Summary-Based Compositional Analysis for Soft Contract Verification

Bram Vandenbogaerde, Quentin Stiévenart, Coen De Roover
Vrije Universiteit Brussel
{bram.vandenbogaerde, quentin.stievenart, coen.de.roover}@vub.be

***Abstract*—Design-by-contract is a development best practice that requires the interactions between software components to be governed by precise specifications, called contracts. Contracts often take the form of pre- and post-conditions on function definitions, and are usually translated to (frequently redundant) run-time checks. So-called soft contract verifiers have been proposed to reduce the run-time overhead introduced by such contract checks by verifying parts of the contracts ahead of time, while leaving those that cannot be verified as residual run-time checks. In the state of the art, static analyses based on the Abstracting Abstract Machines (AAM) approach to abstract interpretation have been proposed for implementing such soft verifiers. However, these approaches result in whole-program analyses which are difficult to scale.**

In this paper, we propose a scalable summary-based compositional analysis for soft contract verification, which summarises both the correct behaviour and erroneous behaviour of all functions in the program using symbolic path conditions. Information from these summaries propagates backwards through the call graph, reducing the amount of redundant analysis states and improving the overall performance of the analysis. This backwards flow enables path constraints associated with erroneous program states to flow to call sites where they can be refuted, whereas in the state of the art they can only be refuted using the information available at the original location of the error.

To demonstrate our improvements in both precision and performance compared to the state-of-the-art, we implemented our analysis in a framework called *MAF* (short for *Modular Analysis Framework*) — a framework for the analysis of higher-order dynamic programming languages. We conducted an empirical study and found an average performance improvement of 21%, and an average precision improvement of 38.15%.

***Index Terms*—Static Program Analysis, Design-by-contract**

I. Introduction

Design-by-contract, first proposed by Meyer et al. for Eiffel [1], is a development best practice that aims to render programs more robust by annotating their components with *contracts* that govern the component interactions. These contracts are typically enforced at function boundaries, where they take the form of pre- and post-condition constraints.

Contract languages have been an active area of research. In particular Findler et al. [2] have proposed contract languages for higher-order programming languages such as Scheme. Higher-order programs pose an additional challenge. Next to regular values (such as booleans, strings, lists, ...), functions can also be provided to other functions as an argument. These functional values can also be described by a contract, meaning that not all contracts can be checked upon function application, but need to be *delayed* until the functional value is used.

Racket, a dialect of Scheme, features an expressive contract language in which contracts can be combined using *contract combinators*. Because contracts are first-class values in Racket, these contract combinators can be implemented using ordinary functions. For example, in the listing below a function named *or/c* is depicted, which takes two contracts and returns a new `flat` contract whose satisfiability is described using a boolean predicate by `check`ing the input contracts and combining their boolean results using an `or`.

```
(define (or/c contract1 contract2)
 (flat (lambda (v)
    (or (check contract1 v) (check contract2 v)))))
```

The read-eval-print-loop interaction depicted below defines a function with an `or/c` contract specifying that its argument `x` should be either a number or a string (line 3-5). Calling the function with the boolean value `#f` (false), which does not satisfy the contract, results in a *contract violation* (line 6). Associated with a contract violation is a *blame object*, which captures the parties involved in the contract violation.

```
1  ; Accept a number or a string as argument
2  ; and can return anything.
3  > (define/contract (number-or-string x)
4         (-> (or/c number? string?) any/c)
5         '...)
6  > (number-or-string #f)
7  number-or-string: contract violation
8    expected: (or/c number? string?)
9    given: #f
10   in: the 1st argument of
11         (-> (or/c number? string?) any/c)
12   contract from: (function number-or-string)
13   blaming: anonymous-module
14    (assuming the contract is correct)
15   at: 3-unsaved-editor:3:18
```

The standard library of Racket has contracts annotated on all of its functions, making their usage robust against programmer errors. Dimoulas et al. [3] demonstrate the expressiveness of Racket's contract system outside of its standard library by extending it with support for specifying protocols, restrictions on (in)direct method calls, etc.

Unfortunately, enforcing such expressive contracts at run time introduces an additional overhead [4]. Static analyses have therefore been proposed, called *soft contract verifiers* [5], [6], to verify the program against these contracts ahead of time in order to elliminate its run-time contract checks. The emphasis is on the *soft verifier* part, meaning that it tries

Partially funded by Research Foundation Flanders (FWO) (grant number 1187122N).

to verify as many contract checks as possible. However, to ensure program correctness, the contract checks that cannot be verified are left as residual checks in the program, to be verified at run time. In the remainder of the paper, we will refer to 'veryifying the program against its contracts' as 'veryifying the contract checks'.

Soft contract verification in the state of the art [6] is based on abstracting abstract machines [7] (or *AAM*), which is a systematic approach to transforming the abstract machine describing the concrete semantics of a programming language into a static analysis. Classical AAM-style analyses can be classified as *whole-program* and *top-down*, which renders them difficult to scale because of the large number of program paths to be considered (i.e., due to their whole-program nature), and few opportunities for re-using earlier analysis states (i.e., due to their top-down nature). Therefore, real-world implementations often attempt to *modularise* the AAM-style analysis to improve its scalability.

The state-of-the-art soft contract verification analysis [6] is modular in the sense that contract-annotated functions can be analysed independently, but the the analysis of those functions continues in a whole-program fashion.

A static analysis is *modular* if it splits the program into multiple components (e.g., functions) and analyses them independently. Whereas an analysis is *compositional* [8] if the analysis results of these components can be combined without requiring a re-analysis of these components.

In this paper we propose a soft contract verification analysis that is both modular and compositional. The analysis consists of two phases: a modular call graph generation phase, and a compositional blame and path propagation phase. The first phase computes a call graph and summarises the behaviour of each function by performing an effect-driven modular analysis of the program with limited precision. The second phase subsequently performs a bottom-up compositional analysis that propagates contract violations (blames) and paths backwards from the callees to the callers. The use of function summaries results in a reduction of the number of analysis states and enables re-use of analysis results. We evaluate our approach on a set of benchmark programs and observe that our approach improves the precision by **38.15%** and performance by **21%** on average compared to a base analysis similar to the state-of-the-art in its characteristics (see Section V). In summary, our contributions are as follows:

1) We reformulate the state-of-the-art soft contract verification in the setting of a modular effect-driven analysis called ModF.
2) Then, we render this analysis compositional using *function summaries*, which summarise functions using its paths and erroneous behaviour. These function summaries are then propagated backwards from callee to caller to propagate paths and refute erroneous program states on function call-sites.
3) We implement our approach in an open source static analysis framework[1], and evaluate it against a set of benchmark programs for which we observe an improvement in precision by **38.15%** and in performance by **21%** on average.

II. Background and Motivation

A. Symbolic Verification using Abstraction

In symbolic execution [9], as opposed to concrete execution, the program under analysis is executed using symbolic inputs. For example a symbolic execution of the function `(lambda (x) (+ x 1))` yields $X + 1$ with X being a symbolic variable corresponding to the function argument, as opposed to a concrete numerical value. Programs typically contain a set of *branching points* (e.g, `if` statements). These statements require the introduction of a *path condition* that captures the constraints that need to be satisfied in order to reach a particular program state. The path condition typically consists of a series of conjoined predicates on symbolic variables and arithmetic expressions. A symbolic executor exhaustively explores all possible program paths, while keeping track of this path condition for each explored state. If, at any time, the path condition becomes unsatisfiable (i.e., there is no assignment of symbolic variables that satisfies the predicates in the path condition), the exploration of that path is aborted.

