; C(\mathtt{x}) = 1 \\ \mathtt{x}n, n \in \mathbb{N} & \text{otherwise} \end{cases}$$

The translation function handles the satisfiability checking in one direction: the direction from the analysis to the SMT solver. A concolic execution engine, however, requires bidirectional communication by constructing a model satisfying the path constraint and feeding it back to the next concolic execution.

To this end, we define an inverse function getModel (depicted below), to compute an abstract model from the SMT model. We also define a mapping from *SMT* variables to their original abstract symbolic variables so that an abstract model can be computed from the *SMT* model.

$$\mathit{SMTModel} = \mathit{SMTVariable} \to \mathit{ProgramValue} \quad \mathcal{N}^{-1} :: \mathit{SMTVariable} \to \widehat{\mathit{SymbolicVariable}}$$

$$\mathsf{getModel} :: \mathit{SMTModel} \times \mathcal{N}^{-1} \to \widehat{\mathit{Model}} \qquad \mathsf{getModel}(M, N) = \bigsqcup_{\mathtt{x} \in M} [N(\mathtt{x}) \mapsto M(\mathtt{x})]$$

As every abstract symbolic variable can have multiple corresponding *SMT* variables, all corresponding SMT variables are joined into a single abstract symbolic variable, resulting in an over-approximation of the concrete model.

5.6 Abstracting the Visited Set

The visited set is an important component in the concrete semantics as it ensures that the concrete concolic execution engine does not perpetually alternate between same branches of the program. This visited set needs to be abstracted too, as it would otherwise lead the abstract interpreter to conclude that the concolic execution never finishes, resulting in a $\perp$ value.

Definition 5.6.1. *An abstracted visited is formed by a* has-*visited and a* may-*visited set. The former corresponds to the branches that have been visited, while the latter corresponds to the branches that may have been taken.*

$$\hat{V} \in \widehat{\mathit{Visited}} ::= \mathcal{P}(\mathit{Formula}) \times \mathcal{P}(\mathit{Formula})$$

Definition 5.6.2. *The least upper bound $\sqcup$ and partial ordering $\leq$ for $\widehat{\mathit{Visited}}$ is defined as follows.*

$$(h_1, m_1) \leq (h_2, m_2) \triangleq h_1 \supseteq h_2 \wedge m_1 \subseteq m_2 \quad (h_1, m_1) \sqcup (h_2, m_2) \triangleq (h_1 \cap h_2, m_1 \mathit{\Delta} m_2)$$

where $m_1 \mathit{\Delta} m_2$ is the symmetric union of m_1 and m_2.

When computing the least upper bound of two states of the concolic machine, paths visited in both states end up in the *has*-set while the others are kept in the *may*-set. This leads us to the following definition of the abstract in operation.

Definition 5.6.3. *The abstract* in *operation is defined as follows.*

$$\mathsf{in}(e,(h,m),a_{\psi}1,a_{\psi}2) = \begin{cases} \{a_{\psi}1, a_{\psi}2\} & \textit{if } e \in m \\ \{a_{\psi}1\} & \textit{if } e \notin h \wedge e \notin m \\ \{a_{\psi}2\} & \textit{if } e \in h \wedge e \notin m \end{cases}$$

We also define an abstract operation to add elements to the visited set.

Definition 5.6.4. *The operation* add *is defined as follows:*

$$\mathsf{add}(e,(h,m)) = (\{e\} \cup h, m)$$

5.7 Abstract Stepping Relation

Having defined an abstraction of the standard *CESK* machine components, an abstraction of symbolic expressions and their formulae and a translation to SMT formulae, we can put everything together and define an abstract version of the small-stepping relation $\widehat{\rightsquigarrow}$ on the concolic machine (depicted in fig. 6).

Atomic evaluation remains largely unchanged except that literals are now injected in the abstract domain (through their abstraction functions α) for program and symbolic values. All rules are adapted to take the abstract count mapping C into account. This mapping propagates mostly unchanged through the stepping relation except for the [Input] and [ST-Backtrack] rules. The [Input] rule "allocates" a fresh abstract symbolic variable based on the source location of the matching input statement and the current abstract program context. The abstract count mapping is then updated to take this allocation into account by applying the inc meta-function to the current abstract count for that abstract symbolic variable. [ST-Backtrack'] *resets* the abstract count mapping as the program execution is restarted.

The remaining rules also need to take the abstractions of the continuation stores into account. Rules [ST-IfTrue'] and [ST-IfFalse'] perform *weak updates* on both the continuation and the value store. This weak update is performed by joining the new value at a specified address with its old value. While our semantics does not preclude *strong updates*, incorporating them would require additional changes to the abstract count mapping. Finally, rule [ST-Backtrack'] is updated to first translate the abstract path condition to one suitable for the model function and then translating its results back into an abstract model using the getModel function. The new model M' is joined with the old model M so that the number of possible abstract models remain finite. Next, we formulate and prove soundness of our abstract concolic semantics and prove that the resulting analysis terminates on any program input.

