Compositional Symbolic Execution for Correctness and Incorrectness Reasoning

Andreas Lööw
Imperial College London, UK

Daniele Nantes-Sobrinho
Imperial College London, UK

Sacha-Élie Ayoun
Imperial College London, UK

Caroline Cronjäger
Ruhr-Universität Bochum, Germany

Petar Maksimović
Imperial College London, UK
Runtime Verification Inc., Chicago, IL, USA

Philippa Gardner
Imperial College London, UK

Abstract

The introduction of separation logic has led to the development of symbolic execution techniques and tools that are (functionally) compositional with function specifications that can be used in broader calling contexts. Many of the compositional symbolic execution tools developed in academia and industry have been grounded on a formal foundation, but either the function specifications are not validated with respect to the underlying separation logic of the theory, or there is a large gulf between the theory and the implementation of the tool.

We introduce a formal compositional symbolic execution engine which creates and uses function specifications from an underlying separation logic and provides a sound theoretical foundation for, and indeed was partially inspired by, the Gillian symbolic execution platform. This is achieved by providing an axiomatic interface which describes the properties of the consume and produce operations used in the engine to update compositionally the symbolic state, for example when calling function specifications. This consume-produce technique is used by VeriFast, Viper, and Gillian, but has not been previously characterised independently of the tool. As part of our result, we give consume and produce operations inspired by the Gillian implementation that satisfy the properties described by our axiomatic interface. A surprising property is that our engine semantics provides a common foundation for both correctness and incorrectness reasoning, with the difference in the underlying engine only amounting to the choice to use satisfiability or validity. We use this property to extend the Gillian platform, which previously only supported correctness reasoning, with incorrectness reasoning and automatic true bug-finding using incorrectness bi-abduction. We evaluate our new Gillian platform by using the Gillian instantiation to C. This instantiation is the first tool grounded on a common formal compositional symbolic execution engine to support both correctness and incorrectness reasoning.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Program analysis; Theory of computation → Separation logic; Theory of computation → Automated reasoning

Keywords and phrases separation logic, incorrectness logic, symbolic execution, bi-abduction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.25

Related Version *Extended Version*: https://doi.org/10.48550/arXiv.2407.10838 [18]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: https://doi.org/10.4230/DARTS.10.2.13

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 25; pp. 25:1–25:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding This work was supported by the EPSRC Fellowship "VetSpec: Verified Trustworthy Software Specification" (EP/R034567/1).

Acknowledgements We would like to thank Nat Karmios for help with preparing the artefact for this paper. We would also like to thank the anonymous reviewers for their comments.

1 Introduction

One of the main challenges that modern program analysis tools based on static symbolic execution [1] must face is *scalability*, that is, the ability to tractably analyse large, dynamically changing codebases. Such scalability requires symbolic techniques and tools that are *functionally compositional* (or simply compositional) where the analysis works on functions in isolation, at any point in the codebase, and then records the results in simple function specifications that can be used in broader calling contexts. However, the traditional symbolic execution tools and frameworks based on first-order logic, such as CBMC [16] and Rosette [32], can only be compositional for functions that manipulate the variable store, but not for functions that manipulate the heap, limiting their usability.

A key insight is that, to obtain compositionality, the analysis should work with function specifications that are *local*, in that they should describe the function behaviour only on the *partial* states or resources that the function accesses or manipulates, and a mechanism for using such specifications when the function is called by code working on a larger state. This insight was first introduced in separation logic (SL) [24, 29], a modern over-approximating (OX) program logic for compositional verification of correctness properties, which features local function specifications that can be called on larger state with the help of the *frame rule*. Recently, these ideas have been adapted to under-approximate (UX) reasoning in the context of incorrectness separation logic (ISL) [28] for compositional true bug-finding.

Ideas from SL and ISL have led to the development of efficient compositional symbolic execution tools in academia and industry, such as the tool VeriFast [14] and the multi-language platforms Viper [23] and Gillian [21] for semi-automatic OX verification based on SL, and Meta's multi-language platform Infer-Pulse [17] for automatic UX true bug-finding based on ISL. However, there are issues with the associated formalisms of the tools: either the function specifications created and used by the tool are not validated with respect to the underlying separation logic; or there is a large gulf between the formalism and the implementation of the tool. VeriFast, Viper, and Gillian, on the one hand, all come with a sound compositional symbolic operational semantics that closely model the tools. These tools handle the creation and use of function specifications (and the folding and unfolding of predicates) using two operations, called *consume* and *produce*, which, respectively, removes from and adds to a given symbolic state the symbolic state corresponding to a given assertion. The formalisms accompanying the tools, however, do not properly connect their function specifications to the underlying separation logics. Thus, function specifications created by the tools cannot soundly be used by other tools, and vice versa. On the other hand, the formalism of Infer-Pulse describes compositional symbolic execution as proof search in ISL, and similarly with its SL-predecessor Infer [4], thereby making the connection to its separation logic direct. However, the gap between the formalism and the tool is considerable.

In this paper, inspired by the Gillian platform [11, 21], we formally define a compositional symbolic execution (CSE) engine that provides a sound theoretical foundation for building compositional OX and UX analysis tools. Our engine is described by a compositional symbolic operational semantics using the consume and produce operations to interface with function

specifications valid in either SL or ISL. A surprising property of our semantics is that it is simple to switch between OX and UX reasoning. In more detail, our CSE engine features the following theoretical contributions:

1. *specification interoperability*, in the sense that it can both create and use function specifications validated in an underlying separation logic, allowing for a mix-and-match of specifications validated in various ways from various sources;
2. *an axiomatic approach to compositionality*, in that we provide an axiomatic description of properties for the consume and produce operations, which have not been previously characterised axiomatically;
3. *a general soundness result*, which states that, assuming the axiomatic properties of the consume and produce operations and the validity of function specifications that are used with respect to the underlying separation logic, the CSE engine is sound and creates function specifications that are valid with respect to the said logic;
4. *a unified semantics*, which captures the essence of *both* the OX reasoning of VeriFast, Viper, and Gillian, and the UX reasoning in Infer-Pulse, with the difference amounting *only* to the choice of using satisfiability or validity, and different axiomatic properties of the consume and produce operations.

We instantiate our general soundness result by giving example implementations of the consume and produce operations, inspired by those found in Gillian, which we prove satisfy the properties laid out by the axiomatic interface. For clarity of presentation, although both our CSE engine and our consume and produce operations are inspired by Gillian, we have opted to work with a fixed, linear memory model and a simple while language instead of the parametric memory model approach of Gillian and its goto-based intermediate language GIL. The move from a fixed to a parametric memory model is straightforward and planned future work.

In addition, this paper brings the following practical contributions:

1. a demonstrator Haskell implementation of our CSE engine with example implementations of the consume and produce operations;
2. an extension of the Gillian platform with automatic compositional UX true bug-finding using UX bi-abduction in the style of Infer-Pulse, making Gillian the first unified tool for OX and UX compositional reasoning about real-world code.

The Gillian platform already supported whole-program symbolic testing as found in, for example, CBMC, and semi-automatic OX compositional verification underpinned by SL as in, for example, VeriFast. Because our CSE engine has pinpointed the small differences required for the switch between OX and UX reasoning, we were able to simply extend Gillian with automatic compositional UX true-bug finding without affecting its other analyses. Interestingly, UX true bug-finding has not been implemented in a consume-produce engine before. We demonstrate the additional UX reasoning by extending the CSE engine with UX bi-abductive reasoning [5, 6, 28, 17], an automatic technique which has enabled compositional reasoning to scale to industry-grade codebases, and which works by generating function specifications from their implementations by incrementally constructing the calling context. We implement this technique following the approach pioneered by OX tool JaVerT 2.0 [10], where missing-resource errors are used to generate fixes that drive the specification construction. We evaluate this extension of Gillian using its Gillian-C instantiation, on a real-world Collections-C data-structure library [25], obtaining promising initial results and performance.

An extended version of this paper is available on arXiv [18], which includes additional definitions and proofs of the theorems discussed in this paper.

2 Overview: Compositional Symbolic Execution

We give an overview of our CSE engine, together with example analyses that we show can be hosted on top of this engine. Our CSE engine consists of three engines built on top of each other, labelled by different reasoning modes, OX and UX, that are appropriate for different types of analyses. In short, the reasoning modes can be characterised as follows:

Mode	Guarantee	Consequence rule	Analysis
OX	Full path coverage	Forward logical consequence	Full verification in SL
UX	Path reachability	Backward logical consequence	True bug-finding in ISL

The *core engine* (§4) is a simple symbolic execution engine for our demonstrator programming language (§3). It does not use assertions to update symbolic state and is hence compatible with both OX and UX reasoning. It is sufficient to capture whole-program symbolic testing found in, e.g., CMBC and Gillian. The *compositional engine* (§5, §6) is built on top of this core engine. It can switch between either the OX or UX mode of reasoning, providing support for the use of SL and ISL function specifications by extending the core engine with consume and produce operations for updating the symbolic state. In OX mode, it captures the full verification found in e.g. VeriFast and Gillian, soundly underpinned by SL. For the first time, in UX mode it also captures ISL analysis, not previously found in a symbolic execution tool. We demonstrate this by building the UX *bi-abductive engine* (§7) on top of the UX compositional engine to automatically fix missing-resource errors (e.g., a missing heap cell) using the UX bi-abductive technique from Infer-Pulse, to capture automatic true bug-finding underpinned by ISL.

2.1 Core Engine

The core symbolic execution engine provides a foundation on top of which the other components are built. It is essentially a standard symbolic execution engine that is slightly adapted to handle both usual language errors and the missing-resource errors, which can occur now that we are working with partial state.

Our engine operates over partial symbolic states $\hat{\sigma} = (\hat{s}, \hat{h}, \hat{\pi})$ comprising: a symbolic variable store $\hat{s}$, holding symbolic values for the program variables; a partial symbolic heap $\hat{h}$, representing the memory on which programs operate; and a symbolic path condition $\hat{\pi}$, holding constraints accumulated during symbolic execution. We work with a simple demonstrator programming language (cf. §3) and linear heaps: that is, partial-finite maps from natural numbers to values. The core engine is both OX- and UX-sound, also referred to as exact (EX) [20], as established by Thm. 1.

Example. In Fig. 1 (left), we define a simple function `f`. In Fig. 1 (middle), we illustrate its symbolic execution, which starts from $\hat{\sigma} = (\{\mathsf{c} \mapsto \hat{c}, \mathsf{x} \mapsto \hat{x}, \mathsf{r} \mapsto \texttt{null}\}, \emptyset, \mathsf{true})$, with the function parameters (`c` and `x`) initialised with some symbolic variables ($\hat{c}$ and $\hat{x}$), the local function variables (`r`) initialised to `null`, the heap set to empty and the path condition set to `true`. Next, executing the if-statement with condition $\mathsf{c} \geq 42$ yields three branches: one in which `c` is not a natural number, in which the execution fails with an evaluation error; one in which $\mathsf{c} \geq 42$, in which the execution continues; and one in which $\mathsf{c} < 42$, in which the program throws a user-defined error. Next, executing the lookup $\mathsf{r} := [\mathsf{x}]$ results in two more branches: one in which `x` is not a heap address (natural number), yielding a type error and one in which `x` is a heap address. In that branch, the lookup fails with a missing-resource error as the heap is empty.

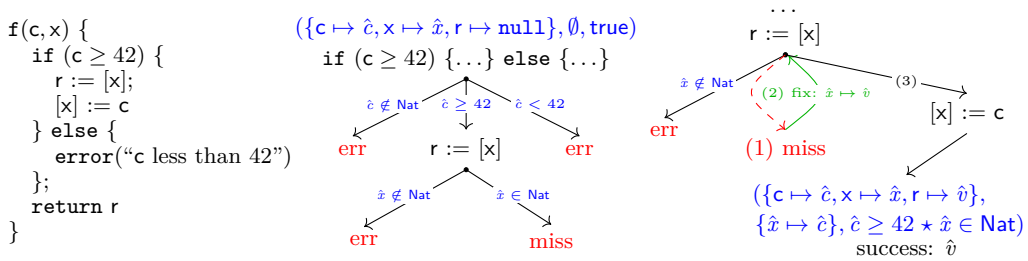


Figure 1 Definition and symbolic execution of function `f`.

Analysis: EX Whole-program Symbolic Testing. The core engine can be used to perform whole-program symbolic testing in the style of CBMC [16] and Gillian [11], in which the user creates symbolic variables, imposes some initial constraints on them, runs the symbolic execution to completion, and asserts that some final constraints hold.