Symbolic execution has been shown to be a good fit for soft contract verification; contracts encode a set of constraints on *unknown* symbolic inputs. For example, the `(or/c string? number?)` contract encodes three paths (for all X): (a) $\mathit{string?}(X)$ (b) $\neg \mathit{string?}(X) \land \mathit{number?}(X)$ and (c) $\neg \mathit{string?}(X) \land \neg \mathit{number?}(X)$. These paths can then be used to verify whether other contracts can be violated by the values satisfying the constraints on this path. For example, when using X as part of an arithmetic expression on path (b), it would be impossible to violate the requirements of the arithmetic operation, since X is guaranteed to be a number. Unfortunately, it quickly becomes apparent that the number of program paths grows exponentially. A full symbolic execution of a program is therefore unpractical, and would take a significant amount of time.

Finally, it is well known that symbolic execution is not guaranteed to terminate. In fact, it explores all possible program paths, of which there can be infinitely many.

In this paper, we address the exponential explosion problem by using function summaries with accompanying composition rules, as well as the termination problem by using abstractions in cyclic function calls.

B. Bottom-Up Two-Phase Compositional Analysis

A compositional analysis usually proceeds in a *bottom-up* manner. This means that it analyses components (i.e., functions) completely independently, without taking their invocation context (i.e., call-site) into account. The behaviour of each component is then summarised in terms of its input, output and its effects.

[1] The implementation of the approach presented in this paper is also open source and available at https://github.com/softwarelanguageslab/maf.

The behaviour of a single function often depends on the functions that it calls. The summaries of two functions therefore need to be *composed* together. This composition needs to happen in the correct order for the semantics of the program to be captured properly. A call graph is typically used to obtain this order by topologically sorting its nodes (i.e., functions). As topological sorting requires a *directed acyclic graph*, cycles in the call graph must be collapsed first. The resulting order takes the calling relation between functions into account, such that callees are analysed before their callers.

Unfortunately, in dynamic higher-order programming languages such as Racket or Scheme, no call graph is available before the analysis is ran. We therefore propose an analysis that consists of *two phases*. The first phase performs an effect-driven abstract interpretation [10] to build an approximation of the call graph. Then, the second phase (also called *propagation phase*), propagates blames and path conditions (where applicable) based on a topological order of the call graph.

C. Function Summaries using Symbolic Execution

Contracts are often reused across a program's source code. Listing 1 depicts a contract on a *struct*. The `struct/c` contract combinator can be defined as an `and/c` contract that combines constraints on the type of the value (i.e., a struct of type *block*) and on the fields of the struct. The constraints on a symbolic value introduced by such a contract only need to be generated once, after which they can be reused and integrated with constraints on contract check-sites.

```
; Represents a block in 3D-space
(struct block (x y z))
; A value satisfying this contract is a struct of type
; "block" that has three real numbers as its fields
(define block/c (struct/c block real? real? real?))
```

Listing 1: An example of a reusable struct contract

The aim of a function summary is to describe a sound and reusable approximation of a function's possible behaviour. Symbolic execution can be used to construct such a summary, by calling the function with symbolic variables (e.g., X_i) for its arguments and exhaustively exploring all execution paths within. For the above `block/c` contract, the constraints resulting from such a symbolic execution are $\mathit{struct?}(X) \land \mathit{real?}(X.0) \land \mathit{real?}(X.1) \land \mathit{real?}(X.2)$, where $X.i$ indicates the ith field of the structure represented by X.