Theorem 5.7.1. $\widehat{\rightsquigarrow}$ *is a sound over-approximation of* $\rightsquigarrow$. *That is, for every* $\varsigma \rightsquigarrow \varsigma'$ *there exists an approximation* $\widehat{\varsigma} \mathrel{\widehat{\rightsquigarrow}} \widehat{\varsigma'}$ *given that* $\alpha(\varsigma) \sqsubseteq \widehat{\varsigma}$ *such that* $\alpha(\varsigma') \sqsubseteq \widehat{\varsigma'}$.

$$\frac{(v,\overline{v'}) = \mathsf{lookupModel}(M,\mathtt{x},\overline{v}) \qquad \mathtt{x} = \mathsf{fresh}(\ell(\mathsf{input}),\mathsf{ctx})}{\begin{array}{c}\langle \mathsf{ev}(\mathsf{input},\rho),\sigma,a_\kappa,\kappa,a_\psi,\psi,\overline{v},\mathsf{ctx},M,C,\varphi,\widehat{V}\rangle \\ \widehat{\leadsto}\langle \mathsf{ap}((v,\mathtt{x})),\sigma,a_\kappa,\kappa,a_\psi,\psi,\overline{v'},\mathsf{ctx},M,C[\mathtt{x}\mapsto \mathsf{inc}(C(\mathtt{x}))],\varphi,\widehat{V}\rangle\end{array}}\ \textsc{Input}$$

$$\frac{\begin{array}{c}[\![ae]\!](\rho,\sigma) \ni (true,s) \qquad \varphi_t = \varphi \wedge s \qquad \varphi_f = \varphi \wedge \neg s \\ a'_\psi \in \mathsf{in}(\varphi_f,\widehat{V},\mathsf{alloc}(\ell(ae)),a_\psi) \qquad \psi' = \psi'[a'_\psi \mapsto \psi(a'_\psi) \sqcup \{\mathsf{branch}(\varphi_f) :: a_\psi\}]\end{array}}{\begin{array}{c}\langle \mathsf{ev}(\mathsf{if}\ ae\ e_1\ e_2,\rho),a_\kappa,\sigma,\kappa,a_\psi,\psi,\overline{v},\mathsf{ctx},M,C,\varphi,\widehat{V}\rangle \\ \widehat{\leadsto}\langle \mathsf{ev}(e_1,\rho),\sigma,a_\kappa,\kappa,a'_\psi,\psi',\overline{v},\mathsf{ctx},M,C,\varphi_t,\mathsf{add}(\varphi_f,\widehat{V})\rangle\end{array}}\ \text{ST-IfTrue'}$$

$$\frac{\begin{array}{c}[\![ae]\!](\rho,\sigma) \ni (false,s) \qquad \varphi_t = \varphi \wedge s \qquad \varphi_f = \varphi \wedge \neg s \\ a'_\psi = \mathsf{in}(\varphi_t,\widehat{V},\mathsf{alloc}(\ell(ae)),a_\psi) \qquad \psi' = \psi'[a'_\psi \mapsto \psi(a_\psi) \sqcup \{\mathsf{branch}(\varphi_t) :: a_\psi\}]\end{array}}{\begin{array}{c}\langle \mathsf{ev}(\mathsf{if}\ ae\ e_1\ e_2,\rho),\sigma,a_\kappa,\kappa,a_\psi,\psi,\overline{v},\mathsf{ctx},M,C,\varphi,\widehat{V}\rangle \\ \widehat{\leadsto}\langle \mathsf{ev}(e_2,\rho),\sigma,a_\kappa,\kappa,a'_\psi,\psi',\overline{v},\mathsf{ctx},M,C,\varphi_f,\mathsf{add}(\varphi_t,\widehat{V})\rangle\end{array}}\ \text{ST-IfFalse'}$$

$$\begin{array}{c}\text{ST-Backtrack'} \\ \dfrac{M' = \mathsf{getModel}(\mathsf{model}(\mathsf{translate}(\varphi))) \qquad \mathsf{branch}(\varphi,\overline{x'}) :: a'_\psi \in \psi(a_\psi)}{\langle \mathsf{ap}(v),\sigma,\mathsf{Hlt},\kappa,a_\psi,\psi,\overline{v},\mathsf{ctx},M,C,\varphi',\widehat{V}\rangle \widehat{\leadsto}\langle \mathsf{ev}(e_0,\rho_0),\sigma_0,\mathsf{Hlt},\emptyset,a'_\psi,\psi,\overline{v'},\emptyset,M \sqcup M',\emptyset,\emptyset,\widehat{V}\rangle}\end{array}$$

Fig. 6: Abstract stepping relation $\widehat{\leadsto}$, highlighted in grey are the parts changed in comparison to the concrete stepping relation. Rules for atomic evaluation, [St-Let1] and [St-Let2] are omitted for brevity but are available in the online appendix.

Proof. Assuming that there exists a concrete transition, $\varsigma \leadsto \varsigma'$, we show by case analysis on the applied rules that there is an abstract transition $\hat{\varsigma}\ \widehat{\leadsto}\ \hat{\varsigma}'$ so that the proposition $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$ holds for any given $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$. A full version of the proof is available in the online appendix.[4] □

Theorem 5.7.2. *Termination.* $\widehat{Eval}$ *always terminates.*

Proof. Proof by analysis of the cardinality of each state-space component. A full version of the proof is available in the online appendix. □

6 Variations on the Analysis

In contrast to the state of the art in soft contract verification, the systematic abstraction of a concolic execution machine enables a number of variations on the resulting abstract machine. These variations either render the analysis results more precise, or improve the analysis' performance. To demonstrate the possible variations, we take a similar approach as in Glaze et al. [11] and vary the machine by global widening and per-state widening.