2.2 Compositional Engine

Our compositional engine extends the core engine to support calling, in its respective OX and UX mode, SL and ISL function specifications, denoted $\{\vec{x} = \vec{x} \star P\}\ \mathsf{f}(\vec{x})\ \{ok : Q_{ok}\}\ \{err : Q_{err}\}$ and $[\vec{x} = \vec{x} \star P]\ \mathsf{f}(\vec{x})\ [ok : Q_{ok}]\ [err : Q_{err}]$,[1] respectively, where P is a pre-condition and Q_{ok} and Q_{err} are success and error post-conditions. A SL specification gives an OX description of the function behaviour whereas an ISL specification gives a UX description:

(SL) All terminating executions of the function `f` starting in a state satisfying P either end successfully in a state that satisfies Q_{ok} or fault in a state that satisfies Q_{err}.

(ISL) Any state satisfying either the success Q_{ok} or error post-conditions Q_{err} is reachable from some state satisfying the pre-condition P by executing the function `f`.

The engine also adds support, in both OX and UX mode, for folding and unfolding of user-defined predicates, describing inductive data-structures such as linked lists.

In both cases, the call to function specifications and the folding and unfolding of predicates are implemented following the consume-produce engine style, underpinned by consume and produce operations, which, in essence, remove (consume) and add (produce) the symbolic state corresponding to a given assertion from and to the current symbolic state. For example, in Fig. 2, the symbolic execution is in a symbolic state $\hat{\sigma}$ and calls a function $\mathsf{f}(\vec{x})$ by its specification in ISL mode. The (successful) function call is implemented by first consuming the symbolic state $\hat{\sigma}_P$ corresponding to the pre-condition P, leaving the symbolic frame $\hat{\sigma}_f$, and then producing into $\hat{\sigma}_f$ the symbolic state $\hat{\sigma}_{Q_{ok}}$ corresponding to the post-condition Q_{ok}.

Our approach is novel in two ways: (1) we provide an axiomatic interface that captures the sufficient properties of the consume and produce operations for the engine to be sound; and (2) we provide example implementations (in the same style as the rest of the engine, that is, using inference rules) for the consume and produce operations that we prove satisfy the axiomatic interface. Moreover, our consume and produce operations switch their behaviour between the mode of reasoning (OX/UX), as described next.

[1] UX quadruples can be split into two triples, but OX quadruples cannot. To unify our presentation, we consider both types of specifications in quadruple form.

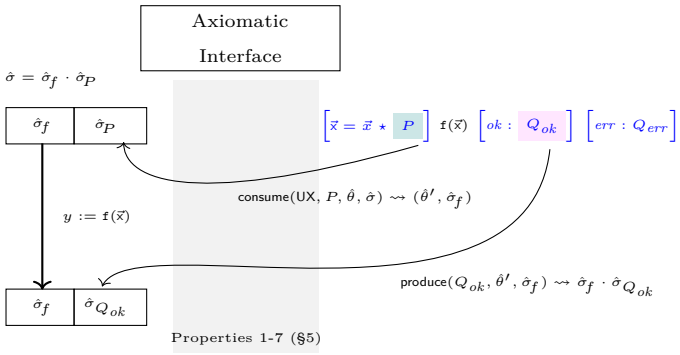


Figure 2 UX function-call rule: successful case.

Axiomatic Interface. We have identified sufficient properties for the consume and produce operations to be OX or UX sound (cf. Thm. 3). Here we will describe a general idea of the consume operation, the more complex of the two operations, the signature of which is:

$$\mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) \mid \mathit{abort}(\hat{v})$$

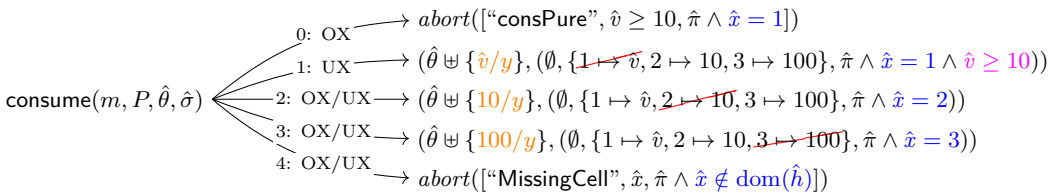
The consume operation takes a mode m (OX or UX), an assertion P, a substitution $\hat{\theta}$, and a symbolic state $\hat{\sigma}$, where the substitution $\hat{\theta}$ links known logical variables in P to symbolic values in $\hat{\sigma}$. The operation finds which part of $\hat{\sigma}$ could match P, resulting in potentially multiple successful or unsuccessful matches, and then, per match, returns either the pair $(\hat{\theta}', \hat{\sigma}_f)$, which comprises a substitution $\hat{\theta}'$ and a resulting symbolic state $\hat{\sigma}_f$, or it aborts with error information $\hat{v}$. Some properties the interface of consume mandates are the following:

1. In successful consumption, there exists a symbolic state $\hat{\sigma}_P$ such that $\hat{\sigma} = \hat{\sigma}_f \cdot \hat{\sigma}_P$ (where $\cdot$ denotes state composition, which composes the corresponding components of the two states together) and that every concrete state described by $\hat{\sigma}_P$ satisfies P. This tells us that the matched part of $\hat{\sigma}$ does correspond to P, that the effect of consume is its removal from $\hat{\sigma}$, and that consumption can be viewed as the frame inference problem [3], with the resulting state $\hat{\sigma}_f$ constituting the frame;
2. In OX mode, consume does not drop paths; in UX mode, it does not drop information.

The interface allows, e.g., tool developers to design an OX-consume that (soundly) drops certain information deemed unneeded or a UX-consume that (soundly) drops paths according to a tool-specific criteria (e.g., as in UX bi-abduction in Infer-Pulse).

Example Implementation: Consume. We provide example implementations for the consume and produce operations (§6) that explore the similarities between OX and UX reasoning, and allow us to maintain unified implementations across both reasoning modes. Our consume operation has a mode switch m, allowing for OX- or UX-specific behaviour, which we use to control the *only difference* in our implementation between the two modes: the consumption of pure (non-spatial) information (cf. Fig. 7, left). For soundness, our implementation of the consume operation has to be compatible with the SL and ISL guarantees: in OX mode, consume requires full path coverage, and in UX, it requires path reachability.

We illustrate our consume implementation by example. Consider the symbolic state $\hat{\sigma} \triangleq (\emptyset, \hat{h}, \hat{\pi})$, where $\hat{h} \triangleq \{1 \mapsto \hat{v}, 2 \mapsto 10, 3 \mapsto 100\}$ and $\hat{\pi} \triangleq \hat{x} > 0 \land \hat{v} > 5$, and let us consume the assertion $P \triangleq x \mapsto y \star y \geq 10$ from $\hat{\sigma}$ knowing that $\hat{\theta} = \{\hat{x}/x\}$, meaning that the logical variable x of P is mapped to the symbolic variable $\hat{x}$ of $\hat{\sigma}$. This consumption is presented in the diagram below:



- the arrows are labelled with the mode m of operation of consume, being either only OX, or only UX, or OX/UX when the consumption has the same behaviour in both modes;
- both our OX and UX consumption branch on all possible matches: in this case, the cell assertion $x \mapsto y$ can be matched to any of the three cells in the heap (branches 0–3), but it could also refer to a cell that is outside of the heap (branch 4);
- when branching occurs, then the branching condition is added to the path condition of the resulting state (the constraints in blue), ensuring information is not dropped;
- the heap cell corresponding to $x \mapsto y$ is removed when matched successfully (branches 1, 2, 3), and in those cases we learn the value corresponding to y (substitution extension in orange where $\uplus$ denotes disjoint union);
- for the heap cell $\{1 \mapsto \hat{v}\}$, our OX consumption (branch 0) must abort since the $\hat{\pi}$ does not imply $y \geq 10$ when $y = \hat{v}$, whereas UX consumption (branch 1) can proceed by restricting the path condition (constraint in magenta), since dropping paths is sound in UX; this allows our UX consumption to successfully consume more assertions; OX consumption cannot do the same since that would render e.g. the function-call rule, which is implemented in terms of consume, unsound in OX mode;
- our UX consumption could alternatively drop the missing-cell abort outcome (branch 4), however, some analyses, such as bi-abduction, have use for this error information so we do not drop it.

Analysis: Verification. We use our compositional engine to provide semi-automatic OX verification: that is, given a function implementation and an OX function specification, the engine checks if the implementation satisfies the specification. This analysis is semi-automatic in that the user may have to provide loop invariants as well as ghost commands for, e.g., predicate manipulation and lemma application. It is implemented in the standard way for consume-produce tools.

2.3 Bi-abductive Engine

Bi-abduction is a technique that enables automatic compositional analysis by allowing incremental discovery of resources needed to execute a given piece of code. It was introduced in the OX verification setting [5, 6], later forming the basis of the bug-finding tool Infer [4], and was recently ported to the UX setting of true bug-finding, underpinning Infer-Pulse [17].

Our UX bi-abduction advances the state of the art in two ways. Firstly, UX bi-abduction has thus far been intertwined with the proof search of the symbolic execution tool it has formulated for [28, 17]. Inspired by an approach developed in the OX tool JaVerT 2.0 [10], we add UX bi-abduction as a layer on top of CSE by generating fixes from missing-resource errors. This covers both missing-resource errors from the execution of the commands of the language (e.g., in heap lookup, the looked-up cell might not be in the heap) as well as invocations of consume (e.g., if the resource required by a function pre-condition is not in the heap). In more detail, when a missing-resource error occurs, a fix is generated and applied to

the current symbolic state, allowing the execution to continue. Secondly, our UX bi-abduction is able to reason about predicates, allowing us to synthesise and soundly use a broader range of function specifications from other formalisms and tools, in particular specifications that capture unbounded behaviour rather than bounded or single-path behaviour.

Analysis: Specification Synthesis and True Bug-finding. We use bi-abduction to power automatic synthesis of UX function specifications, obtaining one specification per each constructed execution path. Such function specification synthesis forms the back-end of Pulse-style true bug-finding, where specifications describing erroneous executions, after appropriate filtering, can be reported as bugs. Given the guarantees of UX reasoning, any bug (represented by a synthesised erroneous function specification) found during this process will be a true bug.

Example: Specification Synthesis. We illustrate how bi-abduction can be used for the synthesis of UX function specifications, using again the simple function $\texttt{f(c,x)}$ from Fig. 1 (left). The first and the third branches of Fig. 1 (middle) yield the following specifications:

$$\left[\mathsf{c} = c \star \mathsf{x} = x\right]\ \texttt{f(c,x)}\ \left[err : \mathsf{err} = [\text{“ExprEval”}, \text{“c} \geq 42\text{”}] \star c \notin \mathsf{Nat}\right]$$
$$\left[\mathsf{c} = c \star \mathsf{x} = x\right]\ \texttt{f(c,x)}\ \left[err : \mathsf{err} = [\text{“Error”}, \text{“c less than 42”}] \star c < 42\right]$$

noting that information about local variables is discarded, the error value is returned in the dedicated program variable err, and symbolic variables are replaced with logical variables.

Using bi-abduction, the second branch of Fig. 1 (middle) now becomes Fig. 1 (right). The second branch of Fig. 1 (middle) has one branch in which x is not a heap address (natural number), yielding a type error and the following specification

$$\left[\mathsf{c} = c \star \mathsf{x} = x\right]\ \texttt{f(c,x)}\ \left[err : \mathsf{err} = [\text{“Type”}, \text{“x”}, x, \text{“Nat”}] \star c \geq 42 \star x \notin \mathsf{Nat}\right]$$

and one branch in which x is a heap address. In that branch, the lookup fails with a missing-resource error as the heap is empty, but in bi-abductive execution, that is, Fig. 1 (right), instead of failing we first generate the fix $\hat{x} \mapsto \hat{v}$, where $\hat{v}$ is a fresh symbolic variable, and then add it to the heap and re-execute the lookup, which now succeeds. The rest of the function is executed without branching or errors, the function terminates and returns the value of r, which is $\hat{v}$. This branch results in the following specification:

$$\left[\mathsf{c} = c \star \mathsf{x} = x \star x \mapsto v\right]\ \texttt{f(c,x)}\ \left[ok : x \mapsto c \star c \geq 42 \star \mathsf{ret} = v\right]$$

which illustrates an essential principle of bi-abduction, which is to add the fixes applied during execution (also known as *anti-frame*, highlighted in red) to the specification pre-condition.

Example: Specification Synthesis with Predicates. To exemplify how predicates can be useful during specification synthesis, consider the following variant of the standard singly-linked list predicate: $\mathsf{list}(x; xs, vs)$, where x denotes the starting address of the list, and xs and vs denote node addresses and node values, respectively.[2] Both addresses and values are exposed in the predicate to ensure that no information is lost when the predicate is folded, making it suitable for UX reasoning. The predicate is defined as follows:

$$\begin{aligned}\mathsf{list}(x; xs, vs) \triangleq\ & (x = \texttt{null} \star xs = [] \star vs = []) \lor \\ & (\exists v, x', xs', vs'.\, x \mapsto v, x' \star \mathsf{list}(x'; xs', vs') \star xs = x : xs' \star vs = v : vs')\end{aligned}$$

[2] We use the semicolon notation for predicates to be consistent with the main text, where the notation is used for automation – for the purpose of this section, these semicolons can be read as commas.