D. Backwards Blame and Path Propagation

During symbolic execution, paths are typically propagated forward, meaning that previously-encountered paths form a part of the abstract program state that is under analysis. Unfortunately, this may result in the exploration of redundant program states. Consider the `factorial` function depicted in Listing 2. Its argument has to be a number, so that contracts on primitive arithmetic operators within its body (e.g., +, −, ...) can be satisfied. The definition of the `factorial` function has been annotated with the `(-> number? number?)` contract to this end, stipulating that its argument and return value should be a number.

```
(define/contract (factorial n)
  (-> number? number?)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Listing 2: Example contract for `factorial` function.

A forward propagating analysis would include the path condition $\mathit{number?}(X)$ in the initial abstract program state for the `factorial` function, rendering the analysis *path sensitive*. Unfortunately, path-sensitive analyses tend to suffer from an exponential growth in the number of components. However, for soft contract verification, the interesting property is whether contract violations (which result in blames) can occur. The `factorial` function can therefore be summarised using the paths through the function, and importantly, using its possible contract violations paired with the paths on which they occur.

For example, for the `factorial` function in Listing 2, a contract violation summary for the $\star$ operator would be generated that has the following constraints: $\neg\mathit{number?}(X) \land \neg(X = 0)$. The first part of the conjunction stems from the $\mathit{number?}$ constraint introduced by the $\star$ expression, while the second is introduced by negation of the condition in the *if*-expression. Essentially, this summary denotes that a contract violation would occur *if* X is neither a number, nor equal to zero. If any of these constraints can be refuted, then the contract violation cannot occur.

Unfortunately, these constraints cannot be refuted at the *call site* of $\star$. Therefore the blame must be propagated backwards along the call stack of the `factorial` function. In this example, the propagation ends at the entry point of the analysis (called `main`), where the constraints of the `(-> number? number?)` contract result in an unsatisfiable set of constraints (i.e., $\mathit{number?}(X) \land \neg\mathit{number?}(X)$).

III. The λ_{SCV} Language

Without loss of generality, we will present our compositional approach to soft contract verification for λ_{SCV}, a didactic higher-order language featuring Racket-style contracts. We formally define its syntax and concrete semantics in this section. Section IV continues with our soft verification approach for the λ_{SCV} contracts.

A. Syntax

The syntax of λ_{SCV} is depicted in Fig. 1. The language features literals, such as strings, numbers and symbols. λ_{SCV} follows the λ-calculus in terms of λ-abstraction and λ-application (i.e., second line of the figure). For simplicity, λ_{SCV}'s functions only support a single argument. However, this restriction can easily be lifted, and its syntax and semantics generalised. λ_{SCV} also features support for conditionals in the form of **if** expressions.

For specifying contracts, λ_{SCV} supports the **flat**, and → special forms. The former wraps a boolean-valued function that determines contract conformance or violation into a *flat contract* (cf. `or/c` contract in Section I). The latter describes a

$$\begin{aligned} e \in \mathit{Expr} ::=\ & l \\ \mid\ & \lambda x.e \ \mid\ e\ e \\ \mid\ & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \\ \mid\ & \mathbf{flat}\ e \mid \mathbf{mon}\ e\ e \mid e \to e \\ l \in \mathit{Literal} ::=\ & \mathbb{N} \mid \mathbb{R} \mid \mathit{String} \mid \mathit{Symbol} \mid \mathit{Boolean} \end{aligned}$$

Fig. 1. Syntax of the λ_{SCV} language

higher-order contract, specifying a contract on the domain (the arguments), and the range (the return value) of the function.

The purpose of **mon** is two-fold: it can be used to check a flat contract against a value, or to install a monitor on a function. A *monitored* function (as depicted in Listing 2, is a function that checks, upon its application, whether its call-site arguments satisfy its domain contract (d), and whether its return value satisfies its range contract (r). A monitored function is therefore described by a value (arr given below) that wraps a function with its contract (described by a grd value).

B. Semantics

We define the semantics of λ_{SCV} using a so-called *CESK* state machine [11], the states of which consist of four components: a *C*ontrol, *E*nvironment, *S*tore and *K*ontinuation. The control component denotes whether the machine should evaluate an expression, apply a continuation, or throw an error. We omitted the definition of continuations for brevity. The environment consists of a mapping from variables to addresses, while the store is a mapping from these addresses to actual values. Finally, the continuation component corresponds to a stack of continuations, used in the evaluation of compound expressions. A transition relation $a \to b$ determines which steps from machine state a to b are valid.

$$\begin{aligned} & \zeta \in \mathit{State} ::= \langle c, s, \kappa \rangle \\ & k \in \mathit{Continuation} ::= ifk \mid monk1 \mid \ldots \\ & c \in \mathit{Control} ::= \mathsf{ev}(e, \rho) \mid \mathsf{ap}(v) \mid \mathsf{error}(\mathit{blame}) \\ & s \in \mathit{Store} ::= \mathit{Address} \to \mathit{Value} \\ & v \in \mathit{Value} ::= \mathit{Closure} \mid \mathit{ContractValue} \mid \mathit{Int} \mid \mathit{Boolean} \\ & \mathit{Closure} ::= \mathsf{clo}(\lambda x.e, \rho) \\ & \mathit{ContractValue} ::= \mathsf{grd}(d \to r) \mid \mathsf{arr}(d \to r, v_f) \\ & \rho \in \mathit{Environment} ::= \mathit{Identifier} \to \mathit{Address} \\ & \kappa ::= k :: \kappa \mid \mathsf{hlt} \end{aligned}$$

Halting States The machine reaches a halting state when an error occurs, or when the hlt continuation is applied. In λ_{SCV}, errors only occur when a contract violation is detected. These kind of errors are called *blames*. A blame includes two pieces of information: the party that is *to blame* for the contract violation, and the party that causes the blame to be registered. The former party usually points to the location of a call site in case a contract on the arguments of a function is violated, or to a function body in case the range contract is violated. The latter party usually points to the violated contract itself.

Atomic Evaluation Lambda expressions and literals can be evaluated in a single step. We call this *atomic evaluation*. As described by the Abs rule given below, a lambda expression evaluates to a closure that captures its definition environment, and keeps track of its parameter and body. The evaluation of literals is depicted in the Lit rule.

$$\begin{aligned} &\textsc{Abs} \quad \langle \mathsf{ev}(\lambda x.e, \rho), s, \kappa \rangle \to \langle \mathsf{ap}(\mathsf{clo}(\lambda x.e, \rho)), s, \kappa \rangle \\ &\textsc{Lit} \quad \langle \mathsf{ev}(l, \rho), s, \kappa \rangle \to \langle \mathsf{ap}(l), s, \kappa \rangle \end{aligned}$$

Conditionals For evaluating **if** expressions, we introduce three transition rules: IfCnd, IfCsq, and IfAlt. The former evaluates the condition of the if expression to a boolean, and the latter two evaluate either the consequent or the alternative based on the value of the evaluated condition.

$$\begin{aligned} &\textsc{IfCnd} \\ &\langle \mathsf{ev}(\mathbf{if}\ e_1\ then\ e_2\ else\ e_3, \rho), s, \kappa \rangle \to \langle \mathsf{ev}(e_1, \rho), s, \mathsf{ifk}\ e_2\ e_3\ \rho :: \kappa \rangle \\ &\textsc{IfCsq} \\ &\langle \mathsf{ap}(true), s, \mathsf{ifk}\ e_2\ e_3\ \rho :: \kappa \rangle \to \langle \mathsf{ev}(e_2, \rho), s, \kappa \rangle \\ &\textsc{IfAlt} \\ &\langle \mathsf{ap}(false), s, \mathsf{ifk}\ e_2\ e_3\ \rho :: \kappa \rangle \to \langle \mathsf{ev}(e_3, \rho), s, \kappa \rangle \end{aligned}$$