[4] `https://doi.org/10.5281/zenodo.16568308`

6.1 Global Widening

One variation of the machine is to widen its components so that they become *shared* between all program states. More specifically, all machine components can be widened except for the control component, the addresses for the top of the failure and regular continuation stack, and the program and iteration context. One could also widen *path constraint* so that it becomes shared between all program states, which leads to a whole-program invariant. However, since the path constraints across a whole program can differ substantially, the most generic path constraint usually is the empty one. The *model*, on the other hand, can be widened to a global version since its symbolic variables are indexed by the path constraint for which the model was computed. Widening the model to be shared with all program states therefore does not impact its values since its values would be joined anyway. Finally, widening the abstract counts leads to imprecise information for the count of each symbolic variable. Every abstract count will be widened to ∞ causing each symbolic variable in the path condition to be translated into a unique SMT variable.

The widened state space Σ', consisting of a set of small step states, and the shared components, is depicted below. The abstracted evaluation function $\widehat{Eval}$ is adapted to operate on these widened states by iterating over each of its small-step states, lifting them to a non-widened small-step state, applying the evaluation relation and joining its results together. Note that this transformation does not impact the formal semantics.

$$\varsigma \in \Sigma ::= \langle c, a_\kappa, a_\psi, \mathsf{ctx}, \varphi \rangle$$
$$\Sigma' = \mathcal{P}(\Sigma) \times (\mathit{Store} \times \mathit{ContSto} \times \mathit{FailSto} \times \mathit{Model})$$
$$\mathsf{lift}(\langle c, a_\kappa, a_\psi, \mathsf{ctx}, \varphi \rangle, (\sigma, \kappa, \psi, M)) = \langle c, \sigma, a_\kappa, \kappa, a_\psi, \psi, M, \mathsf{ctx}, \varphi \rangle$$
$$\widehat{Eval}(\Sigma, S) = \Sigma \sqcup \bigsqcup_{\substack{\varsigma \in \Sigma \\ \mathsf{lift}(\varsigma, S) \widehat{\leadsto} \varsigma' \\ \mathsf{lift}(\varsigma'', S') = \varsigma'}} (\{\varsigma''\}, S')$$

Conclusion This transformation shows that our abstract machine can be easily adapted to globally widen its components. Although this global widening drastically decreases the time complexity of resulting analysis, we argue against widening the symbolic execution components of the machine, as they will decrease the precision of its results.

6.2 Per-state Widening

A more practical form of widening is *per-state widening*. This form of widening entails that some analysis components are passed onto the next analysis states instead of making them part of the next analysis state itself. The widened state space Σ', depicted below, consists of pairs of which the second element is a mapping between abstract states $\varsigma \in \Sigma$ and their widened components. The abstract evaluation function $\widehat{Eval}$ is adapted accordingly.

$$\varsigma \in \Sigma = \langle c, a_\kappa, a_\psi, \mathsf{ctx}, PC\rangle \qquad \Sigma' = \mathcal{P}(\Sigma) \times S$$
$$S = \Sigma \rightarrow (Store \times ContSto \ \times FailSto \ \times Model)$$
$$\mathsf{lift}_1(\langle c, a_\kappa, a_\psi, \mathsf{ctx}, \varphi\rangle, (\sigma, \kappa, \psi, M)) = \langle c, \sigma, a_\kappa, \kappa, a_\psi, \psi, M, \mathsf{ctx}, \varphi\rangle$$
$$\mathsf{lift}_2(\varsigma, S) = \mathsf{lift}_1(\varsigma, S(\varsigma))$$
$$\widehat{Eval}(\Sigma, S) = \Sigma' \cup \bigsqcup_{\substack{\varsigma \in \Sigma \\ \mathsf{lift}_2(\varsigma, S) \widehat{\rightsquigarrow} \varsigma' \\ \mathsf{lift}_1(\varsigma'', S') = \varsigma'}} (\{\varsigma''\}, S[\varsigma'' \mapsto S'])$$

This reduces the number of states drastically, as each modification to the shared components no longer leads to a different program state.

Having defined the widened transfer function, we can assess its impact on the precision of the analysis result. Widening per state usually results in widening of components in the presence of *loops*. This means that components such as the path constraint get widened to loop iterations resulting in a *loop invariant*. The abstract count mapping gets similarly widened to each loop iteration, resulting in a ∞ count for symbolic variables that get produced in a loop. Again, the results of the model are not affected by these changes as they are already uniquely identified by their symbolic variable combined with the path constraint for which the model was computed.

Conclusion: Again, our abstract machine can be easily adapted to support per-state widening. This widening approach is preferable over globally widening the symbolic components, as per-state widening naturally leads to widening at loop-heads, which is preferable is most cases.