Further, consider the predicate $\mathsf{listHead}(x; xs)$, which tells us that x is the head of the list xs if xs is not empty and `null` otherwise, defined as

$$\mathsf{listHead}(x; xs) \triangleq (xs = [] \star x = \texttt{null}) \vee (\exists xs'.\, xs = x : xs'),$$

and the following specifications of two list-manipulating functions (e.g., proven using pen-and-paper), which capture the exact behaviour of inserting a value in the front of a list (`LInsert`) and swapping of the first two values in a list (`LSwapFirstTwo`):

$[\mathsf{x} = x \star \mathsf{v} = v \star \mathsf{list}(x; xs, vs)]$ `LInsert`(x, v) $[ok : \mathsf{list}(\mathsf{ret}; \mathsf{ret}{:}xs, v{:}vs) \star \mathsf{listHead}(x; xs)]$

$[\mathsf{x} = x \star \mathsf{list}(x; xs, vs)]$ `LSwapFirstTwo`(x) $[err : \mathsf{list}(x; xs, vs) \star |vs| < 2 \star \mathsf{err} = \text{“List too short!”}]$

Using these specifications, we can bi-abduce the following UX true-bug specification of client code calling these functions, where the discovered anti-frame is again highlighted in red, but this time contains a predicate:

$[\mathsf{x} = x \star \mathsf{list}(x; xs, vs)]$
$\mathsf{x} :=$ `LInsert`$(\mathsf{x}, 42); \mathsf{y} :=$ `LSwapFirstTwo`(x)
$[err : \exists x'.\ \mathsf{list}(x'; x'{:}xs, 42{:}vs) \star \mathsf{listHead}(x; xs) \star |42{:}vs| < 2 \star \mathsf{err} = \text{“List too short!”}]$

3 Programming Language

We present a simple imperative heap language with function calls on which our analysis engine operates. The language is standard, except that, in line with previous work on compositional reasoning and incorrectness [8, 9, 10, 12, 28], we track freed cells in the heap, and separate language errors and missing-resource errors to preserve the compositionality of the semantics. We sometimes refer to the definitions of this section as *concrete* to differentiate them from the *symbolic* definitions used in the symbolic engine introduced in subsequent sections.

Syntax. The values are given by: $v \in \mathsf{Val} ::= n \in \mathsf{Nat} \mid b \in \mathsf{Bool} \mid s \in \mathsf{Str} \mid \texttt{null} \mid [\vec{v}]$, where $\vec{v}$ denotes a vector of values. The types are given by: $\tau \in \mathsf{Type} ::= \mathsf{Val} \mid \mathsf{Nat} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{List}$. The types are used to define the semantics of the language; the language itself is dynamically typed. The expressions, $\mathsf{E} \in \mathsf{PExp}$, comprise the values, program variables $\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots \in \mathsf{PVar}$, and expressions formed using the standard operators for numerical and Boolean expressions. The commands are given by the grammar:

$$\begin{aligned}\mathsf{C} \in \mathsf{Cmd} ::= {} & \texttt{skip} \mid \mathsf{x} := \mathsf{E} \mid \mathsf{x} := \texttt{nondet} \mid \texttt{error}(\mathsf{E}) \mid \mathsf{x} := [\mathsf{E}] \mid [\mathsf{E}] := \mathsf{E} \mid \mathsf{x} := \texttt{new} \mid \\ & \texttt{free}(\mathsf{E}) \mid \texttt{if}\ (\mathsf{E})\ \mathsf{C}\ \texttt{else}\ \mathsf{C} \mid \mathsf{C}; \mathsf{C} \mid \mathsf{y} := f(\vec{\mathsf{E}})\end{aligned}$$

comprising the variable assignment, variable assignment of a non-deterministically chosen natural number, user-thrown error, heap lookup, heap mutation, allocation, deallocation, command sequencing, conditional control-flow and function call. Our results extend to other control-flow commands, e.g. loops, since these can be implemented using conditionals and recursive functions. The sets of program variables for expressions $\mathsf{pv}(\mathsf{E})$ and commands $\mathsf{pv}(\mathsf{C})$ are standard.

Functions and Function Implementation Contexts. A function implementation, denoted $f(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \texttt{return}\ \mathsf{E}\,\}$, comprises: an identifier, $f \in \mathsf{Fid} \subseteq \mathsf{Str}$; the parameters, given by a list of distinct program variables $\vec{\mathsf{x}}$; a body, $\mathsf{C} \in \mathsf{Cmd}$; and a return expression, $\mathsf{E} \in \mathsf{PExp}$, with $\mathsf{pv}(\mathsf{E}) \subseteq \{\vec{\mathsf{x}}\} \cup \mathsf{pv}(\mathsf{C})$. A function implementation context, γ, maps function identifiers to their implementations: $\gamma : \mathsf{Fid} \rightharpoonup_{fin} \mathsf{PVar\ List} \times \mathsf{Cmd} \times \mathsf{PExp}$, where $\rightharpoonup_{fin}$ denotes that the function is a finite partial function. We often write $f(\vec{\mathsf{x}})\{\mathsf{C}; \texttt{return}\ \mathsf{E}\} \in \gamma$ for $\gamma(f) = (\vec{\mathsf{x}}, \mathsf{C}, \mathsf{E})$.

Stores, Heaps and States. A variable store, $s : \mathsf{PVar} \rightharpoonup_{fin} \mathsf{Val}$, is a function from program variables to values. A *partial* heap, $h : \mathsf{Nat} \rightharpoonup_{fin} (\mathsf{Val} \uplus \{\varnothing\})$, is a function from natural numbers to values extended with a dedicated symbol $\varnothing \notin \mathsf{Val}$ recording that a heap cell has been freed. Two heaps are disjoint, denoted $h_1 \sharp h_2$, when their domains are disjoint. Heap composition, denoted $h_1 \uplus h_2$, is given by the disjoint union of partial functions which is only defined when the domains are disjoint. A *partial* program state, $\sigma = (s, h)$, is a pair comprising a store and a heap. State composition, denoted $\sigma_1 \cdot \sigma_2$, is given by $(s_1, h_1) \cdot (s_2, h_2) \triangleq (s_1 \cup s_2, h_1 \uplus h_2)$ for $\sigma_1 = (s_1, h_1)$ and $\sigma_2 = (s_2, h_2)$, which is only defined when variable stores are equal on their intersection and the heap composition is defined.

Operational Semantics. We use a standard expression evaluation function, $[\![\mathsf{E}]\!]_s$, which evaluates an expression E with respect to a store s, assuming that expressions do not affect the heap. It results in either a value or a dedicated symbol $\lightning \notin \mathsf{Val}$, denoting, an evaluation error, such as a variable not being in the store or a mathematical error. The operational semantics of commands is a big-step semantics using judgements of the form $\sigma, \mathsf{C} \Downarrow_\gamma o : \sigma'$ which reads "the execution of command C in state σ and function implementation context γ results in a state σ' with outcome o", where $o ::= \mathit{ok} \mid \mathit{err} \mid \mathit{miss}$ denotes, respectively, a successful execution, a language error, and a missing-resource error due to the absence of a required cell in the partial heap. The separation of the missing-resource errors from the language errors is important for compositional reasoning, since the language satisfies both the standard OX and UX frame properties when the outcome is not missing. The semantics is standard and given in full in [18], along with the frame properties it satisfies.

4 Compositional Symbolic Execution: Core Engine

We present our CSE engine in two stages. In this section, we present the core CSE engine, given by a standard compositional symbolic operational semantics presented here to establish notation and introduce key concepts to the non-specialist reader: the definitions are similar to those for whole-program symbolic execution; the difference is with the use of partial state which has the effect that we have the distinction between language errors and missing-resource errors. In §5.3, we extend the core engine with our semantic rules for function calls and folding/unfolding predicates, using an axiomatic description of the consume and produce operations given in §5.2.

4.1 Symbolic States

Let SVar be a set of symbolic variables, disjoint from the set of program variables, PVar, and values, Val. Symbolic values are defined as follows:

$$\hat{v} \in \mathsf{SVal} ::= v \mid \hat{x} \mid \hat{v} + \hat{v} \mid \ldots \mid \hat{v} = \hat{v} \mid \neg\, \hat{v} \mid \hat{v} \wedge \hat{v} \mid \hat{v} \in \tau$$

A symbolic store, $\hat{s} : \mathsf{PVar} \rightharpoonup_{fin} \mathsf{SVal}$, is a function from program variables to symbolic values. A partial symbolic heap, $\hat{h} : \mathsf{SVal} \rightharpoonup_{fin} (\mathsf{SVal} \uplus \{\varnothing\})$, is a function from symbolic values to symbolic values extended with $\varnothing$. A path condition, $\hat{\pi} \in \mathsf{SVal}$, is a symbolic Boolean expression that captures constraints imposed on symbolic variables during execution. A (partial) symbolic state is a triple of the form $\hat{\sigma} = (\hat{s}, \hat{h}, \hat{\pi})$. Throughout the paper, we only work with well-formed states $\hat{\sigma}$, denoted $\mathcal{W}f(\hat{\sigma})$, the definition is uninformative (it ensures, e.g., that the addresses of the symbolic heap are interpreted as natural numbers), see [18]. We write $\hat{\sigma}.\mathsf{pc}$ and $\hat{\sigma}[\mathsf{pc} := \hat{\pi}']$ to denote, respectively, access and update the state path condition. We write $\mathtt{sv}(\mathsf{X})$ to denote the set of symbolic variables of a given construct X: e.g., $\mathtt{sv}(\hat{s})$ for symbolic stores, $\mathtt{sv}(\hat{h})$ for symbolic heaps, etc.

$$\textsc{Lookup}\quad \frac{[\![\mathsf{E}]\!]^{\hat{\pi}}_{\hat{s}} \Downarrow (\hat{v}, \hat{\pi}') \quad \hat{h}(\hat{v}_l) = \hat{v}_m \quad \hat{\pi}'' \triangleq (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \quad \mathtt{SAT}(\hat{\pi}'')}{(\hat{s}, \hat{h}, \hat{\pi}), \mathsf{x} := [\mathsf{E}] \Downarrow_\Gamma ok : (\hat{s}[\mathsf{x} \mapsto \hat{v}_m], \hat{h}, \hat{\pi}'')}$$

$$\textsc{Lookup-Err-Val}\quad \frac{[\![\mathsf{E}]\!]^{\hat{\pi}}_{\hat{s}} \Downarrow (\lightning, \hat{\pi}') \quad \hat{v}_{err} \triangleq [\text{“ExprEval”}, \mathsf{str}(\mathsf{E})]}{(\hat{s}, \hat{h}, \hat{\pi}), \mathsf{x} := [\mathsf{E}] \Downarrow_\Gamma err : (\hat{s}_{err}, \hat{h}, \hat{\pi}')}$$

$$\textsc{Lookup-Err-Missing}\quad \frac{[\![\mathsf{E}]\!]^{\hat{\pi}}_{\hat{s}} \Downarrow (\hat{v}, \hat{\pi}') \quad \mathtt{SAT}(\hat{\pi}' \wedge \hat{v} \in \mathsf{Nat} \wedge \hat{v} \notin \mathsf{dom}(\hat{h})) \quad \hat{v}_{err} \triangleq [\text{“MissingCell”}, \mathsf{str}(\mathsf{E}), \hat{v}]}{(\hat{s}, \hat{h}, \hat{\pi}), \mathsf{x} := [\mathsf{E}] \Downarrow_\Gamma miss : (\hat{s}_{err}, \hat{h}, \hat{\pi}' \wedge \hat{v} \in \mathsf{Nat} \wedge \hat{v} \notin \mathsf{dom}(\hat{h}))}$$

Figure 3 Excerpt of symbolic execution rules, where $\hat{s}_{err} = \hat{s}[\mathsf{err} \mapsto v_{err}]$.

Symbolic Interpretations. A symbolic interpretation, $\varepsilon : \mathsf{SVar} \rightharpoonup_{fin} \mathsf{Val}$ maps symbolic variables to concrete values, and is used to define the meaning of symbolic states and state the soundness results of the engine. We lift interpretations to symbolic values, $\varepsilon : \mathsf{SVal} \rightharpoonup_{fin} \mathsf{Val}$, with the property that it is undefined if the resulting concrete evaluation faults. Satisfiability of symbolic values is defined as usual, i.e., $\mathtt{SAT}(\hat{\pi}) \triangleq \exists \varepsilon.\ \varepsilon(\hat{\pi}) = \mathsf{true}$. We further lift symbolic interpretations to stores, heaps, and states, overloading the ε notation.