Function Application Rules App, App1 and App2 define the semantics for function application. Function application proceeds as usual by first evaluating the operator and operands, followed by an application of the resulting closure to the resulting operand values.

$$\begin{aligned} &\textsc{App1} \quad \langle \mathsf{ev}(e_1 e_2, \rho), s, \kappa \rangle \to \langle \mathsf{ev}(e_1, \rho), s, \mathsf{appk1}(e_2, \rho) :: \kappa \rangle \\ &\textsc{App2} \quad \langle \mathsf{ap}(v), s, \mathsf{appk1}(e_2, \rho) :: \kappa \rangle \to \langle \mathsf{ev}(e_2, \rho), s, \mathsf{appk}(v) :: \kappa \rangle \\ &\textsc{App} \\ &\frac{\alpha = alloc \qquad \rho'' = \rho'[x \mapsto \alpha] \qquad s' = s[\alpha \mapsto v_a]}{\langle \mathsf{ap}(v_a), s, \mathsf{appk}(\mathsf{clo}(\lambda x.e, \rho') :: \kappa \rangle \to \langle \mathsf{ev}(e, \rho''), s', \kappa \rangle} \end{aligned}$$

Contract Monitoring λ_{SCV} features two types of contracts: *flat* contracts which serve as wrappers around a boolean-valued function, and *guard* contracts which can be used to guard the execution of a function. Rules MonFlat, MonFlatTrue and MonFlatBlame cover the evaluation and checking of flat contracts. When a flat contract is violated (i.e., it evaluates to false), an error $\mathsf{error}(blm)$ is produced, containing a *blame object* (*blame*) that is computed as usual [14].

$$\begin{aligned} &\textsc{Mon} \quad \langle \mathsf{ev}(\mathbf{mon}\ e_1\ e_2, \rho), s, \kappa \rangle \to \langle \mathsf{ev}(e_1), s, \mathsf{monk1}(e_2, \rho) :: \kappa \rangle \\ &\textsc{MonVal} \quad \langle \mathsf{ap}(v_c), s, \mathsf{monk1}(e_2, \rho) :: \kappa \rangle \to \langle \mathsf{ev}(e_2), s, \mathsf{monk2}(v_c) :: \kappa \rangle \\ &\textsc{MonFlat} \\ &\langle \mathsf{ap}(v), s, \mathsf{monk2}(\mathsf{flat}(p)) :: \kappa \rangle \to \langle \mathsf{ap}(v), s, \mathsf{appk}(p) :: \mathsf{monk3}(v) :: \kappa \rangle \\ &\textsc{MonFlatTrue} \\ &\langle \mathsf{ap}(true), s, \mathsf{monk3}(v) :: \kappa \rangle \to \langle \mathsf{ap}(v), s, \kappa \rangle \\ &\textsc{MonFlatBlame} \\ &\langle \mathsf{ap}(false), s, \mathsf{monk3}(v) :: \kappa \rangle \to \langle \mathsf{error}(blm), s, \kappa \rangle \end{aligned}$$

MonArr describes the evaluation of a contract-monitored function, which evaluates to an arr value that wraps the function with a guard (grd) contract. Rules ArrApp, ArrDom and ArrRet govern the application of monitored functions. First, the domain contract is checked against the argument of the function. Next, the body of the function is executed, and finally the return contract is checked against the return value of the function.

MonArr

$$\langle \mathsf{ap}(v_f), s, \mathsf{monk2}(\mathsf{grd}(d \to r)) :: \kappa \rangle \to \langle \mathsf{ap}(\mathsf{arr}(d \to r, v_f)), s, \kappa \rangle$$

ArrApp

$$\frac{\kappa' = s, \mathsf{monk2}(d) :: \mathsf{arr1}(r, v_f) :: \kappa}{\langle \mathsf{ap}(v_a), s, \mathsf{appk}(\mathsf{arr}(d \to r), v_f) :: \kappa \rangle \to \langle \mathsf{ap}(v_a), \kappa' \rangle}$$

ArrDom

$$\langle \mathsf{ap}(v_a), s, \mathsf{arr1}(r, v_f) :: \kappa \rangle \to \langle \mathsf{ap}(v_a), s, \mathsf{appk}(v_f) :: \mathsf{arr2}(r) :: \kappa \rangle$$

ArrRet

$$\langle \mathsf{ap}(v_r), s, \mathsf{arr2}(r) :: \kappa \rangle \to \langle \mathsf{ap}(v_r), s, \mathsf{monk2}(r) :: \mathsf{arr3}(v_r) :: \kappa \rangle$$

IV. A Static Analysis for λ_{SCV}

We now propose a modular and compositional static analysis for soft verification of λ_{SCV} contracts. The proposed analysis consists of two phases, both of which are constructed as a ModF-style analysis [10]:

- **Collection Phase**: constructs function summaries for each function in the analysed program. A function summary consists of a path summary and a blame summary. All of a function's execution paths are summarised as path conditions computed through symbolic execution (cf. Section II-A). The potential blames during the function's executions are summarised by recording the program paths on which they might occur. Additionally, the collection phase constructs a *call graph* which is used during the propagation phase to compute an efficient propagation schedule.
- **Propagation Phase**: the propagation phase propagates function summary information from callee to callers. Blames and path conditions therefore propagate *backwards* which enables the analysis to re-evaluate the necessary conditions for a blame at each call site in the approximated call stack.

The remainder of this section is structured as follows. We first describe our base analysis ScvModF. Then we formally describe the collection phase in terms of this base analysis. We conclude by formally describing the propagation phase.

A. Base Analysis: ScvModF

Approach Our analysis derives from ModF [10], an effect-driven modular analysis. ModF derivatives target programs written in higher-order dynamically-typed languages. Therefore, no static call graph is available before running the analysis. Consequently, ModF is only modular in the sense that it analyses each component separately (i.e., function calls), but discovers these components and the dependencies between them during the analysis itself. ModF reifies dependencies between components, and between components and addresses in the store using *effects* (i.e., call effects and read/write effects). A ModF analysis consists of two interleaving analyses: an intra-analysis, which analyses a single component, and an inter-analysis which schedules components for (re-) analysis based on the effects discovered during the previous intra-analysis. The result of a ModF analysis is a *call and dependency graph*. In the remainder of this section, we describe the changes required to convert the concrete small-step λ_{SCV} semantics presented in Section III-B into a sound and terminating ModF-based static analysis. We will highlight these changes in gray.

Abstract Values A sound static analysis is necessarily imprecise. Abstract values instead of concrete values are used to account for unknown inputs and non-determinism. For the purpose of soft contract verification, we use a product lattice [12] as abstract value domain. Its factors correspond to a constant propagation domain $CP(Int)$ for integers, a constant propagation $CP(Boolean)$ for booleans and two powerset lattices: one for closures and one for contract values.

$$\begin{aligned} v \in \mathit{Value} ::= \ & CP(\mathit{Int}) \times CP(\mathit{Boolean}) \\ & \times \mathcal{P}(\mathit{Closure}) \times \mathcal{P}(\mathit{ContractValue}) \end{aligned}$$

Components A ModF analysis runs an intra-analysis on a single *component* at a time. In ModF, a component corresponds to a *function* with an additional context parameter (e.g., one of the function's call sites). For simplicity, we omit this context from the formalisation, but it can be used to render the analysis context-sensitive and thus improve its precision.

$$\mathit{cmp} \in K ::= (\lambda x.e, \mathit{env})$$

Global Store Instead of keeping track of a local store in each analysis state $\hat{\zeta}$, each analysis state points to a single global store $\hat{s}$. At each analysis step, the next reachable stores are joined with the previous global store (see intra-analysis function $\mathcal{F}$ below). The transition relation $\to$ is updated accordingly, and now includes the effects produced by a single step of the intra-analysis.

$$(\to) \in (\widehat{\mathit{State}} \times \widehat{\mathit{Store}} \times \mathcal{P}(\mathit{Effect})) \times (\widehat{\mathit{State}} \times \widehat{\mathit{Store}} \times \mathcal{P}(\mathit{Effect}))$$

$$E \in \mathit{Effect} ::= \mathsf{read}(\alpha) \mid \mathsf{write}(\alpha) \mid \mathsf{call}(\mathit{cmp})$$

Symbolic Store and Path Condition We augment the evaluation state with an additional component Γ which consists of a path condition φ (cf. Section II-A) which collects the constraints on the path leading to this state, and a symbolic store m that contains mappings from global store addresses to their corresponding symbolic representations.

$$\hat{\zeta} \in \widehat{\mathit{State}} ::= \langle c, \Gamma, \kappa \rangle$$

$$\Gamma \in \mathit{SymbolicState} ::= (\varphi, m)$$

$$\varphi \in \mathit{PathCondition} ::= \bigwedge_{i=0}^{n} p_i(x_1, x_2, \ldots, x_m)$$

$$m \in \mathit{SymbolicStore} ::= \mathit{Address} \to \mathit{Expr}$$