7 Evaluation

We evaluate our approach by instantiating it for the problem of *soft contract verification* [16,17,24]. In this evaluation, we instantiate our abstract concolic execution engine for soft contract verification. To this end, contract specifications are compiled into low-level *assertions* that can be checked by the machinery of our approach. We compare our approach to traditional soft contract verification on a dataset provided by the state of the art. This dataset consists of a number of Racket programs annotated with contract specifications. All programs in this dataset are considered to be *safe* (e.g., do not contain any contract violations) by the original authors [16], and we also manually verified that this was indeed the case. We use this ground truth to compare our approach to traditional soft contract verification by measuring the number of unverified contracts.

Listing 2 The rules central to the translation from CFCP to ordinary λ-calculus.

$$\mathsf{mon}^{j,k}\ (\mathsf{flat}\ v_1)\ v_2 \to \mathsf{if}\ (v_1\ v_2)\ v_2\ (\mathsf{blame}\ j)$$
$$\mathsf{mon}^{j,k}\ (\kappa_1 \to \kappa_2)\ (\lambda x.e) \to (\lambda x.\mathsf{mon}^{j,k}\ \kappa_2\ (e[x \mapsto \mathsf{mon}^{k,j}\ \kappa_1\ x]))$$

7.1 Instantiating Soft Contract Verification

Findler et al. [14] propose a contract system for specifying the expected behaviour of higher-order functions. In the listing shown below, we again depict a contract for the `square` function, stating that its input should be a number while guaranteeing that its output will be positive:

```
def square(x: number?): positive? = return x*x
```

In the language proposed by Findler et al., called CPCF, this code example would be translated into the following term:

$$\mathsf{letrec}\ \mathit{square} = \mathsf{mon}^{j,k}\ (\mathit{number}? \to \mathit{positive}?)\ (\lambda x.(x * x))$$

The contract-annotated function is transformed to a λ-expression containing the original parameters of the function and its body. Furthermore, the contract is translated to a *contract monitor*, mon, which attaches the contract to the function so that when the function is applied the *number*? contract is checked on the input value and the *positive*? contract is checked on the output value.

The translation of these contract systems into the language used in our approach is straightforward and partially given by Dimoulas et al. [5,6]. Listing 2 depicts the rules for translating CFCP terms to ordinary λ-calculus terms. These rules can then be applied recursively to obtain a contract-free program that can be analysed using our approach. The full translation is a bit more involved, as it also translates Racket-specific constructs such as `contract-out` and contracts on structs, but their translations are omitted here for brevity[5]

7.2 Implementation

We implemented our approach in Monarch [25], a framework for static analysis through abstract interpretation written in Haskell. This implementation is **publicly available as a replication package at** `https://doi.org/10.5281/zenodo.16410896`. The implementation follows the formalization for the most part, but extends the analysed language with support for strings, heap-allocated pairs, and vectors. Moreover, the analysed language also supports functions accepting multiple arguments. It uses a constant propagation domain as the abstract domain for program values, and power-set lattices for abstracting pointers and closures. Z3 is used an SMT solver.

[5] The full translation is included in our replication package.

7.3 Experimental Setup

To measure the performance and accuracy of our approach, we run our analysis on a collection of benchmark programs from existing soft contract verification work [24], and we compare the number of false positives detected by our approach to the number of false positives detected by related work. The ground truth is constructed by looking at benchmark programs that contain no contract violations, making every detected contract violation a false positive.

Table 1: An overview of the set of benchmark programs. Depicted are the number of lines of code (according to `sloc`) in the original and translated program.

Name	Original	Processed	Name	Original	Processed
games-snake	134	4525	mochi-zip	13	2684
games-tetris	250	6690	sergey-blur	12	2570
games-zombie	230	5110	sergey-eta	8	2543
mochi-fold-div	11	2657	sergey-kcfa2	10	2563
mochi-hors	12	2613	sergey-kcfa3	15	2573
mochi-hrec	8	2648	sergey-loop2	18	2593
mochi-l-zipunzip	15	2712	sergey-mj09	12	2560
mochi-map-foldr	9	2707	sergey-sat	28	2675
mochi-mappend	10	2689	softy-append	6	2595
mochi-mem	11	2695	softy-cpstak	21	2637
mochi-mult	7	2645	softy-last-pair	5	2573
mochi-neg	10	2610	softy-last	13	2626
mochi-nth0	11	2608	softy-length-acc	8	2589
mochi-r-file	25	2719	softy-length	6	2583
mochi-r-lock	8	2609	softy-member	6	2579
mochi-reverse	9	2620	softy-recursive-div2	8	2598
mochi-sum	7	2581	softy-subst	9	2617
			softy-tak	9	2616

We first translate the benchmark programs to *ANF*, compile each contract down to λ_s in the manner described above, and insert function calls to each contracted function. A summary of the resulting set of programs is depicted in Table 1. The translated programs are larger than the original programs because they are translated into an *ANF* form, include a prelude of primitive functions annotated with contracts and some built-in contracts, and have the contract specifications compiled down into base-level assertions. To measure the precision of the analysis, the analysis is executed and the number of unique contract violations is counted.