4.2 Core Engine

The symbolic expression evaluation relation, $[\![\mathsf{E}]\!]^{\hat{\pi}}_{\hat{s}} \Downarrow (\hat{w}, \hat{\pi}')$, evaluates a program expression E with respect to a symbolic store $\hat{s}$ and path condition $\hat{\pi}$. It results in either a symbolic value or an evaluation error, $\hat{w} \triangleq \hat{v} \mid \lightning$, and a satisfiable path condition $\pi' \Rightarrow \pi$, which may contain additional constraints arising from the evaluation (e.g., to prevent branching on division by zero). The core CSE semantics is described using the usual single-trace semantic judgement (below, left) which is used to state UX properties. It also induces the collecting semantic judgement (below, right), which is used to state OX properties.

$$\hat{\sigma}, \mathsf{C} \Downarrow_\gamma o : \hat{\sigma}' \qquad \hat{\sigma}, \mathsf{C} \Downarrow_\gamma \hat{\Sigma}' \iff \hat{\Sigma}' = \{(o, \hat{\sigma}') \mid \hat{\sigma}, \mathsf{C} \Downarrow_\gamma o : \hat{\sigma}'\}$$

We give the lookup rules for illustration in Fig. 3: for example, the rule Lookup branches over all possible addresses in the heap that can match the given address.

Our CSE semantics is both OX- and UX-sound, which we call exact: OX soundness captures that no paths are dropped by stating that the symbolic semantics includes all behaviour w.r.t. the concrete semantics; UX soundness captures that no information is dropped by stating that the symbolic semantics does not add behaviour w.r.t. the concrete semantics.

▶ **Theorem 1** (OX and UX soundness).

$$\textit{(OX)} \qquad \hat{\sigma}, \mathsf{C} \Downarrow_\gamma \hat{\Sigma}' \wedge \varepsilon(\hat{\sigma}), \mathsf{C} \Downarrow_\gamma o : \sigma' \implies \exists \hat{\sigma}', \varepsilon' \geq \varepsilon.\ (o, \hat{\sigma}') \in \hat{\Sigma}' \wedge \sigma' = \varepsilon'(\hat{\sigma}')$$

$$\textit{(UX)} \qquad \hat{\sigma}, \mathsf{C} \Downarrow_\gamma o : \hat{\sigma}' \wedge \varepsilon(\hat{\sigma}') = \sigma' \implies \varepsilon(\hat{\sigma}), \mathsf{C} \Downarrow_\gamma o : \sigma'$$

where $\varepsilon' \geq \varepsilon$ denotes that ε' extends ε, i.e., $\varepsilon'(\hat{x}) = \varepsilon(\hat{x})$ for all $\hat{x} \in \mathsf{dom}(\varepsilon)$.

5 Compositional Symbolic Execution: Full Engine

Our core CSE engine is limited in that it does not call function specifications written in a program logic, and it cannot fold and unfold user-defined predicates to verify, e.g., specifications of list algorithms. What is missing is a general description of how to update symbolic state using assertions from the function specifications and predicate definitions. In

VeriFast, Viper and Gillian, this symbolic-state update is given by their implementations of the consume and produce operations. We instead give an *axiomatic interface* for describing such symbolic-state update by providing a general characterisation of these consume and produce operations (§5.2). Using this interface, we are able to give general definitions of the function-call rule (§5.3) and the folding and unfolding of predicates that are independent of the underlying tool implementation. Assuming the appropriate properties stated in the axiomatic interface, we prove that the resulting CSE engine is either OX sound or UX sound. Moreover, because the axiomatic interface relates the behaviour of the consume and produce operations to the standard satisfaction relation of SL and ISL, our function-call rule is able to use any function specification proven correct with respect to the standard function specification validity of SL and ISL, including functions specification proven outside our engine. In the next section (§6), we demonstrate that the Gillian implementation of the consume and produce operations are correct with respect to our axiomatic interface.[3]

5.1 Assertions and Extended Symbolic States

We present assertions suitable for both SL and ISL reasoning, and also extend our symbolic states to account for predicates. It is helpful to make a clear distinction between the logical assertions and symbolic states: since we work with the linear heap, the gap between assertions and symbolic states is quite small; with more complex memory models and optimised symbolic representations of memory, the gap is larger and this distinction becomes essential.

Assertions. Let $x, y, z \in \mathsf{LVar}$ denote logical variables, distinct from program and symbolic variables. The set of logical expressions, $E \in \mathsf{LExp}$, extends program expressions to include logical variables. We work with the following set of assertions (other assertions are derivable):

$$\begin{aligned} \pi \in \mathsf{BAsrt} &\triangleq E \mid E \in \tau \mid \neg\pi \mid \cdots \mid \pi_1 \wedge \pi_2 \\ P \in \mathsf{Asrt} &\triangleq \pi \mid \mathsf{False} \mid P_1 \Rightarrow P_2 \mid P_1 \vee P_2 \mid \exists x.\, P \mid \\ &\quad \mathsf{emp} \mid E_1 \mapsto E_2 \mid E \mapsto \varnothing \mid P_1 \star P_2 \mid p(\vec{E_1}; \vec{E_2}) \end{aligned}$$

where $E, E_1, E_2 \in \mathsf{LExp}$, $x \in \mathsf{LVar}$, and $p \in \mathsf{Str}$. The assertions should by now be familiar from separation logic. They comprise the lift of the usual first-order Boolean assertions π, assertions built from the usual first-order connectives and quantifiers, and assertions well-known from separation logic: the empty assertion emp, the cell assertion $E_1 \mapsto E_2$ describing a heap cell at an address given by E_1 with value given by E_2, the less well-known assertion $E \mapsto \varnothing$ describing a heap cell at address E that has been freed, the separating conjunction $P_1 \star P_2$, and the predicate assertions $p(\vec{E_1}; \vec{E_2})$. The parameters of predicate assertions $p(\vec{E_1}; \vec{E_2})$ are separated into in-parameters $\vec{E_1}$ ("ins") and out-parameters $\vec{E_2}$ ("outs") for automation purposes, as we discuss in §6; this separation does not affect the logical meaning of the predicate assertions. We write $\mathtt{lv}(\mathsf{X})$ to denote free logical variables of a construct X: e.g., $\mathtt{lv}(E)$ for logical expressions, $\mathtt{lv}(P)$ for assertions, etc. We say that an assertion P is *simple* if it does not syntactically feature the separating conjunction; simple assertions are used in the definition of matching plans (§6.1).

Predicates. Predicate definitions are given by a set Preds containing elements of type $\mathsf{Str} \times \vec{\mathsf{LVar}} \times \vec{\mathsf{LVar}} \times \mathsf{Asrt}$, with the notation $p(\vec{x}_{\text{in}}; \vec{x}_{\text{out}})\ \{P\} \in \mathsf{Preds}$, where the string p denotes the predicate name, the lists of disjoint parameters $\vec{x}_{\text{in}}, \vec{x}_{\text{out}}$ denote the predicate

[3] To our best understanding, there is a large overlap between Gillian's consume and produce operations and those of Viper and VeriFast. We therefore expect them to also satisfy the OX properties of our interface (we have however not proven this fact).

ins and outs respectively, and the assertion $P = \bigvee_i (\exists \vec{y}_i.\ P_i)$ denotes the predicate body, which does not contain program variables and whose free logical variables are contained in $\{\vec{x}_{\text{in}}\} \cup \{\vec{x}_{\text{out}}\}$ which are disjoint from the bound variables $\vec{y}_i$, and where the P_i's (denoting the assertions of the predicate definition) do not contain disjunctions or existential quantifiers.

Satisfiability. The meaning of an assertion P is defined by capturing the models of P using the standard satisfaction relation $\theta, \sigma \models P$ where $\theta : \mathsf{LVar} \rightharpoonup_{\mathtt{fin}} \mathsf{Val}$ is a logical interpretation represented by a function from logical variables to values and σ is a program state (as defined in §3). The formal definition is included in the extended version of this paper [18].

Function Specifications. The quadruples $\{\mathsf{x} = \vec{x} \star P\}\ f(\vec{\mathsf{x}})\ \{ok : Q_{ok}\}\ \{err : Q_{err}\}$ and $[\mathsf{x} = \vec{x} \star P]\ f(\vec{\mathsf{x}})\ [ok : Q_{ok}]\ [err : Q_{err}]$ denote, respectively, a SL and an ISL function specification, as explained in §2.2. We write $\langle\!\langle \mathsf{x} = \vec{x} \star P \rangle\!\rangle\ f(\vec{\mathsf{x}})\ \langle\!\langle ok : Q_{ok} \rangle\!\rangle\ \langle\!\langle err : Q_{err} \rangle\!\rangle$ to refer to either. Both quadruples record successful executions and language errors. They are unable to record missing-resource errors, as these errors do not satisfy the OX and UX frame properties. Missing errors can be removed automatically via UX bi-abduction (see §7).

Formally, we define function specifications using internalisation [20]. In short, internalisation relates internal specifications, which describe the internal behaviour of functions, to external specifications, which describe the external behaviour of functions. Internalisation is needed for ISL to allow the logic to drop information about function-local program variables at function boundaries, since dropping information is in general not allowed in ISL. The full definitions of function specifications and internalisation are included in [18].

A function specification context, $\Gamma \in \mathsf{Fid} \rightharpoonup_{fin} \mathcal{P}(\mathcal{ES}pec)$, maps function identifiers to a finite set of external specifications $\mathcal{ES}pec$. To simplify the presentation of the paper, we assume existential quantifiers only occur at the top level of external specifications. We denote the validity of Γ with respect to γ by $\models (\gamma, \Gamma)$, and validity of a function specification with respect to Γ by $\Gamma \models \langle\!\langle \mathsf{x} = \vec{x} \star P \rangle\!\rangle\ f(\vec{\mathsf{x}})\ \langle\!\langle ok : Q_{ok} \rangle\!\rangle\ \langle\!\langle err : Q_{err} \rangle\!\rangle$.

Extended Symbolic States. To reason about unbounded execution to verify, for example, specifications of list algorithms, we extend the partial symbolic states defined in §4.1 with symbolic predicates of the form $p(\vec{v}_1; \vec{v}_2)$, with $p \in \mathsf{Str}$ and $\vec{v}_1, \vec{v}_2 \in \mathsf{S}\vec{\mathsf{Val}}$. An extended symbolic state $\hat{\sigma}$ is a tuple $(\hat{s}, \hat{H}, \hat{\pi})$ comprising a partial symbolic state $\hat{s}$, a symbolic resource $\hat{H} = (\hat{h}, \hat{\mathcal{P}})$ with symbolic heap $\hat{h}$ and multiset of symbolic predicates $\hat{\mathcal{P}}$, and a symbolic path condition $\hat{\pi}$. Definitions of well-formedness of symbolic state and symbolic interpretations are extended as expected. We define $\varepsilon, \sigma \models (\hat{s}, \hat{H}, \hat{\pi})$ analogously to assertion satisfaction since the interpretation of a symbolic state with respect to symbolic interpretation $\varepsilon : \mathsf{SVar} \rightharpoonup_{fin} \mathsf{Val}$ is a relation and not a function (cf. §4) due to the presence of symbolic predicates.

The composition of two extended symbolic states is defined by:

$$(\hat{s}_1, \hat{H}_1, \hat{\pi}_1) \cdot (\hat{s}_2, \hat{H}_2, \hat{\pi}_2) \triangleq (\hat{s}_1 \cup \hat{s}_2, \hat{H}_1 \cup \hat{H}_2, \hat{\pi}_1 \wedge \hat{\pi}_2 \wedge \mathcal{W}\!fc(\hat{H}_1 \cup \hat{H}_2))$$

where $\hat{H}_1 \cup \hat{H}_2$ denotes the pairwise union of the components of the symbolic resource and $\mathcal{W}\!fc(\hat{H}_1 \cup \hat{H}_2)$ ensures that the composition is well-formed.