A path condition φ is represented by a conjunction of predicates $p_1, p_2, \ldots, p_n$ on symbolic expressions $x_1, x_2, \ldots, x_n$.

Post-Values In a soft contract verification analysis [6], a limited form of symbolic execution is used to obtain constraints on the input of the program. To enable this, we augment the ap control component with a symbolic representation of its returned value. Formally: $c \in Control ::= \mathsf{ev}(e, \rho) \mid \mathsf{ap}(v, e) \mid \mathsf{error}$

In practice, symbolic representations can be produced by applications of primitive procedures on values with symbolic representations. For instance, `(+ x 1)` might result in the symbolic expression `(+ X 1)` if $\alpha_x \mapsto X$ is in the symbolic store.

Collecting constraints Constraints are added to the path condition when an **if**-expression is evaluated, and when an expression monitored by a flat contract is evaluated. Depending on the branch to evaluate, the path condition is either augmented with the symbolic representation of the if-expression predicate, or with its negation. Similarly, the path condition is extended with the symbolic representation of the flat contract whenever a mon expression applies it to a value. Note that transition rule MonFFlatBlame produces an error state, but still extends the state's path condition so it can be included in the blame summary.

IfCsq

$$\langle \mathsf{ap}(true, e), \Gamma, \mathsf{ifk}\ e_1\ e_2\ \rho :: \kappa\rangle, s, E \rightarrow \langle \mathsf{ev}(e_1, \rho), \Gamma', \kappa\rangle, s, E$$
$$\text{where } (\varphi, m) = \Gamma, \Gamma' = ((\varphi \wedge e), m)$$

IfAlt

$$\langle \mathsf{ap}(false, e), \Gamma, \mathsf{ifk}\ e_1\ e_2\ \rho :: \kappa\rangle, s, E \rightarrow \langle \mathsf{ev}(e_2, \rho), \Gamma', \kappa\rangle, s, E$$
$$\text{where } (\varphi, m) = \Gamma, \Gamma' = ((\varphi \wedge \neg e), m)$$

MonFlatTrue

$$\langle \mathsf{ap}(true, e), \Gamma, \mathsf{monk3}(v) :: \kappa\rangle \rightarrow \langle \mathsf{ap}(v, e), \Gamma' \kappa\rangle$$
$$\text{where } (\varphi, m) = \Gamma, \Gamma' = ((\varphi \wedge e), m)$$

MonFlatBlame

$$\langle \mathsf{ap}(false, e), \Gamma, \mathsf{monk3}(v) :: \kappa\rangle \rightarrow \langle \mathsf{error}(blm), \Gamma' \kappa\rangle$$
$$\text{where } (\varphi, m) = \Gamma, \Gamma' = ((\varphi \wedge \neg e), m)$$

Intra-Analysis The intra-analysis computes all states reachable from a given initial state. To this end, a *transfer function* $\mathcal{F}$ (parameterized by a component cmp and the initial global store s_0) can be defined that given a set of reachable states, a global store, and a set of effects, computes a new set of reachable states. This triple consisting of a set of states, a global store and a set of effects form a lattice too, where the join operator is defined component-wise. The global store is treated as a map lattice: values with corresponding keys are joined together. The path condition φ is used after each analysis step $\rightarrow$ to determine whether the path leading up to that analysis state is feasible. The intra-analysis can then be defined as a least fixed point of the transfer function. The transfer function is guaranteed to reach a fixed point since a state is only reachable from itself for recursive function applications which are handled by the inter-analysis. The intra-analysis uses the return value of the called function that was computed during its early intra-analysis (if any, otherwise $\bot$).

$$\mathcal{F}^{s_0}_{cmp}(S, s, E) = \langle\{\hat{\zeta_0}\}, \hat{s_0}, E\rangle \sqcup \bigsqcup_{\substack{\zeta \in S \\ \zeta, s, E \rightarrow \zeta', s', E' \\ \mathsf{isReachable}(\zeta')}} \langle \zeta', s', E'\rangle$$
$$Intra(cmp, s_0) = lfp(\mathcal{F}^{s_0}_{cmp})$$
$$\mathsf{isReachable}(\langle c, (\varphi, m), \kappa\rangle) = feasible(\varphi)$$
$$\zeta_0 = \langle \mathsf{ev}(e_{cmp}), \Gamma_0, \mathsf{ret}\rangle$$

Inter-Analysis The inter-analysis collects the effects generated by the intra-analysis, and schedules the re-analysis of the potentially impacted components. It is defined as a function $\mathcal{G}$ parametrized by a worklist W, to keep track of which components to analyse next, a global store s, a map S of components to reachable states ($K \rightarrow \mathcal{P}(State)$), and a map R of addresses to their read dependencies ($Address \rightarrow \mathcal{P}(K)$). The result of $\mathcal{G}$ is an over-approximation of all reachable program states.

$$\mathcal{G}(W, s, R, S) = \begin{cases} S & \text{if } W = \emptyset \\ \mathcal{G}(W \setminus \{cmp\} & \cup \bigcup_{\substack{\mathsf{call}(c) \in E \\ c \notin dom(S)}} c \\ & \cup \bigcup_{\substack{\mathsf{w}(\alpha) \in E \\ c \in R(\alpha)}} c, \\ & s \sqcup s', \\ & R \sqcup \bigsqcup_{\mathsf{r}(\alpha) \in E} [\alpha \mapsto \{cmp\}], \\ & S[cmp \rightarrow S']) \text{ otherwise} \end{cases}$$
$$\text{where } cmp \in W$$
$$\langle S', s', E\rangle = Intra(cmp, s)$$

The inter-analysis $\mathcal{G}$ is defined as a recursive function that exhaustively explores the worklist until it is empty. Additionally, we define an auxiliary function $Inter$ that analyses a program e.

$$Inter(e) = \mathcal{G}(\mathcal{I}(e), s_0, \emptyset, \emptyset)$$

where $\mathcal{I}$ injects expression e into a component *Main*.

Detecting Contract Violations A contract violation is represented as an analysis state with an error for its control component. The inter-analysis function $\mathcal{G}$ yields the set of reachable states. Therefore, a contract violation is detected if such an error state is an element of the reachable states. Formally: $Blames \equiv \{b \mid \langle \mathsf{error}(b), \_, \_\rangle \in range(Inter(e))\}$

B. Collection Phase: Function Summary and Call Graph Construction

Function Summaries During the collection phase, we collect *function summaries* f_s of the analysed components. A function summary consists of a set of path conditions Φ associated with their return value, a set of pairs of blames and their paths $\mathcal{B}$, and the initial symbolic store m_0 which contains a symbolic representation for the language's primitives and for the function parameter.

$$\Phi ::= Value \rightarrow \mathcal{P}(PathCondition)$$
$$\mathcal{B} ::= Blame \rightarrow \mathcal{P}(PathCondition)$$
$$f_s \in FunctionSummary ::= (\Phi, \mathcal{B}, m_0)$$

This function summary can be constructed using our base intra-analysis. Given a component cmp and initial store s_0, the function summary f_s of cmp is computed as follows:

$$\langle S, s, E\rangle = \mathit{Intra}(cmp, s_0)$$
$$\Phi = \bigsqcup_{\langle \mathsf{ap}(v,\_),(\varphi,\_),\mathsf{ret}\rangle \in S} [v \mapsto \{\varphi\}]$$
$$\mathcal{B} = \bigsqcup_{\langle \mathsf{error}(b),(\varphi,\_),\_\rangle \in S} [b \mapsto \{\varphi\}]$$

Call Graph MODF keeps track of *call* dependencies during the intra-analysis. We adapt the inter-analysis function $\mathcal{G}$ to produce the call graph C ($cmp \to \mathcal{P}(cmp)$) and set of function summaries F ($cmp \to \mathit{FunctionSummary}$)) as an output of the analysis. The adapted intra-analysis function $\mathcal{G}$ is depicted in the equation below, the base case of the function $\mathcal{G}$ is omitted for brevity.

$$\mathcal{G}(W, s, R, S, C, F) = \mathcal{G}($$
$$\text{as before, } C \sqcup \bigsqcup_{\mathsf{call}(c) \in E} [cmp \to \{c\}], F[cmp \to f_s]$$
$$)$$

C. Propagation Phase: Function Summary Composition

During the propagation phase, information from function summaries propagates in the call graph opposite to the direction of the call edges. We augment the small-step relation to include a *function summary store* F which is represented by a mapping $F \in \mathit{FSS} ::= \mathcal{P}(K \to \mathit{FunctionSummary})$.

$$\begin{aligned}(\to) \in\ & (\mathit{State} \times \mathit{Store} \times \mathcal{P}(\mathit{Effect}) \times \mathit{FSS}) \\ & \times (\mathit{State} \times \mathit{Store} \times \mathcal{P}(\mathit{Effect}) \times \mathit{FSS})\end{aligned}$$