The evaluation features two configurations of the analysis: a version, called "local" with no widening mirroring our formalization closely, and a version with per-state widening called "flow". Both configurations are instantiated with *k-CFA* context sensitivity for function calls, and with path constraints as its branching sensitivity. In the evaluation, we compare multiple values for k bounded to 5 call-sites, and measure whether it has an impact on the analysis precision.

For comparing the precision and performance of our approach with the state of the art, we replicate the results from Vandenbogaerde et al. [24] by running their replication package on the aforementioned benchmarks and measuring their running time and precision. For measuring the performance, we repeated the analysis of each benchmark program *20 times* with a timeout of 15 minutes and report its summary statistics. The benchmarks are executed on a *AMD EPYC 9384X* machine having 12 GiB of available memory. No limitations on garbage collection have been imposed.

7.4 Results

Precision Results Table 2 depicts the precision results of our benchmark programs. Benchmark programs for which all configurations result in either a timeout, or an error (in case of missing features in the analysed programming language), are omitted from the table. The omitted benchmarks comprise 11 out of the 35 benchmarks from the benchmark set. The state of the art did not time out on the omitted benchmarks. The table is split into three parts. The first part depicts the number of false positives found in the state-of-the-art approach [24], the second and third part depict the "flow" and "local" configurations (cf. above). These configurations are further subdivided into different values for k, which a tunable parameter changing how many function calls should be retained in the calling context. Higher values of k typically result in a higher precision at the cost of performance. However, as we noticed in the benchmarks such as *sergey_eta*, a higher precision could also lead to improved performance as abstract values grow smaller due to the increased precision of the analysis.

Table 2: Precision results for each benchmark program. Cells containing a ∞ indicate a timeout for that combination of benchmark and configuration.

	[24]	Flow						Local					
k		0	1	2	3	4	5	0	1	2	3	4	5
Name													
mochi_fold-div	5	4	1	1	1	1	1	∞	∞	∞	∞	∞	∞
mochi_hors	2	3	3	3	3	3	3	∞	∞	∞	∞	∞	∞
mochi_hrec	5	6	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_l-zipunzip	7	2	2	∞	∞	∞	2	∞	∞	∞	∞	∞	∞
mochi_map-foldr	1	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_mappend	4	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_mem	20	3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_mult	5	4	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_neg	4	4	4	4	4	4	4	∞	∞	∞	∞	∞	∞
mochi_nth0	3	4	3	3	3	3	3	∞	∞	∞	∞	∞	∞
mochi_r-file	1	0	0	0	0	0	∞	∞	∞	∞	∞	∞	∞
mochi_r-lock	2	4	4	4	4	4	4	∞	∞	∞	∞	∞	∞
mochi_sum	4	0	0	0	0	0	0	0	0	0	0	0	0
sergey_eta	0	0	0	0	0	0	0	0	0	0	0	0	0
sergey_kcfa2	0	0	0	0	0	0	0	0	0	0	0	0	0
sergey_kcfa3	0	0	0	0	0	0	0	0	0	0	0	0	0
sergey_mj09	0	0	0	0	0	0	0	0	∞	0	0	0	0
sergey_sat	0	0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_append	3	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_cpstak	3	3	3	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_last	0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_length	6	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_length-acc	6	2	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_tak	0	3	3	3	∞	∞	∞	∞	∞	∞	∞	∞	∞

The results show that the "local" configuration of the analysis is barely useable with only 5 out of the 24 benchmarks terminating before the timeout is reached. This outcome is expected, as the "local" configuration results in an exponential number of states, as has been observed in prior work on AAM [11].

The results for the "flow" configuration are more interesting, as they are more likely to terminate before the timeout is reached. We found that in 10 out of the 24 benchmark programs our approach reports fewer false positives than the state-of-the-art approach. In 9 out of the 24 benchmark programs, the number of reported false positives is the same as the state-of-the-art approach, while for 5 benchmark programs, our approach reports more false positives.

Conclusion: The results show that our approach is able to achieve a higher or equal level of precision in the majority of tested benchmark programs, at the cost of being slower than existing approaches and failing to analyse 10 out of the 34 benchmark programs within a time budget of 15 minutes.

Table 3: Performance results for each benchmark program. Each cell contains the mean running time or ∞ indicating a timeout for that combination of benchmark program and configuration. For each combination, the coefficient of variation never exceeds 21.9%, however in 75% of the combinations never exceed 1.3%.