5.2 Axiomatic Interface for Consume and Produce

We present our axiomatic interface for the consume and produce operations, used to update the symbolic state during function call and to fold and unfold the predicates. Given the substitution $\hat{\theta} : \mathsf{LVar} \rightharpoonup_{\mathtt{fin}} \mathsf{SVal}$, the consume and produce operations have the signatures:

$$\mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) \mid abort(\hat{v}) \qquad \mathsf{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$$

We assume the following holds initially: state $\hat{\sigma}$ and substitution $\hat{\theta}$ are well-formed; and $\hat{\theta}$ covers P for produce, that is, $\mathtt{lv}(P) \subseteq \mathrm{dom}(\hat{\theta})$. For properties 1–4 below, consider the following executions:

$\mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f)$ *where* $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$ *and* $\hat{\sigma}_f = (\hat{s}', \hat{H}_f, \hat{\pi}')$

$\mathsf{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$ *where* $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$ *and* $\hat{\sigma}' = (\hat{s}', \hat{H}', \hat{\pi}')$

▶ **Property 1** (Well-formedness). *The variable store is not altered: that is,* $\hat{s}' = \hat{s}$ *and*

(consume) $\mathcal{W}f(\hat{\sigma}_f)$ *and* $\mathcal{W}f(\hat{\theta}', \hat{\pi}')$ (*produce*) $\mathcal{W}f(\hat{\sigma}')$

▶ **Property 2** (Path Strengthening). $\hat{\pi}' \Rightarrow \hat{\pi}$

▶ **Property 3** (Consume Covers P). $\hat{\theta}' \geq \hat{\theta}$ *and* $\mathrm{dom}(\hat{\theta}') \supseteq \mathtt{lv}(P)$

▶ **Property 4** (Soundness).

(consume) $\exists \hat{H}_P.\ \hat{H} = \hat{H}_f \cup \hat{H}_P \wedge (\forall \varepsilon, \sigma.\ \varepsilon, \sigma \models \hat{\sigma}_P \implies \varepsilon(\hat{\theta}'), \sigma \models P)$ *where* $\hat{\sigma}_P \triangleq (\emptyset, \hat{H}_P, \hat{\pi}')$[a]

(produce) $\exists \hat{H}_P.\ \hat{H}' = \hat{H} \cup \hat{H}_P \wedge (\forall \varepsilon, \sigma.\ \varepsilon, \sigma \models \hat{\sigma}_P \implies \varepsilon(\hat{\theta}), \sigma \models P)$ *where* $\hat{\sigma}_P \triangleq (\emptyset, \hat{H}_P, \hat{\pi}')$

▶ **Property 5** (Completeness: OX consume). *If abort* $\notin$ *consume*$(\mathsf{OX}, P, \hat{\theta}, \hat{\sigma})$ *and* $\varepsilon, \sigma \models \hat{\sigma}$*, then*

$$\exists \hat{\theta}', \sigma_f, \hat{\sigma}_f.\ \mathit{consume}(\mathsf{OX}, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) \wedge \varepsilon, \sigma_f \models \hat{\sigma}_f$$

▶ **Property 6** (Completeness: UX consume). *If* $\mathit{consume}(\mathsf{UX}, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f)$*,* $\varepsilon(\hat{\pi}') = \mathsf{true}$ *and* $\varepsilon(\hat{\theta}'), (\emptyset, h_P) \models P$*, then*

$$\varepsilon, (\emptyset, h_P) \models \hat{\sigma}_P \wedge (\forall h_f.\ \varepsilon, (s, h_f) \models \hat{\sigma}_f \wedge h_P \sharp h_f \implies \varepsilon, (s, h_f \uplus h_P) \models \hat{\sigma})$$

▶ **Property 7** (Completeness: produce). *If* $\varepsilon, (s, h) \models \hat{\sigma}$ *and* $\varepsilon(\hat{\theta}), (\emptyset, h_P) \models P$ *and* $h \sharp h_P$*, then*

$$\exists \hat{\sigma}_P.\ \mathit{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma} \cdot \hat{\sigma}_P \wedge \varepsilon, (\emptyset, h_P) \models \hat{\sigma}_P$$

[a] We choose the empty symbolic store for $\hat{\sigma}_P$; P does not have program variables so this choice is arbitrary. The symbolic state $\hat{\sigma}_P$ is the one used in Prop. 6.

Figure 4 The axiomatic interface for the consume and produce operations.

Recall the use of the consume and produce operations in the function call illustrated in Fig. 2 of §2.2. For $\mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma})$, the initial substitution $\hat{\theta}$ comes from replacing the function parameters with symbolic values given by the arguments in the function call; consume matches the precondition P and substitution $\hat{\theta}$ against part of $\hat{\sigma}$, removing the appropriate resource $\hat{\sigma}_P$ and returning the frame $\hat{\sigma}_f$ and the substitution $\hat{\theta}'$ which extends $\hat{\theta}$ with further information given by the match. For $\mathsf{produce}(Q_{ok}, \hat{\theta}', \hat{\sigma}_f)$, the produce takes the postcondition Q_{ok} and this resulting substitution $\hat{\theta}'$ and creates a symbolic state which is composed with $\hat{\sigma}_f$ to obtain $\hat{\sigma}'$. Notice that the consume operation can abort with error information if no match is found. The produce operation does not abort, but it may render states unsatisfiable, in which case the branch is cut.

In Fig. 4, we present the axiomatic interface of the consume and produce operations, identifying sufficient properties to prove OX and UX soundness for the function-call rule and the folding and unfolding of predicates, as we demonstrate in the next section (§5.3). Properties 1–3 ensure that the operations are compatible with the expected properties of symbolic execution, including well-formedness $\mathcal{W}f(\hat{\theta}', \hat{\pi}')$ of the symbolic substitutions with respect to path conditions. This property guarantees that the $\hat{\pi}'$ implies that $\hat{\theta}'$ does not map logical variables into $\lightning$, that is, $\hat{\pi}' \vDash \mathrm{codom}(\hat{\theta}') \subseteq \mathsf{Val}$.

Properties 4–7 give conditions for consume and produce to soundly decompose and compose symbolic states respectively, while being compatible with symbolic and logical interpretations. These properties will come as no surprise to those with a formal knowledge of symbolic execution. However, their identification was not easy, requiring a considerable amount of back and forth between the soundness proof and the properties to pin them down properly. We describe the more interesting of the properties described in Fig. 4:

- Prop. 2 states that path conditions may only get strengthened: OX and UX consume may strengthen $\hat{\pi}$ to due to branching; additionally UX consume may strengthen $\hat{\pi}$ arbitrarily due to cutting; and produce may add constraints to $\hat{\pi}$ arising from P.
- Prop. 4, for consume (and similarly for produce), states that the operation is sound: the symbolic resource of $\hat{\sigma}$ can be decomposed as $\hat{H} = \hat{H}_f \cup \hat{H}_P$, i.e., it consists of the symbolic resources of $\hat{\sigma}_P$ and $\hat{\sigma}_f$, respectively, and $\forall \varepsilon, \sigma.\ \varepsilon, \sigma \models \hat{\sigma}_P \implies \varepsilon(\hat{\theta}''), \sigma \models P$ states that all models of $\hat{\sigma}_P$ are models of P.
- Prop. 5 captures that successful OX consumptions do not drop paths: if no branch aborts, and we have a model $\varepsilon, \sigma \models \hat{\sigma}$, then there exists a branch $\hat{\sigma}_f$ with a model using the same ε, i.e., there exist σ_f such that $\varepsilon, \sigma_f \models \hat{\sigma}_f$.
- Prop. 6 is as follows: in successful UX consume, any model of the consumed assertion P must also model the consumed state $\hat{\sigma}_P$ (obtained from Prop. 4), and when extended with a compatible model of the output state $\hat{\sigma}_f$ it must model the input state $\hat{\sigma}$.

Assuming these properties of the consume and produce operations, we are able to prove that the function-call rule and the predicate folding and unfolding are sound, and thus that our whole CSE engine is sound (Thm. 3). In §5.3, we give an example (Ex. 2) illustrating some of the properties in action during a function call.

5.3 Full CSE Engine

We introduce our full CSE engine, extending the core CSE engine (§4) with the ability to soundly call valid SL and ISL function specifications (§5.1), and to fold/unfold predicates. We extend and adapt the compositional symbolic operational semantics to carry a specification context Γ and the mode of execution m (OX or UX), obtaining the judgement $\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^m o : \hat{\sigma}'$, and extend the possible outcomes with *abort*, as consume can abort. The rules are analogous except for the rules for function calls and predicate folding/unfolding, as detailed below.[4]

Function-Call Rule. The unified success rule for a function call is in Fig. 5, using the notation $\Gamma(f)|_m$ to isolate the m-mode specifications of f. A description of each step is included in the rule itself. In short, given an initial state $\hat{\sigma}$, the rule selects a function specification, consumes the specification pre-condition from $\hat{\sigma}$, resulting in $\hat{\sigma}'$, and produces the post-condition of the specification into $\hat{\sigma}'$, resulting in the final state $\hat{\sigma}''$. The steps in grey are uninteresting (about renamings and fresh variables) and can be ignored on a first reading.

▶ **Example 2.** We show a possible execution of a function call using a function specification, where we assume we have been given consume and produce example implementations that satisfy the axiomatic interface. Consider the function f given in §2.1 and the ISL specification given in §2.3: $\left[\mathsf{c} = c \star \mathsf{x} = x \star P\right]\ \mathtt{f}(\mathsf{c}, \mathsf{x})\ \left[ok : x \mapsto c \star c \geq 42 \star \mathsf{ret} = v\right]$ where P is $x \mapsto v$,

[4] The satisfiability check $\mathtt{SAT}(\hat{\pi})$ used by the rules over-approximates the existence of a model for $(\hat{s}, \hat{H}, \hat{\pi})$, due to the presence of symbolic predicates; for sound reasoning in UX mode, our engine addresses this source of over-approximation by under-approximating the satisfiability check once by bounded unfolding of predicates at the end of execution.

$$
\begin{array}{lr}
(1)\ \langle\!\langle \vec{x} = \vec{x} \star P \rangle\!\rangle\ f(\vec{x})\ \langle\!\langle ok : Q_{ok} \rangle\!\rangle\ \langle\!\langle err : Q_{err} \rangle\!\rangle \in \Gamma(f)|_m & \text{get function specification} \\
(2)\ [\![\vec{\mathsf{E}}]\!]^{\hat{\pi}}_{\hat{s}} \Downarrow (\vec{v}, \hat{\pi}') \text{ and } \hat{\theta} \triangleq \{\vec{v}/\vec{x}\} & \text{evaluate function parameters} \\
(3)\ \mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma}[\mathsf{pc} := \hat{\pi}']) \rightsquigarrow (\hat{\theta}', \hat{\sigma}') & \text{consume pre-condition} \\
(4)\ Q_{ok} = \exists \vec{y}.\ Q'_{ok} & \text{as } Q_{ok} \text{ is a post-condition} \\
(5)\ \hat{\theta}'' \triangleq \hat{\theta}' \cup \{\vec{z}/\vec{y}\} & \text{extend substitution to cover } Q_{ok} \\
(6)\ \vec{z}, r, \hat{r} \text{ fresh} & \text{fresh variables} \\
(7)\ Q''_{ok} = Q'_{ok}\{r/\mathsf{ret}\} \text{ and } \hat{\theta}''' = \hat{\theta}'' \cup \{\hat{r}/r\} & \text{set up return value} \\
(8)\ \mathsf{produce}(Q''_{ok}, \hat{\theta}''', \hat{\sigma}') \rightsquigarrow \hat{\sigma}'' & \text{produce post-condition} \\
\hline
\hat{\sigma}, \mathsf{y} := f(\vec{\mathsf{E}}) \Downarrow^m_\Gamma\ ok : \hat{\sigma}''[\mathsf{sto} := \hat{s}[\mathsf{y} \mapsto \hat{r}]] & \\
\end{array}
$$

Figure 5 Unified function-call rule for CSE: success case, where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$.

here assumed to be in the function specification context Γ. Suppose the symbolic execution is in a state $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$ and that the next step is $\hat{\sigma}, \mathsf{y} := \mathsf{f}(50, 1) \Downarrow^{\mathsf{UX}}_\Gamma ok : \hat{\sigma}'$. Let $\hat{H} = (\{\hat{x} \mapsto \hat{c}, \hat{y} \mapsto 1, 3 \mapsto 5\}, \emptyset)$ be the symbolic resource, and $\hat{\pi} = \hat{c} \geq 42 \wedge \hat{x} \neq \hat{y} \wedge \hat{x}, \hat{y} \in \mathsf{Nat}$ be the symbolic path condition (the symbolic store $\hat{s}$ is irrelevant to this computation and left opaque). We now follow the steps (1) - (8) described in the function-call rule in Fig. 5.