The closure application rules are adapted to integrate the function summary of the callee with the function summary of the caller. For this, we define the compose and propagate meta-functions. The former combines the path condition of the callee with that of the caller based on the component and symbolic store m_0 of the callee, and the symbolic store of the caller. The latter composes the set of blames together. Both meta functions rewrite the path conditions in such a way that corresponding symbolic expressions (for example on the arguments of a function) are correctly rewritten such that constraints on the argument values at the call site are linked to the constraints generated by analyzing the callee's function body. We leave the definition of these functions abstract such that they can be tuned to adapt the precision of the analysis.

We split the application rules into two: a regular function application rule, and a *blame summary propagation* rule. The former rule behaves similarly to the function application rule of the first analysis phase, but propagates path conditions from the called function using compose. The latter rule propagates blames from the callee to the caller, while rewriting the associated path conditions so that symbolic variables are correctly mapped to their corresponding symbolic representation at the call site.

APP

$$\frac{\begin{gathered}\alpha = \mathit{alloc}(\mathsf{clo}(\lambda x.e, \rho')) \qquad s' = s \cup s[\alpha \mapsto v_a] \\ E' = E \cup \{\mathsf{call}(\mathsf{clo}(\lambda x.e, \rho')), \mathsf{write}(\alpha)\} \\ (\Phi, \mathcal{B}, m_0) = F(\mathsf{clo}(\lambda x.e, \rho')) \qquad (v_r, \varphi_{cmp}) \in \Phi \\ \varphi' = \mathsf{compose}(\varphi_{cmp}, \varphi, \mathsf{clo}(\lambda x.e, \rho'), m, m_0)\end{gathered}}{\begin{gathered}\langle \mathsf{ap}(v_a), (\varphi, m), \mathsf{appk}(\mathsf{clo}(\lambda x.e, \rho') :: \kappa\rangle, s, E, F \\ \to \langle \mathsf{ap}(v_r), (\varphi', m), \kappa\rangle, s', E', F\end{gathered}}$$

APPBLAME

$$\frac{\begin{gathered}\alpha = \mathit{alloc}(\mathsf{clo}(\lambda x.e, \rho')) \qquad s' = s \cup s[\alpha \mapsto v_a] \\ E' = E \cup \{\mathsf{call}(\mathsf{clo}(\lambda x.e, \rho')), \mathsf{write}(\alpha)\} \\ (\Phi, \mathcal{B}, m_0) = F(\mathsf{clo}(\lambda x.e, \rho')) \qquad (v_r, \varphi_{cmp}) \in \Phi \\ (blm, \varphi_{blm}) \in \mathcal{B} \\ \varphi'_{blm} = \mathsf{propagate}(\varphi_{blm}, m, m_0) \qquad \varphi' = \varphi \wedge \varphi'_{blm}\end{gathered}}{\begin{gathered}\langle \mathsf{ap}(v_a), (\phi, m), \mathsf{appk}(\mathsf{clo}(\lambda x.e, \rho') :: \kappa\rangle, s, E, F \\ \to \langle \mathsf{error}(blm), (\varphi', m)), \kappa\rangle, s', E', F\end{gathered}}$$

Importantly, because of the isReachable(ζ) predicate in our intra-analysis function $\mathcal{F}$, blames with unsatisfiable paths are no longer propagated since their states become unreachable. The idea is that in safe programs (i.e., programs that do not contain any contract violation), the path condition associated with a particular blame *eventually* becomes unsatisfiable during its backwards propagation.

Intra-Analysis Function summaries are computed at the end of each intra-analysis. We therefore adapt our intra-analysis function $\mathcal{F}$ to include the function summary store as as an additional fixed parameter.

$$\mathcal{F}^{s_0, F}_{cmp}(S, s, E) = \langle \{\hat{\zeta_0}\}, \hat{s_0}, E\rangle \sqcup \bigsqcup_{\substack{\zeta \in S \\ \zeta, s, E, F \to \zeta', s', E', F \\ \mathsf{isReachable}(\zeta')}} \langle \zeta', s', E'\rangle$$

$$\mathit{Intra}(\zeta_0, s_0, F) = \mathit{lfp}(\mathcal{F}^{s_0, F}_{cmp})$$

Detecting Contract Violations Contract violations can no longer be collected by merely inspecting the set of reachable states. It might be possible to refute blames that propagate transitively from callee to caller at one of the call sites on the approximated call stack. Therefore we define a *contract violation boundary* at which blames become *real* errors, and count them as detected contract violations. In the context of MODF we define this boundary as the entry point of the program. At this boundary, blames can no longer be refuted, and count towards actual contract violations. What follows is an adapted definition of the set of contract violations:

$$Blames \equiv \{b \mid \langle \mathsf{error}(b), \_, \_\rangle \in \mathit{Inter}(e)(\mathit{Main})\}$$

Scheduling During the propagation phase, function summaries are updated to take blames and path conditions of the callee into account. To do so efficiently, we define a fixed order based on the topological sorting of the call graph C as computed during the collection phase. As the call graph may contain cycles, we collapse the cycles using Tarjan's strongly connected components algorithm. This sorting is then used as a schedule in the worklist of our inter analysis $\mathcal{G}$.

V. Implementation

We implemented our approach in the open-source static analysis framework MAF (Modular Analysis Framework) for an R5RS-complaint programming language extended with support for structures, modules and match expressions. We provide four different configurations: ScvModF, ScvModFSensitive, ScvModFSummary and ScvModFTopos.

All configurations share the same abstract domain: a product lattice where closures and contract values are represented by powerset lattices, and a constant propagation lattice otherwise. In contrast to the work by Nguyen et al. [6] we do not include predicate-refined abstract values, neither do we implement abstract operations on them. Improving the abstract domain in our implementation would yield similar precision results as Nguyen et al. [6]. However, the purpose of this paper is to showcase how different propagation strategies influence the precision in a purely symbolic reasoner.

The context sensitivity of our analysis is 0-m-cfa [13] for most function calls. An argument sensitivity is used when one of the arguments is a closure, meaning that a new component is created for each different closure value that is used as an argument. This is to ensure that path constraints propagate more precisely in higher-order functions.

ScvModF implements a baseline analysis (Section IV-A) that does not share path conditions, nor symbolic stores across function calls. Being almost equivalent to ModF [10], it shares its performance and precision characteristics.

ScvModFSensitive implements the approach from Nguyen et al. [6], differing in the aforementioned abstract domain semantics. It can be expressed in our formal semantics by keeping track of the path condition and symbolic store in the context of the component itself.

$$\mathit{cmp} \in K = (\lambda x.e, \mathit{env}, \varphi, m)$$

The intra-analysis function $\mathcal{F}$ then injects this path condition and symbolic store into its initial analysis state. Finally, to avoid non-termination, ScvModFSensitive reverts to ScvModF when analysing a recursive function.

ScvModFSummary and ScvModFTopos implement the approach proposed in this paper. The former uses ModF's effect system to schedule the analysis during the propagation phase, while the latter uses a topological sorting of the components. Both configurations share the same semantics, and therefore yield the same analysis results. However, the latter configuration is more likely to obtain an optimal schedule for analysing the components during the propagation phase.

VI. Evaluation

A. Experimental Setup

To evaluate our approach, we execute the four aforementioned analysis configurations (Section V) on a set of benchmark programs. We refer to these configurations by numbers: (1) is ScvModF, (2) is ScvModFSensitive, (3) is ScvModFSummary and (4) is ScvModFTopos. Our analysis is run on a 2015 Dell PowerEdge R730 with 2 Intel Xeon 2637 639 processors and 256GB of RAM, OpenJDK 1.8.0_312 and Scala 3.1.0. The JVM was given a maximum of 128GB RAM. We measured the following metrics:

TABLE I
Summary of our set of benchmark programs

Name	# Programs	# Lines (average)	# Contract Checks (median)
snake	1	255	408
tetris	1	471	857
zombie	1	271	178
mochi	15	15	33
sergey	6	17	22
softy	10	11	34.5

- **Running Time**: we measured the running time of each of the configurations, and compared them against each other. To this end, we ran the analysis on each benchmark program 5 times (as a warm-up phase for the Java Virtual Machine), after which the analysis is run 20 more times to obtain a statistically significant number of benchmark results.