	State-of-the-art	Flow						Local					
		0	1	2	3	4	5	0	1	2	3	4	5
Name													
mochi_fold-div	0.059s	9.10s	2.32s	2.34s	2.35s	2.37s	2.38s	∞	∞	∞	∞	∞	∞
mochi_hors	0.063s	39.65s	99.83s	289.14s	168.42s	492.21s	791.92s	∞	∞	∞	∞	∞	∞
mochi_hrec	0.055s	834.24s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_l-zipunzip	0.074s	146.97s	1566.51s	∞	∞	∞	525.16s	∞	∞	∞	∞	∞	∞
mochi_map-foldr	0.061s	1104.61s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_mappend	0.055s	773.74s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_mem	1.239s	498.28s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_mult	0.058s	347.69s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
mochi_neg	0.043s	132.16s	405.17s	774.56s	1135.49s	1251.37s	1503.50s	∞	∞	∞	∞	∞	∞
mochi_nth0	0.033s	56.58s	78.22s	81.52s	83.02s	83.73s	86.27s	∞	∞	∞	∞	∞	∞
mochi_r-file	0.062s	14.53s	62.26s	397.81s	672.42s	1588.63s	∞	∞	∞	∞	∞	∞	∞
mochi_r-lock	0.076s	30.29s	64.99s	90.67s	103.44s	119.79s	154.05s	∞	∞	∞	∞	∞	∞
mochi_sum	0.047s	0.19s	0.18s	0.17s	0.18s	0.18s	0.18s	0.15s	0.16s	0.16s	0.16s	0.16s	0.16s
sergey_eta	0.002s	0.44s	0.27s	0.19s	0.19s	0.19s	0.19s	1.52s	0.33s	0.19s	0.19s	0.19s	0.19s
sergey_kcfa2	0.045s	1.13s	1.69s	1.98s	1.57s	1.45s	1.25s	8.52s	111.47s	36.05s	7.71s	7.83s	4.18s
sergey_kcfa3	0.035s	1.43s	2.64s	2.02s	1.98s	2.34s	2.52s	12.33s	624.24s	107.40s	251.81s	105.78s	45.88s
sergey_mj09	0.020s	1.69s	3.43s	0.20s	0.20s	0.20s	0.20s	121.53s	∞	0.21s	0.21s	0.22s	0.22s
sergey_sat	0.039s	170.62s	979.98s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_append	0.039s	887.45s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_cpstak	0.084s	151.59s	1291.34s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_last	0.040s	1241.82s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_length	0.059s	708.55s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_length-acc	0.082s	764.85s	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
softy_tak	0.059s	92.14s	1225.51s	1673.31s	∞	∞	∞	∞	∞	∞	∞	∞	∞

Performance Results Table 3 depicts the running times for each combination of benchmark programs and configurations. The results show that our approaches is able to analyse each program in a few minutes. More interestingly, different values of k seem to increase and sometimes decrease the running times of the analysis. In the case of `sergey_kcfa` the resulting difference is quite pronounced since the benchmark program is designed to decrease running times with higher values of k. Others, such as `mochi_neg` only increase their running times with higher values of k. We conclude that impact of k is highly dependent on the benchmark program. Again, we observe that the "local" configuration is outperformed by the "flow" configuration. The results show that the "flow" configuration outperforms the "local" configuration by several orders of magnitude.

Comparing our approach with the state-of-the-art approach for soft contract verification, we observe that our approach is several orders of magnitude slower, but still takes only a couple of minutes to complete. This is because the state-of-the-art approach uses a bespoke *compositional analysis* design for their analysis. Since their approach is compositional, functions can be analysed in isolation regardless of on which paths calls to those functions occur. This substantially reduces the size of the state space, but reduces the precision of the analysis since function calls are no longer analysed in a path sensitive manner. Moreover, while doing so the state-of-art approach chooses an ad-hoc configuration for its abstract machine, and does not consider abstractions for its path constraint, symbolic variables and expressions. Instead, when detecting loops, the state-of-art approach simply discards the path constraint instead of widening it to the least general constraint. A similar phenomenon occurs for its symbolic expressions which are simply discarded instead of widened.

However, one of the strengths of our approach is that it is amenable to a myriad of optimizations proposed in the *AAM* literature, ranging from store deltas [11], to modularization [18], parallelization [22], and abstract garbage collection [7,10]. In Section 6 we already demonstrated *global store widening*, a common optimization in *AAM* literature. This shows that these common optimizations can be easily adapted to our context.

Timed-out benchmarks. Compared to the terminated benchmarks, the timed-out ones are typically larger and contain more complex contracts. For instance, the *games* benchmarks contain contracts on more complex nested data structures for representing elements of the game. Since our analysis is *path-sensitive*, these constraints often result in an exponentially larger state space. Again, other performance-enhancing techniques from the AAM literature might be used to reduce this state space further.

Conclusion: The results demonstrate that most benchmark programs can be analysed within a few minutes. They also highlight the importance of incorporating widening techniques, as the "local" approach alone proves to be insufficient. Compared to the state-of-the-art, our approach achieves higher precision, though at the expense of performance. Our approach is the first to systematically abstract concolic execution, and explore its different configurations. Thus, it remains amenable to a myriad of *AAM* optimizations that have been successful in the past for rendering other *AAM*-based analysis techniques scalable.

8 Related Work

State Merging. Sen et al. [21] propose a state merging technique for reducing the path explosion problem in dynamic symbolic execution. They do so by merging multiple *DSE* states together into a *value summary* therefore reducing the number of states to be explored. Moreover, these summaries allow redundant execution paths to be pruned reducing the state space even more. However, these summaries still contain concrete path constraints and concrete values, thereby only solving the problem of redundant state exploration, but not of termination. Other state merging techniques, also in the context concolic execution, have been proposed [26] with the same termination problem.