Step (1) is above. Step (2) evaluates the parameters of the function call which in this case yields the initial substitution is $\hat{\theta} = \{50/c, 1/x\}$. Step (3) is to consume the pre-condition of f: $\hat{\theta}$ identifies the logical variable x with 1, and thus, this $\hat{\theta}$ maps P into $1 \mapsto v$; now we check whether there exists a resource in $\hat{H}$ that matches this. There are two possibilities: either $\hat{x} = 1$ and $v = \hat{c}$; or $\hat{y} = 1$ and $v = 1$. Let us choose the first match. Thus, with our axiomatic description of a consume operation, $\mathsf{consume}(\mathsf{UX}, x \mapsto v, \hat{\theta}, \hat{\sigma})$ gives the pair $(\hat{\theta}', \hat{\sigma}_f)$, with substitution $\hat{\theta}' = \{50/c, 1/x, \hat{c}/v\}$ and symbolic frame $\hat{\sigma}_f = (\hat{s}, (\{\hat{y} \mapsto 1, 3 \mapsto 5\}, \emptyset), \hat{\pi} \wedge \hat{x} = 1)$. Here $\hat{\theta}' \geq \hat{\theta}$ as described by Prop. 3 of Fig. 4, the new path condition $\hat{\pi} \wedge \hat{x} = 1$ is stronger than the initial $\hat{\pi}$, as required by Prop. 2.

Steps (4) - (5) are straightforward: Q_{ok} is not existentially quantified and the domain of $\hat{\theta}'$ covers $Q_{ok} = x \mapsto c \star c \geq 42 \star \mathsf{ret} = v$. Steps (6) - (7) set up the return value by renaming ret with a fresh logical variable r as in $Q''_{ok} = Q_{ok}\{r/\mathsf{ret}\}$ and defining the substitution $\hat{\theta}'' = \hat{\theta}' \uplus \{\hat{r}/r\}$, with $\hat{r}$ a fresh symbolic variable. Step (8) produces the post-condition which results in $(\hat{s}, (\{\hat{y} \mapsto 1, 3 \mapsto 5, 1 \mapsto 50\}, \emptyset), \hat{\pi}')$, for some $\hat{\pi}'$ that is satisfiable.

We illustrate the general execution of the function-call rule in Fig. 6. Successful $\mathsf{consume}$ may branch (in the figure: $\hat{\sigma}_{f_1}, \ldots, \hat{\sigma}_{f_k}$) due to different ways of matching with the symbolic state $\hat{\sigma}$, and the function call will branch accordingly. In both modes, in each successful branch, say with frame state $\hat{\sigma}_{f_i}$, the function-call rule will call $\mathsf{produce}$, which will produce both Q_{ok} and Q_{err} postconditions of the function specification. The function call propagates errors from $\mathsf{consume}$, whose error handling can depend on the mode of reasoning. In OX mode, all errors must be reported; the figure shows an example with two *abort* outcomes, one with a symbolic value $\hat{\sigma}_{miss}$, representing a missing outcome, and another abort $\hat{\sigma}_{\hat{v}}$. In UX mode, in contrast, errors can be cut: e.g., a $\mathsf{consume}$ implementation may choose to report missing errors (to be used in e.g. bi-abduction, see §7), but cut other errors, as illustrated in the figure. Lastly, note that $\mathsf{consume}$ implementations must represent missing-resource errors as abort errors. To see why, consider the function $\mathsf{do\_nothing}()\,\{\,\mathtt{skip};\mathtt{return\ null}\,\}$ and the (nonsensical but valid) specification $[5 \mapsto 0]\ \mathsf{do\_nothing}()\ [ok : 5 \mapsto 0 \star \mathsf{ret} = \mathtt{null}]$. Of course, in the concrete semantics, calling the function will never result in a miss. Now, say the symbolic engine calls the function using the provided function specification. If the the resource of the pre-condition is not available in the current symbolic heap, then the consumption of the pre-condition will fail. Because no concrete execution of the function results in a miss, it would be unsound for the consumption to report a missing-resource error in this case.

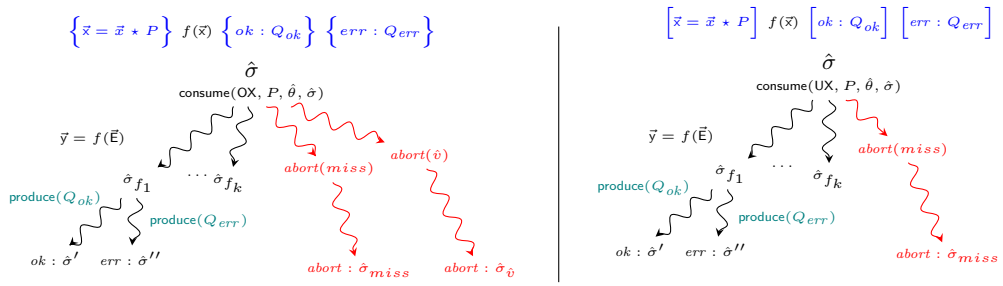


Figure 6 Branching in OX and UX function calls.

Predicate Rules. To handle the folding and unfolding of predicates in symbolic states, we extend the language syntax with the following two ghost commands (also known as tactic commands): $\mathsf{C} \in \mathsf{Cmd} ::= \cdots \mid \texttt{fold}\ p(\vec{\mathsf{E}}) \mid \texttt{unfold}\ p(\vec{\mathsf{E}})$, where $\vec{\mathsf{E}} \in \mathsf{PExp}$ specifies the values of the in-parameters of the predicate p. In the concrete semantics, these commands are no-ops, as they are ghost commands. The symbolic-semantics rules are similar to the function-call rule: in short, a fold of a predicate consumes the body of the predicate, learns the out-parameters of the predicate, and adds the predicate (with the specified in-parameters and learnt out-parameters) to the symbolic state; and an unfold of a predicate finds a corresponding predicate in the symbolic state, learns the out-parameters of the predicate, and produces the body of the predicate.

Soundness. Our CSE engine is sound: OX soundness, expectedly, has no restrictions on the predicates; UX soundness, on the other hand, allows only strictly exact predicates (i.e., predicates whose bodies are satisfiable by at most one heap [33]) to be folded to ensure that no information is dropped.

▶ **Theorem 3** (Compositional OX and UX soundness). *If all UX predicate foldings are limited to strictly exact predicates, then the following hold:*

$$\models (\gamma, \Gamma) \land \hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{OX} \hat{\Sigma}' \land \mathit{abort} \notin \hat{\Sigma}' \land \varepsilon, \sigma \models \hat{\sigma} \land \sigma, \mathsf{C} \Downarrow_\gamma o : \sigma' \implies \exists \hat{\sigma}', \varepsilon' \geq \varepsilon.\ (o, \hat{\sigma}') \in \hat{\Sigma}' \land \varepsilon', \sigma' \models \hat{\sigma}'$$

$$\models (\gamma, \Gamma) \land \hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{UX} o : \hat{\sigma}' \land \varepsilon, \sigma' \models \hat{\sigma}' \implies \exists \sigma.\ \varepsilon, \sigma \models \hat{\sigma} \land \sigma, \mathsf{C} \Downarrow_\gamma o : \sigma'$$

Proof. Proofs of rules not related to function calls and predicates are the same as for Thm. 1. OX soundness of function call follows from soundness of consume (Prop. 4), OX completeness of consume (Prop. 5), and completeness of produce (Prop. 7). UX soundness of function call follows from the soundness of consume and produce (Prop. 4) and UX completeness of consume (Prop. 6). Full details are given in the extended version of this paper [18]. ◀

6 Consume and Produce Implementations

We provide implementations for the consume and produce operations and prove that they satisfy the properties 1–7 of the axiomatic interface (§5.2). We give the complete set of rules implementing these operations in the extended version [18] and only discuss the interesting rules here. Our implementations are inspired by the Gillian OX implementations, although previous work has only given a brief informal sketch of these implementations [10].

6.1 Implementations

Consume Implementation. As is typical for SL-based analysis tools, our consume operation works with a fragment of the assertions with no implications, disjunctions, or existentials (which are handled outside consumption, see, e.g., the function-call rule); which means that input assertions for consumption are $\star$-separated lists of simple assertions. Following the implementation of Gillian, our consume operation works by consuming one simple assertion at a time and is split into two phases, a planning phase and a consumption phase:

$$\mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma}) \triangleq \mathsf{let}\ mp = \mathsf{plan}(\mathrm{dom}(\hat{\theta}), P)\ \mathsf{in}\ \mathsf{consume}_{\mathsf{MP}}(m, mp, \hat{\theta}, \hat{\sigma})$$

Here our interest lies in the consumption phase: the planning phase of Gillian has been formalised and discussed by Lööw et al. [19]. We, however, repeat the necessary background of the planning phase here to keep this paper self-contained.

Consumption Planning. The plan operation has two responsibilities: to resolve the order of consumption and the unknown variables. The operation takes a set of known logical variables (above, $\mathrm{dom}(\hat{\theta})$) and an assertion P to plan and returns a *matching plan* (*MP*) of the form $[(\mathsf{Asrt}, [(\mathsf{LVar}, \mathsf{LExp})])]$. An MP for an assertion $P = P_1 \star \ldots \star P_n$ ensures that (1) the simple assertions P_i of P are consumed in an order such that the *in-parameters* (*ins*) of each simple assertion, i.e., the parameters (logical variables) that must be known to consume the simple assertion, have been learnt during previous consumption; (2) specifies how *out-parameters* (*outs*) are learnt during consumption, i.e., the remaining parameters (logical variables). For instance, the in-parameter of the cell-assertion $x \mapsto z + 1$ is x and the out-parameter is z, where the value of z can be learnt by inspecting the heap and subtracting 1. Another example is given by the pure assertion $x + 1 = y + 3$: here, what the in- and out-parameters are depend on what variables are known, e.g., if we know x we can learn y and vice versa.

▶ **Example 4.** Say we are to plan the assertion $x \leq 10 \star x \mapsto y \star y = z - 10$ knowing that $\hat{\theta} = \{\hat{x}/x\}$, that is, x is known but y and z are not. One MP for this assertion is $[(x \leq 10, []), (x \mapsto y, [(y, \mathsf{O})]), (y = z - 10, [(z, y + 10)])]$, where O is used to refer to the value of the consumed heap cell. First, by consuming $x \leq 10$ we learn nothing (x is already known); second, when consuming $x \mapsto y$ we learn y (from the consumed heap cell); third, since we learn y in the previous step, we can learn z by manipulating the assertion to $z = y + 10$. Another MP is $[(x \mapsto y, [(y, \mathsf{O})]), (x \leq 10, []), (y = z - 10, [(z, y + 10)])]$. Note that there is no MP starting with assertion $y = z - 10$, because y and z are not known initially.

Consuming Assertions. Having discussed the planning phase, we now discuss how pure assertions, cell assertions, and predicate assertions are consumed.

Consuming Pure Assertions. Fig. 7 contains the $\mathsf{consume}_{\mathsf{MP}}$ rules for consuming pure assertions. The rules are defined in terms of the helper operation $\mathsf{consPure}(m, \hat{\pi}, \hat{\pi}') = \hat{\pi}'' \mid \mathit{abort}$ which depends on the current reasoning mode m:

- **(i)** for $m = \mathsf{OX}$, we check $\neg\mathtt{SAT}(\hat{\pi} \wedge \neg\hat{\pi}')$ which is equivalent to $\hat{\pi} \Rightarrow \hat{\pi}'$, hence, the SAT check corresponds to the entailment check seen in traditional OX reasoning;
- **(ii)** for $m = \mathsf{OX}$, if $\mathtt{SAT}(\hat{\pi} \wedge \neg\hat{\pi}')$, that is, $\neg(\hat{\pi} \Rightarrow \hat{\pi}')$, consumption, of course, must abort;
- **(iii)** for $m = \mathsf{UX}$, consPure instead cuts all paths of $\hat{\pi}$ that are not compatible with the input pure assertion $\hat{\pi}'$, i.e., forms $\hat{\pi} \wedge \hat{\pi}'$, and then checks if there are any paths left after the cut, i.e., checks $\mathtt{SAT}(\hat{\pi} \wedge \hat{\pi}')$.

$$\mathsf{consPure}(m, \hat{\pi}, \hat{\pi}') = \begin{cases} \hat{\pi} \wedge \hat{\pi}', & \text{if } m = \mathsf{UX} \text{ and } \mathtt{SAT}(\hat{\pi} \wedge \hat{\pi}') \\ \hat{\pi}, & \text{if } m = \mathsf{OX} \text{ and } \neg\mathtt{SAT}(\hat{\pi} \wedge \neg\hat{\pi}') \\ \mathit{abort}, & \text{if } m = \mathsf{OX} \text{ and } \mathtt{SAT}(\hat{\pi} \wedge \neg\hat{\pi}') \end{cases}$$

$$\frac{P \text{ is pure} \quad \mathit{outs} = [(x_i, E_i)|_{i=1}^n] \quad \hat{\theta}' = \hat{\theta} \uplus \{(\hat{\theta}(E_i)/x_i)|_{i=1}^n\} \quad \mathsf{consPure}(m, \hat{\pi}, \hat{\theta}'(P)) = \hat{\pi}'}{\mathsf{consume_{MP}}(m, [(P, \mathit{outs})], \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}[\mathsf{pc} := \hat{\pi}'])}$$

$$\frac{P \text{ is pure} \quad \mathit{outs} = [(x_i, E_i)|_{i=1}^n] \quad \hat{\theta}' = \hat{\theta} \uplus \{(\hat{\theta}(E_i)/x_i)|_{i=1}^n\} \quad \mathsf{consPure}(\mathsf{OX}, \hat{\pi}, \hat{\theta}'(P)) = \mathit{abort}}{\mathsf{consume_{MP}}(\mathsf{OX}, [(P, \mathit{outs})], \hat{\theta}, \hat{\sigma}) \rightsquigarrow \mathit{abort}([\text{“}\mathsf{consPure}\text{”}, \hat{\theta}'(P), \hat{\pi}])}$$