- **Precision**: we measured the precision of the analysis in terms of the number of false positives (i.e., the number of detected contract violations that do not actually occur at run-time). As a ground truth, we used the benchmark suite by Nguyen et al. [6], which contains a set of "safe" benchmark programs which are known to not contain any contract violations. Any detected contract violation is therefore a false positive, which can be easily measured.

Benchmark Programs As a set of benchmark programs, we used a subset of the publicly available[2] benchmark programs by Nguyen et al. [6]. The benchmark set is divided into four groups: *games*, *softy*, *mochi* and *sergey*. The first set of benchmarks consists of three (larger) programs implementing some real-world games. The last three sets of benchmarks are frequently used for benchmarking *AAM*-based analyses, and were adapted by Nguyen et al. to contain meaningful contracts [14], [6]. Table I depicts a summary of the aforementioned benchmark sets. The table summarises the number of distinct programs in each benchmark set, as well as the average number of lines and contract checks in each program.

B. Results

Running Time Table II depicts the running time of the different analysis configurations. In bold are speedups that are greater than 0. For computing this speedup, we compare the ScvModFSensitive configuration against ScvModFTopos. As expected, the baseline analysis ScvModF has the fastest execution time, but the lowest precision (see below) for all benchmarks, because it does not share path conditions nor symbolic stores across function call boundaries. Overall, in **21 out of the 34** benchmark programs, our proposed approach using function summaries is faster than ScvModFSensitive, while still yielding an improvement in precision (see below). The average speedup is 21%, and the median speedup is 16%.

[2] As found in https://github.com/philnguyen/soft-contract, branch popl18-ae

TABLE II
ANALYSIS EXECUTION TIME (IN MILLISECONDS)

	Average Time (ms)				**Speedup**
	(1)	(2)	(3)	(4)	
games/snake.rkt	668	1538	3751	2769	-44.46%
games/tetris.rkt	5302	10118	7010	6642	**52.33%**
games/zombie.rkt	307	3878	3338	2148	**80.54%**
mochi/fold-div.rkt	83	192	129	127	**51.18%**
mochi/hors.rkt	92	222	119	119	**86.55%**
mochi/hrec.rkt	88	207	152	128	**61.72%**
mochi/l-zipunzip.rkt	98	188	146	146	**28.77%**
mochi/map-foldr.rkt	102	127	121	123	**3.25%**
mochi/mappend.rkt	93	145	124	122	**18.85%**
mochi/mem.rkt	154	778	3041	1944	-59.98%
mochi/mult.rkt	82	204	135	131	**55.73%**
mochi/neg.rkt	75	139	111	109	**27.52%**
mochi/nth0.rkt	73	112	88	88	**27.27%**
mochi/r-file.rkt	77	177	134	141	**25.53%**
mochi/r-lock.rkt	85	140	140	149	-64.00%
mochi/reverse.rkt	85	156	118	122	**27.87%**
mochi/sum.rkt	85	184	134	118	**55.93%**
mochi/zip.rkt	99	127	208	194	-34.54%
sergey/blur.rkt	105	89	110	98	-9.18%
sergey/eta.rkt	5	6	7	7	-14.29%
sergey/kcfa2.rkt	6	7	105	108	-93.52%
sergey/kcfa3.rkt	8	9	89	89	-89.89%
sergey/loop2.rkt	90	248	140	122	**103.28%**
sergey/mj09.rkt	85	72	84	80	-100.00%
sergey/sat.rkt	90	358	105	107	**234.58%**
softy/append.rkt	76	107	97	99	**88.00%**
softy/cpstak.rkt	96	384	162	145	**164.83%**
softy/last-pair.rkt	79	97	87	88	**10.23%**
softy/last.rkt	83	115	100	101	**13.86%**
softy/length-acc.rkt	104	114	152	154	-25.97%
softy/length.rkt	82	100	131	129	-22.48%
softy/member.rkt	4	5	8	7	-28.57%
softy/recursive-div2.rkt	102	102	128	130	-21.54%
softy/subst.rkt	120	291029	2776	1793	**16100%**
softy/tak.rkt	77	164	116	116	**41.38%**

We also compare the reduction in the number of components analysed by ScvModFSummary compared to the ScvModFSensitive configuration.This information is depicted in Fig. 2. Our findings are twofold: ScvModFTopos realises a reduction in the number of components, and this reduction leads to an improvement in execution time. The reduction in the number of components is due to the backwards (instead of forwards) propagation of path conditions, which means that a path condition is no longer part of the context of a component. This avoids analysing the same state multiple times while only differing in its path. Note that both analyses are sound (i.e., they overapproximate the set of reachable program states), but our approach achieves a reduction in the number of abstract analysis states.

Our ScvModFSummary analysis did not realise an improvement in terms of compositionality on our benchmark programs. Most functions are only called once (by the entrypoint of our analysis, or as part of the call chain), or twice (in case of recursive functions). The consequence of this is that the performance improvement is mostly due to the aforementioned reduction in the number of components. Finally, the propagation phase terminates quickly due to the information provided by the collection phase (86%-14% ratio in terms of execution time on average).

The intra-analysis of a component produces paths by evaluating control flow statements in that component, or by integrating paths from the summaries of called components using the **compose** rule. Therefore, our selection of the compose rule, which determines how path conditions are propagated, plays an important role in the performance of the analysis. Informed by our precision results (see below), our decision to opt for only propagating paths originating from contracts seems to be a reasonable approach that does not harm precision.

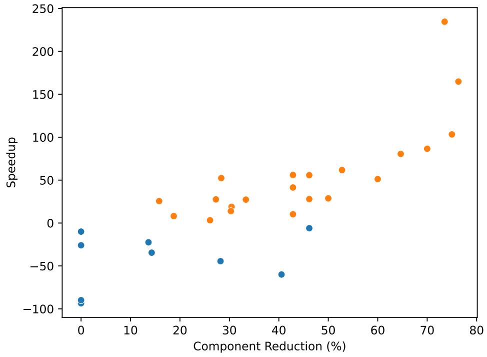

Fig. 2. The relation between component reduction and speedup. In blue, programs with negative speedup, in orange with positive speedup.

TABLE III
ANALYSIS PRECISION USING THE MEDIAN NUMBER OF FALSE POSITIVES, GIVEN IN BOLD WHICH PRECISION IS BEST

Name	(1)	(2)	(3) and (4)
snake	110	90	**62**
tetris	180	165	**112**
zombie	41	41	**11**
mochi	11	11	**4**
sergey	0	0	0
softy	12.5	8	**3**

Precision Table III depicts the precision of the benchmark programs. Overall, ScvModFSummary yields an improvement in precision compared to ScvModFSensitive. In contrast to ScvModFSensitive, path conditions associated with blames flow backwards through the call chain. This means that the feasibility of the contract violation can be checked in the context of one of its callers, while ScvModFSensitive checks the feasibility in its violation context, which is abstracted to the empty path condition in case of loops, thereby losing precision.

Summary Our compositional approach to soft contract verification yields improvements in both execution time and precision. ScvModFSummary realizes a reduction in the number of components by using function summaries and propagating the summarized program path conditions *backwards*. Furthermore, we realize a precision improvement by propagating blame summaries backwards from callee to caller, where more precise constraints on the input values are available.

C. Theoretical Properties

Soundness A *sound* soft contract verifier detects all contract violations in the program, but might yield a number of *false positives*. To test for the soundness of our implementation, we provide *soundness tests* [15], which run a concrete interpreter against the benchmark programs and check whether its results are subsumed by the static analysis. For generating test inputs, we used the contracts given in the benchmark programs, and generate values for them using Racket's built-in input generation facilities. These soundness tests yield an average line coverage of 85% on the benchmarked programs, where all results were subsumed by the static analysis.