Subsumption Checking. Anand et al. [1,2] propose symbolic execution with subsumption checking. Their approach differs from our approach in two ways. First, their approach targets static symbolic execution, while our approach targets a variant of dynamic symbolic execution called *concolic execution*. Using dynamic symbolic execution instead of static symbolic execution allows our approach to replace symbolic values with a symbolic representation of the computed (abstract) program value, instead of reasoning over program values in a purely symbolic way. Therefore, our approach is more precise since it computes an *abstract model* that contains concrete values (depending on the abstract domain used) whenever possible. The second difference is that their analysis results in an under-approximation of the program behaviour, while our analysis is designed to return an over-approximation of the program behaviour at the cost of running into false positives. However, their proposed shape abstractions could be used to improve the abstract domain of *symbolic expressions* in our approach in order to make them more precise.

Combining Symbolic Execution and Abstract Interpretation. The idea of combining symbolic execution with abstract interpretation is not new. For instance, Permenev et al. [19] propose combining static symbolic execution with predicate abstractions [12] through *delayed predicate abstractions*. In their approach, static symbolic execution is used to execute the majority of the program, but then switches over to predicate abstractions to verify the property of interest. The reasoning is that their properties of interest require deep exploration of the static symbolic execution tree, and that using delayed predicate abstraction decreases the required exploration to prove or disprove a property. We follow a similar approach, but use an abstracted version of the state space *right away*. However, depending on the configuration of the abstract machine, we could achieve similar results. For instance, the machine could be configured to reason in a precise way about the possible program states and up to point that the delayed predicate abstraction should occur. To this end, the context component can be modified to include precise calling contexts, and to remove those whenever delayed abstraction is needed.

Sound symbolic execution. Tiraboschi et al. [23] propose *(relational) sound symbolic execution*. Their approach achieves termination by applying a form of *bounded symbolic execution* after which the symbolic execution context is over-approximated to also render the analysis sound. These bounds are implemented through *counters* that are part of the symbolic execution state. Our approach in contrast depends on *arbitrary contexts* to differentiate different abstract concolic states from each-other, which are only joined (i.e., over-approximated) when they are no longer distinguishable. Moreover, our approach is derived from a systematic abstraction of the $CESK_\varphi$ machine using the AAM methodology, which opens it up to a myriad of common optimizations (cf. Section 6), and precision-enhancing techniques (e.g., variants of context sensitivity).

Soft Contract Verification. Nguyen et al. [17] propose *soft contract verification* which is a technique to static analyse software contracts in higher-order programming languages. This work has since been extended to include stateful programs [16], and has been rendered compositional [24]. Our work offers a more generalized way of expressing this soft contract verification. Nguyen et al. propose a machine where the path constraint and symbolic store (i.e., a mapping from addresses to concolic values) is always included in the program state (i.e., corresponding to our "local" configuration), and is never widened to a more abstract value. Instead, their approach solves this problem in ad-hoc manner by simply removing all symbolic expressions and path constraints whenever the program execution reaches a loop. Our approach, in contrast, does not require this loop detection and widens the affected machines components naturally. Finally, the work by Nguyen et al. use a form of static symbolic execution, whereas we use *concolic execution.* The main difference is that their work does not require multiple concolic executions of the same machine, and does not require the computation and therefore abstraction of a model. The lack of a model potentially introduces more imprecision in the analysis results and leads to a higher number of false positives.

Relational Analysis. In a relational analysis, program states are explored in a path-sensitive manner by indexing the program state with predicates on a set of program variables. To render the analysis finite, this set of program variables needs to be finite, and so do their predicates. In traditional relational analyses this set of variables is determined before running the analysis. Other techniques have been proposed to iteratively refine [3] this set of variables until the desired level of precision is reached, or until the precision no longer improves by adding additional variables. Our approach instead derives symbolic variables from the program location of `input` statements, and counts them to determine the number of corresponding concrete symbolic variables.

9 Future Work

In this paper we presented the first systematic abstraction of concolic exection for soft contract verification. In contrast to the state of the art, this systematic abstraction results in a more configurable abstract machine, leading to multiple avenues for future work. As demonstrated by the results, our approach is somewhat slower than the existing state-of-the-art approach for soft contract verification, Therefore, a natural avenue for future work is to further apply several

AAM optimizations [11,18] to render our analysis scalable to larger programs. Finally, another avenue of future work is refining the abstract representation of symbolic expressions. Our representation is currently limited to three levels, a bottom value, a symbolic expression or a fresh value. Other representations could be used to make the analysis more precise, such as those found in [1].

10 Conclusion

We have introduced *abstract concolic execution* as a systematically-derived static analysis supporting soft contract verification. The static analysis is sound and is guaranteed to terminate. The downside is that false positives may be returned, so the analysis is incomplete. We demonstrated, however, that the benefits of concolic execution are retained. For instance, our approach enables analysis developers to replace symbolic representations with abstract values whenever the solver is not sufficiently powerful to solve some of the introduced constraints, or whenever modelling certain programming language features (e.g., closures) becomes a burden on the performance of the solver. To render concolic execution abstract, we introduced a novel extension to the *CESK* machine, $CESK_\varphi$, which models both the instrumented version of the program and the concolic iteration loop itself. Inspired by the *AAM* recipe, we then abstracted this semantics by abstracting the components of the machine, and its small-stepping relation. We have shown that the resulting machine predicts all behaviour found by the concrete machine (i.e., is sound), and that it terminates for any program input. When applying abstract concolic execution to soft contract verification, we found that it outperforms the state-of-the-art approaches in terms of precision while being somewhat slower. We have also demonstrated that our approach is more *configurable* than existing approaches by relying on real abstractions of path constraints, symbolic expressions, and symbolic variables, elliminating the need for a store caches and ad-hoc rules for updating them.