Figure 7 Rules for consPure and $\mathsf{consume_{MP}}$ (excerpt), where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$.

$$\frac{\hat{h} = \hat{h}_f \uplus \{\hat{v}_1 \mapsto \hat{v}_2\} \quad \hat{\pi}' = \hat{\pi} \wedge (\hat{v} = \hat{v}_1) \quad \mathtt{SAT}(\hat{\pi}')}{\mathsf{consCell}(\hat{v}, \hat{\sigma}) \rightsquigarrow (\hat{v}_2, \hat{\sigma}[\mathsf{heap} := \hat{h}_f, \mathsf{pc} := \hat{\pi}'])} \qquad \frac{\mathtt{SAT}(\hat{\pi} \wedge \hat{v} \notin \mathrm{dom}(\hat{h}))}{\mathsf{consCell}(\hat{v}, \hat{\sigma}) \rightsquigarrow \mathit{abort}}$$

$$\frac{\begin{array}{ll} \mathsf{consPure}(m, \hat{\pi}, \hat{\theta}(E_a) \in \mathsf{Nat}) = \hat{\pi}' & \text{check for evaluation error} \\ \mathsf{consCell}(\hat{\theta}(E_a), \hat{\sigma}[\mathsf{pc} := \hat{\pi}']) \rightsquigarrow (\hat{v}, \hat{\sigma}') & \text{branch over all cell consumptions} \\ \hat{\theta}_{subst} = \{\hat{v}/\mathsf{O}\} & \text{substitution with cell contents} \\ \mathit{outs} = [(x_i, E_i)|_{i=1}^n] \text{ and } ((\hat{\theta} \uplus \hat{\theta}_{subst})(E_i) = \hat{v}_i)|_{i=1}^n & \text{collect and instantiate outs} \\ \hat{\theta}' = \hat{\theta} \uplus \{(\hat{v}_i/x_i)|_{i=1}^n\} & \text{extend substitution with outs} \\ \mathsf{consPure}(m, (\hat{\sigma}').\mathsf{pc}, \hat{\theta}'(E_v) = \hat{v}) = \hat{\pi}'' & \text{consume cell contents} \end{array}}{\mathsf{consume_{MP}}(m, [(E_a \mapsto E_v, \mathit{outs})], \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}'[\mathsf{pc} := \hat{\pi}''])}$$

Figure 8 Rules for consCell and $\mathsf{consume_{MP}}$ (excerpt), where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$.

▶ **Example 5.** To exemplify the difference between OX and UX consume, consider calling a function `foo(y)` with the precondition $\mathtt{y} = y \star y \geq 0$. The first step of calling a function using its function specification is to consume its precondition, which we now illustrate. Say we are in a symbolic state with path condition $\hat{\pi} = \hat{v} > 5$ and are calling the function with an argument that symbolically evaluates to $\hat{v}$, i.e., we know $\hat{\theta}(y) = \hat{v}$. In OX mode, the function call aborts: $\mathsf{consume_{MP}}$'s pure consumption error rule is applicable because $\mathsf{consPure}(\mathsf{OX}, \hat{\pi}, \hat{\theta}(y) \geq 10) = \mathit{abort}$ since $\mathtt{SAT}(\hat{\pi} \wedge \neg(\hat{v} \geq 10))$. Intuitively, this means that not all paths described by $\hat{\pi}$ are described by $y \geq 10$, i.e., we are "outside" the precondition of the function. Differently, in UX mode, a call to $\mathsf{consPure}(\mathsf{UX}, \hat{\pi}, \hat{\theta}(y) \geq 10)$ cuts the incompatible paths by strengthen the path condition to $\hat{\pi} \wedge \hat{v} \geq 10$. That is, instead of as in OX mode where execution must abort, in UX mode the execution can continue.

Consuming Cell Assertions. Fig. 8 contains some of the $\mathsf{consume_{MP}}$ rules for consuming a cell assertion. The rules are defined using the helper operation $\mathsf{consCell}(\hat{v}, \hat{\sigma}) \rightsquigarrow (\hat{v}', \hat{\sigma}') \mid \mathit{abort}$, which tries to consume the cell at address $\hat{v}$ in mode m by branching over all possible addresses in the heap, returning the corresponding value in the heap, $\hat{v}'$, and the rest of the state, $\hat{\sigma}'$, and returns *abort* if it is possible for the address $\hat{v}$ to point outside of heap. In the successful $\mathsf{consume_{MP}}$ rule (featured in Fig. 8), the operation consPure is used to consume the contents of the matched cells. The erroneous $\mathsf{consume_{MP}}$ rules are available in [18].

Consuming Predicate Assertions. The $\mathsf{consume_{MP}}$ rules for predicate assertions are generalisations of the rules for cell assertions, with two main differences: predicates may have multiple *ins* and *outs* whereas cells have a single *in* and single *out*, and predicate assertions refers to symbolic predicates whereas cell assertions refer to the symbolic heap.

$$\frac{\hat{h}' \triangleq \hat{h} \uplus \{\hat{l} \mapsto \hat{v}_\varnothing\} \quad \hat{\pi}' \triangleq \hat{\pi} \wedge \hat{l} \notin \mathsf{dom}(\hat{h}) \quad \mathtt{SAT}(\hat{\pi}') \quad \hat{\sigma}' \triangleq \hat{\sigma}[\mathsf{heap} := \hat{h}', \mathsf{pc} := \hat{\pi}']}{\mathsf{prodCell}(\hat{l}, \hat{v}_\varnothing, \hat{\sigma}) = \hat{\sigma}'} \qquad \frac{P \text{ pure} \quad \hat{\pi}' \triangleq \hat{\pi} \wedge \hat{\theta}(P) \quad \mathtt{SAT}(\hat{\pi}') \quad \hat{\sigma}' \triangleq \hat{\sigma}[\mathsf{pc} := \hat{\pi}']}{\mathsf{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'}$$

$$\frac{\hat{\pi}' \triangleq \hat{\pi} \wedge \hat{\theta}(E_a) \in \mathsf{Nat} \wedge \hat{\theta}(E_v) \in \mathsf{Val} \quad \mathsf{prodCell}(\hat{\theta}(E_a), \hat{\theta}(E_v), \hat{\sigma}[\mathsf{pc} := \hat{\pi}']) = \hat{\sigma}'}{\mathsf{produce}(E_a \mapsto E_v, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'}$$

Figure 9 Rules for prodCell and produce (excerpt), where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$ and $\hat{v}_\varnothing$ denotes a symbolic value or $\varnothing$.

Produce Implementation. The implementation of $\mathsf{produce}(Q, \hat{\theta}, \hat{\sigma})$ is straightforward: it extends $\hat{\sigma}$ with the symbolic state corresponding to Q given $\hat{\theta}$, ensuring well-formedness. Unlike consume, produce does not require planning and is not dependent on the mode of execution. An excerpt of rules for produce is given in Fig. 9. Like consume, produce does not support assertion-level implications, which are usually not found in function specifications or predicate definitions. However, produce, unlike consume, supports assertion-level disjunctions since the function specification we synthesise using bi-abduction contains disjunctions (cf. §8).

6.2 Correctness of Implementations

The correctness of the consume and produce implementations amount to showing that they satisfy properties of the axiomatic interface for consume and produce.

▶ **Theorem 6** (Correctness). *The consume and produce operations satisfy properties 1-7 (§5.2).*

7 Bi-abduction

To enable hosting Pulse-style true bug-finding on top of our CSE engine, it must support UX bi-abduction. In this section, we show how the engine presented in §5 can be extended to support UX bi-abduction by catching missing-resource errors that happen during execution and applying *fixes* to enable uninterrupted execution instead of faulting. These fixes, stored in an *anti-frame*, add the missing resource to the current state and allow execution to continue. This style of bi-abduction was introduced in the OX setting by JaVerT 2.0 [10]. Here we show that it can also be applied to the UX setting. We focus on UX bi-abduction for true bug-finding, but also discuss in §8 how the obtained UX results can be used in an OX setting.

We introduce the judgement for the bi-abductive symbolic engine, $\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{bi}} o : (\hat{\sigma}', \hat{H})$, with outcomes $o ::= ok \mid err$, and the anti-frame $\hat{H} = (\hat{h}, \hat{\mathcal{P}})$ containing the anti-heap $\hat{h}$ and the anti-predicates $\hat{\mathcal{P}}$. We do not need *miss* or *abort* as possible outcomes since if they happen during execution they will be either fixed by bi-abduction or cut if not. The new judgement is defined in terms of the judgement $\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{UX}} o : \hat{\sigma}'$ and a partial function `fix` as:

Biab
$$\frac{\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{UX}} o : \hat{\sigma}' \quad \mathsf{not\_Seq}(\mathsf{C}) \quad o \notin \{miss, abort\}}{\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{bi}} o : (\hat{\sigma}', (\emptyset, \emptyset))}$$

Biab-Miss
$$\frac{\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{UX}} o : \hat{\sigma}' \quad \mathsf{not\_Seq}(\mathsf{C}) \quad o \in \{miss, abort\} \quad \mathtt{fix}(\hat{\sigma}') = (\hat{H}, \hat{\pi}) \quad \hat{\sigma} \cdot (\hat{H}, \hat{\pi}), \mathsf{C} \Downarrow_\Gamma^{\mathsf{bi}} o' : (\hat{\sigma}'', \hat{H}')}{\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^{\mathsf{bi}} o' : (\hat{\sigma}'', \hat{H} \cup \hat{H}')}$$

Biab-Seq-Err
$$\frac{\hat{\sigma}, \mathsf{C}_1 \Downarrow_\Gamma^{\mathsf{bi}} o : (\hat{\sigma}', \hat{H}) \quad o \neq ok}{\hat{\sigma}, \mathsf{C}_1; \mathsf{C}_2 \Downarrow_\Gamma^{\mathsf{bi}} o : (\hat{\sigma}', \hat{H})}$$

Biab-Seq
$$\frac{\hat{\sigma}, \mathsf{C}_1 \Downarrow_\Gamma^{\mathsf{bi}} ok : (\hat{\sigma}', \hat{H}_1) \quad \hat{\sigma}', \mathsf{C}_2 \Downarrow_\Gamma^{\mathsf{bi}} o : (\hat{\sigma}'', \hat{H}_2) \quad \mathtt{sv}(\mathsf{dom}(\hat{H}_2)) \cap (\mathtt{sv}(\hat{\sigma}') \setminus \mathtt{sv}(\hat{\sigma})) = \emptyset}{\hat{\sigma}, \mathsf{C}_1; \mathsf{C}_2 \Downarrow_\Gamma^{\mathsf{bi}} o : (\hat{\sigma}'', \hat{H}_1 \cup \hat{H}_2))}$$

where $\mathsf{not\_Seq}(\mathsf{C})$ denotes that C is not a sequence command (i.e., does not have the form $\mathsf{C}_1; \mathsf{C}_2$), $\hat{\sigma} \cdot (\hat{H}, \hat{\pi})$ denotes $\hat{\sigma} \cdot (\emptyset, \hat{H}, \hat{\pi} \wedge \mathcal{W}\!fc(\hat{H}))$, $\mathsf{dom}(\hat{H}_2) = \mathtt{sv}(\mathsf{dom}(\hat{h}_2)) \cup \mathtt{sv}(\mathsf{dom}(\hat{\mathcal{P}}_2))$, and $\mathsf{dom}(\hat{\mathcal{P}})$ denotes all symbolic variables of the *ins* of the predicates in $\hat{\mathcal{P}}$. The rule Biab states that for non-erroneous outcomes, the bi-abductive engine has the same semantics as the underlying UX engine it is built on top of. The rule Biab-Miss, which is the most interesting rule, catches missing-resource errors from the underlying UX engine and uses the `fix` function to add the missing resource to the current symbolic state and anti-frame, such that execution can continue. The two rules Biab-Seq-Err and Biab-Seq are two straightforward sequencing rules for the engine, where the symbolic-variable condition of Biab-Seq ensures that the anti-frame $\hat{H}_2$ does not clash with resource allocated by C_1.