Termination Our analysis is a variation of the MODF analysis [10], [16]. MODF is adapted in the following ways: 1) *Intra-procedural Symbolic Execution* During the abstract interpretation of a procedure, we keep track of a symbolic store and a path condition. The abstract representation of these additional components is unbounded, however they are only extended in interpretation steps of our intra-analysis. Since the size of the abstract syntax tree is finite, its traversal will also be finite. Recursive function calls are the only means of looping. During an intra-analysis, calls are immediately resolved to values. Therefore, an intra-procedural analysis never loops and always terminates in a finite number of steps. 2) *Two-phase analysis* Our analysis runs in two phases: a blame and path collection phase, and a propagation phase. The collection phase is guaranteed to terminate due to the fixpoint properties of MODF [10]. We presented two variants of our analysis: one that uses the effect system of MODF during the propagation phase, and another that uses a topological sorting of the call graph. The former triggers a (re-)analysis of components based on changes to propagate path conditions and blame summaries. Since path conditions between components that are part of a cycle are not propagated, the number of propagated path conditions remains finite. The number of blames in a program is fixed by the program's text, and so are its blame summaries. The latter variant schedules the analysis of a component by using a topological sorting of the call graph (with collapsed cycles) during the second phase. The number of components in this topological sorting is finite. Therefore, both variants of our analysis are guaranteed to terminate.

VII. Related Work

Soft Contract Verification The concept of soft contract verification was first introduced by Tobin-Hochstadt and Van Horn et al. [17], [5]. In these works the idea of *higher-order symbolic execution* is proposed where functions are symbolically represented by their contracts. In later work by Nguyen et al. [6] a static analysis based on the *AAM* technique [7] is proposed. The advantage of this technique is that first-order symbolic execution is sufficient for soft contract verification. The reason for this is that the program is *executed* until a first-order symbolic formula, solvable by off-the-shelf SAT solvers, can be obtained.

Our proposed analysis expands this work by making it *compositional*. Prior work executes the *entire program* symbolically while representing some values (such as contracts and closures) abstractly. This results in an analysis that shares a path condition and symbolic store *forwards* across function boundaries. Instead, our technique summarises the function by its paths and potential contract violations while selectively propagating some paths across function boundaries *backwards*, and propagating all contract violations backwards along the call chain.

Compositional Analysis Cousot and Cousot [8] propose a *modular compositional analysis*, where the analysis of separate components can be easily combined to obtain the analysis of an entire program. For higher-order dynamic programming languages, as is the subject of this paper, such decomposition is generally difficult due to the absence of a call graph. For this reason, we split the analysis into two phases: a collection phase which results in a call graph, and a propagation phase in which function summaries propagate through the call graph.

Separation logic [18] also enables modular verification of heap-related and other program properties. Our analysis does not specifically target heap-related properties, but rather works well on contracts about the input and output of the function with limited amount of state.

Furthermore, properties in separation logic are specified using a fixed logic language, whose semantics are predefined. Our analysis is able to reason about contracts specified in the same programming language as the monitored code. In fact, except for propagation heuristics, our analysis does not differentiate between the execution of a contract check and the execution of the monitored code.

Finally, our analysis provides a *soft* verifier, meaning that the contracts that cannot be verified by our analysis are left as residual checks to be checked at run time. In separation logic, most verifiers aim for a full analysis of the program. However, for unverified modules run-time systems also exist to ensure the same properties as the verified ones [19]. Unfortunately, because separation logic often expresses properties about *ownership*, such run-time systems introduce an additional burden (i.e., support for machine-enforced capabilities). We do not aim for our contract language to support expressing such properties.

VIII. Conclusion

We presented a novel approach to *soft contract verification* using a modular and compositional analysis called SCVMODFSUMMARY. The analysis consists of two phases: a collection phase and a propagation phase. The collection phase results in a call graph that can be topologically sorted to obtain an optimal path condition and blame propagation order. Our analysis is compositional in the sense that all functions are summarised to their paths and potential contract violations. These summaries can then be re-used for the analysis of other functions and composed together during the propagation phase. We evaluated our approach on a number of benchmark programs and demonstrated that our approach improves both the performance (in terms of execution time) and precision compared to the state-of-the-art.

References

[1] B. Meyer, "Design by contract: The Eiffel method," in *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1998, p. 446.

[2] R. B. Findler and M. Felleisen, "Contracts for higher-order functions," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), 2002*, M. Wand and S. L. P. Jones, Eds. ACM, 2002, pp. 48–59.

[3] C. Dimoulas, M. S. New, R. B. Findler, and M. Felleisen, "Oh lord, please don't let contracts be misunderstood (functional pearl)," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*, J. Garrigue, G. Keller, and E. Sumii, Eds. ACM, 2016, pp. 117–131.

[4] D. Feltey, B. Greenman, C. Scholliers, R. B. Findler, and V. St-Amour, "Collapsible contracts: fixing a pathology of gradual typing," vol. 2, no. OOPSLA, 2018, pp. 133:1–133:27.

[5] P. C. Nguyen, S. Tobin-Hochstadt, and D. Van Horn, "Soft contract verification," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14)*, J. Jeuring and M. M. T. Chakravarty, Eds. ACM, 2014, pp. 139–152.

[6] P. C. Nguyen, T. Gilray, S. Tobin-Hochstadt, and D. V. Horn, "Soft contract verification for higher-order stateful programs," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 51:1–51:30, 2018.

[7] D. V. Horn and M. Might, "Abstracting abstract machines," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, September 27-29, 2010*, P. Hudak and S. Weirich, Eds. ACM, 2010, pp. 51–62.

[8] P. Cousot and R. Cousot, "Modular static program analysis," in *Compiler Construction, 11th International Conference (CC 2002)*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Springer, 2002, pp. 159–178.

[9] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[10] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Effect-driven flow analysis," in *Verification, Model Checking, and Abstract Interpretation (VMCAI 2019)*, ser. Lecture Notes in Computer Science, C. Enea and R. Piskac, Eds., vol. 11388. Springer, 2019, pp. 247–274.

[11] M. Felleisen and D. P. Friedman, "A calculus for assignments in higher-order languages," in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1987, pp. 314–325.

[12] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 1977, pp. 238–252.

[13] T. Gilray, M. D. Adams, and M. Might, "Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*, J. Garrigue, G. Keller, and E. Sumii, Eds. ACM, 2016, pp. 407–420.

[14] P. C. Nguyen, S. Tobin-Hochstadt, and D. Van Horn, "Higher order symbolic execution for contract verification and refutation," *J. Funct. Program.*, vol. 27, p. e3, 2017.

[15] E. S. Andreasen, A. Møller, and B. B. Nielsen, "Systematic approaches for increasing soundness and precision of static analyzers," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP@PLDI 2017)*, K. Ali and C. Cifuentes, Eds. ACM, 2017, pp. 31–36.

[16] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "A general method for rendering static analyses for diverse concurrency models modular," *Journal of Systems and Software*, vol. 147, pp. 17–45, jan 2019.

[17] S. Tobin-Hochstadt and D. Van Horn, "Higher-order symbolic execution via contracts," in *Proceedings of the ACM International conference on Object oriented programming systems languages and applications (OOPSLA '12)*, ser. OOPSLA '12. ACM, 2012, pp. 537–554.

[18] P. O'Hearn, "Separation logic," *Commun. ACM*, vol. 62, no. 2, pp. 86–95, jan 2019.

[19] T. V. Strydonck, F. Piessens, and D. Devriese, "Linear capabilities for fully abstract compilation of separation-logic-verified code," *J. Funct. Program.*, vol. 31, p. e6, 2021.