Acknowledgements

This work is partially funded by Research Foundation Flanders (FWO) (grant number 1187122N).

References

1. Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *Proceedings of Model Checking Software, 13th International SPIN Workshop, March 30 - April 1, 2006*, volume 3925 of *Lecture Notes in Computer Science*, pages 163–181. Springer.
2. Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1):53–67, 2009. doi:10.1007/S10009-008-0090-1.

3. Thomas Ball and Sriram K Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, 2002.
4. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.
5. Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on P rinciples of Programming Languages, POPL, January 26-28, 2011*, pages 215–226. ACM.
6. Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *Proceedings of the 21ste European Symposium on Programming Languages and Systems, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012*, volume 7211 of *Lecture Notes in Computer Science*, pages 214–233. Springer.
7. Noah Van Es, Quentin Stiévenart, and Coen De Roover. Garbage-free abstract interpretation through abstract reference counting. In *33rd European Conference on Object-Oriented Programming, ECOOP , July 15-19, 2019*, volume 134 of *LIPIcs*, pages 10:1–10:33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. `doi:10.4230/LIPICS.ECOOP.2019.10`.
8. Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, January 21-23, 1987*, pages 314–325. ACM Press. `doi:10.1145/41625.41654`.
9. Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN International C onference on Functional Programming*, pages 48–59, 2002.
10. Kimball Germane and Jay McCarthy. Newly-single and loving it: improving higher-order must-alias analysis with heap fragments. *Proceedings of the ACM in Programming Languages*, 5(ICFP):1–28, 2021. `doi:10.1145/3473601`.
11. Dionna Glaze, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. *SIGPLAN Notices*, 48(9):443–454, September 2013. `doi:10.1145/2544174.2500604`.
12. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV), June 22-25, 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer. `doi:10.1007/3-540-63166-6\_10`.
13. David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International C onference on Functional Programming, ICFP 2010, September 27-29, 2010*, pages 51–62. ACM. `doi:10.1145/1863543.1863553`.
14. Bertrand Meyer. Design by contract: The Eiffel method. In *26th International Conference on Technology of Object-Oriented Languages and Systems (Tools), 1998*, page 446. IEEE Computer Society.
15. Matthew Might and Olin Shivers. Improving flow analyses via GammaCFA: abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International C onference on Functional Programming, ICFP, September 16-21, 2006*, pages 13–25. ACM. `doi:10.1145/1159803.1159807`.

16. Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *Proceedings of the ACM in Programming Languages*, 2(POPL):51:1–51:30, 2018. `doi:10.1145/3158139`.
17. Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional programming, September 1-3, 2014*, pages 139–152. `doi:10.1145/2628136.2628156`.
18. Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. Effect-driven flow analysis. In *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation , VMCAI, January 13-15, 2019*, volume 11388 of *Lecture Notes in Computer Science*, pages 247–274. Springer. `doi:10.1007/978-3-030-11245-5\_12`.
19. Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP, May 18-21, 2020*, pages 1661–1677. IEEE. `doi:10.1109/SP40000.2020.00024`.
20. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, September 5-9, 2005*, pages 263–272. `doi:10.1145/1081706.1081750`.
21. Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. Multise: multipath symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, August 30 - September 4, 2015*, pages 842–853. ACM.
22. Quentin Stiévenart, Noah Van Es, Jens Van der Plas, and Coen De Roover. A parallel worklist algorithm and its exploration heuristics for static modular analyses. *Journal of Systems and Software*, 181:111042, 2021. `doi:10.1016/J.JSS.2021.111042`.
23. Ignacio Tiraboschi, Tamara Rezk, and Xavier Rival. Sound symbolic execution via abstract interpretation and its application to security. In *Proceedings of the 24th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023*, volume 13881 of *Lecture Notes in Computer Science*, pages 267–295. Springer. `doi:10.1007/978-3-031-24950-1\_13`.
24. Bram Vandenbogaerde, Quentin Stiévenart, and Coen De Roover. Summary-based compositional analysis for soft contract verification. In *Proceedings of the 22nd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*, pages 186–196. `doi:10.1109/SCAM55253.2022.00028`.
25. Bram Vandenbogaerde, Sarah Verbelen, Noah Van Es, and Coen De Roover. Monarch: A modular framework for abstract definitional interpreters in Haskell. In *Proceedings of the 32nd International Symposium on Static Analysis, SAS, October 12-18, 2025*, Lecture Notes in Computer Science. Springer.
26. Maarten Vandercammen and Coen De Roover. State merging for concolic testing of event-driven applications. *Science of Computer Programming*, 242:103264, 2025. `doi:10.1016/J.SCICO.2025.103264`.