To exemplify, say the engine is in symbolic state $(\{\mathsf{v} \mapsto 0, \mathsf{a} \mapsto 13\}, (\emptyset, \emptyset), \mathsf{true})$ and is about execute $\mathsf{v} := [\mathsf{a}]$, i.e., about to retrieve the value of the heap cell with address a. Since this cell is not in the heap, the rule Lookup-Err-Missing from Fig. 3 is applicable, which sets the variable err to $[\text{“}\mathsf{MissingCell}\text{”}, \text{“}a\text{”}, 13]$ and gives outcome *miss*. Now, in the rule Biab-Miss, given the data in the err variable, the `fix` function constructs a fix $((\{13 \mapsto \hat{v}\}, \emptyset), \mathsf{true})$ where $\hat{v}$ is a fresh variable. The rule adds this fix to both the current symbolic state and the anti-frame, resulting in the symbolic state $(\{\mathsf{v} \mapsto \hat{v}, \mathsf{a} \mapsto 13\}, (\{13 \mapsto \hat{v}\}, \emptyset), \mathsf{true})$ and outcome *ok*. Other cases are similar. E.g., when abort outcomes from $\mathsf{consume}$ represent missing resource (e.g., when invoked in a function call), `fix` returns the resources needed for the execution to continue. The following theorem captures the essence of bi-abduction:

▶ **Theorem 7** (CSE with Bi-Abduction: UX Soundness).

$$\hat{\sigma}, \mathsf{C} \Downarrow_{\Gamma}^{bi} o : (\hat{\sigma}', \hat{H}) \implies \hat{\sigma} \cdot (\hat{H}, \mathsf{true}), \mathsf{C} \Downarrow_{\Gamma}^{UX} o : \hat{\sigma}'$$

8 Analysis Applications

We discuss the three analysis applications we have built on top of our unified CSE engine, to demonstrate its wide applicability: EX whole-program automatic symbolic testing (§8.1); OX semi-automatic verification (§8.2); and UX automatic true bug-finding (§8.3).

We have gathered these three analyses from different corners of the literature. EX symbolic testing is well understood in the first-order symbolic execution literature. OX verification is well understood in the consume-produce symbolic execution literature. However, one novelty here is that the correctness proof of the analysis is established with respect to our axiomatic interface rather than the consume/produce operations directly, allowing us to show that function specifications are valid w.r.t. the standard SL definition of validity. In contrast to the other two analyses, UX bug-finding has not previously been implemented in consume-produce style, making this a novel contribution. To simplify the presentation, we consider only non-recursive functions. All applications can be extended to handle bounded recursion by adding a fuel parameter. Unbounded recursion can be handled in verification via user-provided annotations, but is not a good fit for automatic bug-finding.

8.1 EX Whole-program Symbolic Testing

Our EX core engine allows us to implement simple non-compositional analyses, such as whole-program symbolic testing, in the style of CBMC [16] and Gillian [11]. For this analysis, we augment the input language with three additional commands: $\mathsf{x} := \mathsf{sym}$, for creating symbolic variables; $\mathtt{assume}(\mathsf{E})$, for imposing a constraint E on the current state; and $\mathtt{assert}(\mathsf{E})$, for checking that E is true in the current state. The operational semantics for these commands is given in the extended version of this paper [18].

The testing algorithm is as follows. Given a command C and implementation context γ, the analysis starts from the state $\hat{\sigma} \triangleq (\{\mathsf{x} \mapsto \mathtt{null} \mid \mathsf{x} \in \mathtt{pv}(C)\}, \emptyset, \mathsf{true})$, and executes C to completion. The analysis reports back any violations of the assert commands encountered during execution. Given the core engine is UX sound, any bug found will be a true bug. Moreover, if the analysed code contains no unbounded recursion, given the core engine is OX sound, all existing bugs will be found modulo the ability of the underlying SMT solver.

8.2 OX Verification

We formalise an OX verification procedure, `verifyOX`, on top of our CSE engine. Given a specification context Γ, a function $f(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \mathtt{return}\ \mathsf{E}\,\}$ with $f \notin \mathrm{dom}(\Gamma)$, and an OX specification $t_f = \{\vec{\mathsf{x}} = \vec{x} \star P\}\ \{ok : Q_{ok}\}\ \{err : Q_{err}\}$, if $\mathtt{verifyOX}(\Gamma, f, t_f)$ terminates successfully, then we can soundly extend Γ to $\Gamma' = \Gamma[f \mapsto t_f]$. The algorithm is given below:

1. Let $\hat{\theta} \triangleq \{\hat{x}/x \mid x \in \mathtt{lv}(\vec{\mathsf{x}} = \vec{x} \star P)\}$, $\hat{s} \triangleq \{\mathsf{x} \mapsto \hat{x} \mid \mathsf{x} \in \vec{\mathsf{x}}\} \cup \{\mathsf{x} \mapsto \mathtt{null} \mid \mathsf{x} \in \mathtt{pv}(\mathsf{C}) \setminus \vec{\mathsf{x}}\}$, and $\hat{\sigma} = (\hat{s}, \emptyset, \mathsf{true})$.
2. Set up symbolic state corresponding to pre-condition: $\mathsf{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$.
3. Execute the function to completion: $\hat{\sigma}', \mathsf{C}; \mathsf{ret} := \mathsf{E} \Downarrow_{\Gamma}^{\mathsf{OX}} \hat{\Sigma}'$. Then, for every $(o, \hat{\sigma}'') \in \hat{\Sigma}'$:
   (a) If $o = \mathit{miss}$ or $o = \mathit{abort}$, abort with an error.
   (b) If $o = ok$, then let $\hat{\theta}' = \hat{\theta} \uplus \{(\hat{\sigma}''.\mathtt{sto})(\mathsf{ret})/r\}$ for a fresh r and let $Q' = Q_{ok}\{r/\mathsf{ret}\}$. Otherwise, $o = err$, in which case let $\hat{\theta}' = \hat{\theta}$ and $Q' = Q_{err}$.
   (c) Consume the post-condition: $\mathsf{consume}(\mathsf{OX}, Q', \hat{\theta}', \hat{\sigma}'') \rightsquigarrow (\hat{\theta}'', \hat{\sigma}''')$, where $Q' = \exists \vec{y}.\ Q''$.
   (d) If consumption fails or the final heap is not empty, abort with an error.[5]

8.3 UX Specification Synthesis and True Bug-finding

Recall that Pulse-style UX bug-finding is powered by UX specification synthesis, where, after appropriate filtering, synthesised erroneous specification can be reported as bugs. UX specification synthesis, in turn, is powered by UX bi-abduction (as introduced in §7).

To formalise the specification synthesis procedure, we first define the toAsrt function, which takes a symbolic state $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$, where $\hat{H} = (\hat{h}, \hat{\mathcal{P}})$, and returns the corresponding assertion. The function is simple to implement: $\hat{s}$ becomes a series of equalities, $\hat{h}$ becomes a series of cell assertions, $\hat{\mathcal{P}}$ are lifted to predicate assertions and $\hat{\pi}$ is lifted to a pure assertion. Using toAsrt, we can also transform multiple symbolic states into an assertion by transforming them individually and gluing together the obtained assertions using disjunction.

We generate UX function specifications using the $\mathsf{synthesise}(\Gamma, f, P)$ algorithm, which takes a specification context Γ, a function $f(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \mathtt{return}\ \mathsf{E}\,\}$ and its candidate pre-condition, $\vec{\mathsf{x}} = \vec{x} \star P$, and uses bi-abduction to generate a set of UX specifications describing the behaviour of f starting from P. As $P = \mathsf{emp}$ is a valid starting point, $\mathsf{synthesise}$ can be applied to any function without a priori knowledge. The $\mathsf{synthesise}$ algorithm is as follows:

1. Let $\hat{\theta} \triangleq \{\hat{x}/x \mid x \in \mathtt{lv}(\vec{\mathsf{x}} = \vec{x} \star P)\}$, $\hat{s} \triangleq \{\mathsf{x} \mapsto \hat{x} \mid \mathsf{x} \in \vec{\mathsf{x}}\} \cup \{\mathsf{x} \mapsto \mathtt{null} \mid \mathsf{x} \in \mathtt{pv}(\mathsf{C}) \setminus \vec{\mathsf{x}}\}$, and $\hat{\sigma} = (\hat{s}, \emptyset, \mathsf{true})$.
2. Add the symbolic representation of P to $\hat{\sigma}$: $\mathsf{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$
3. Execute the function, obtaining a set of traces: $\hat{\sigma}', \mathsf{C}, \mathsf{ret} := \mathsf{E} \Downarrow_{\Gamma}^{\mathrm{bi}} \{(o_i, (\hat{\sigma}_i', \hat{H}_i))|_{i \in I}\}$.
4. Then, for every obtained $(o_i, ((\hat{s}_i', \hat{H}_i', \hat{\pi}_i'), \hat{H}_i))$:
   a. Complete the candidate pre-condition: $P_i \triangleq P \star \mathsf{toAsrt}((\emptyset, \hat{H}_i, \mathsf{true}))$.

[5] This check is required due to our classical (linear) treatment of resource, appropriate for languages with explicit deallocation rather than garbage collection.

b. Restrict the final store to the return/error variable: $\hat{s}_i'' \triangleq \hat{s}_i'|_{\{\mathsf{x}\}}$, where $\mathsf{x} = \mathsf{ret}$ if $o_i = ok$ and $\mathsf{x} = \mathsf{err}$ otherwise.
c. Create the post-condition: $Q_i \triangleq \mathsf{toAsrt}(\hat{s}_i'', \hat{H}_i', \hat{\pi}_i')$.
d. Return $[P_i]\ f(\vec{\mathsf{x}})\ [o_i :\ \exists \vec{y}.\ Q_i]$, where $\vec{y} \triangleq \mathtt{lv}(Q_i) \setminus \mathtt{lv}(P_i)$.

▶ **Theorem 8** (Correctness of synthesise).

$$[P']\ f(\vec{\mathsf{x}})\ [o :\ \exists \vec{y}.\ Q] \in \mathsf{synthesise}(\Gamma, f, P) \implies \Gamma \models [P']\ f(\vec{\mathsf{x}})\ [o :\ \exists \vec{y}.\ Q]$$

▶ Remark 9. Step 4b corresponds to forgetting the local variables when moving from internal to external post-condition since symbolic states only have program variables in the store.

▶ Remark 10. Front-end heuristics to filter out "interesting" bugs, i.e., synthesised erroneous specification, can be easily implemented on top of our bi-abduction (e.g., filtering for manifest bugs as per Lee et al. [17]); for this paper, however, we are foremost interested in back-end engine development and therefore consider such front-end issues out of scope.

▶ Remark 11. Specifications with the same anti-frame can be coalesced into one via disjunction of their post-conditions. Moreover, if a specification does not branch on symbolic variables created by the execution of C, the pure part of the post-condition can be lifted to the pre-condition to create an EX specification [20], which can then be used both in OX verification and UX true bug-finding.

▶ Remark 12. Automatic predicate folding and unfolding may be required in some cases to prevent redundant fixes: e.g., if $\mathsf{y} := \mathsf{g}(); \mathsf{x} := [\mathsf{y}]$ and g has post-condition $\mathsf{list}(\mathsf{ret}, vs)$, the list predicate should be unfolded for the lookup to access the first value in the list. Gillian has heuristics-based automatic folding and unfolding, but we leave its description and the evaluation of its compatibility with bi-abduction for future work. Without automatic folding and unfolding, code must not break the interface barrier of the data structures it uses.

9 Evaluation

We have evaluated our CSE engine in the following two practical ways.

9.1 Companion Haskell Implementation

With our engine formalism, we have developed a companion Haskell implementation to demonstrate that the formalism is implementable (and to catch errors early by executing simple examples). The Haskell implementation follows the inference rules of $\hat{\sigma}, \mathsf{C} \Downarrow_\Gamma^m o : \hat{\sigma}'$ given in §5, and the specific consume and produce operations in §6. We have implemented the search through the inference rules inside a symbolic execution monad, similar to other monads in the literature [7, 22]; the monad handles, e.g., demonic non-determinism (branching), angelic non-determinism (backtracking), per-branch state and global state.

9.2 Gillian OX and UX Compositional Analysis Platform

Our unified CSE engine took direct inspiration from the Gillian compositional OX platform. Using the ideas presented here, we returned to Gillian and adapted its CSE engine to handle both SL and ISL function specifications with real-world consume-produce implementations. Leveraging the identified difference between OX and UX reasoning, we were able to introduce UX reasoning to Gillian by adding, in essence, a OX/UX flag to the corresponding function. As these changes were isolated, existing analyses implemented in Gillian remain unaffected, including Gillian's whole-program symbolic testing, previously evaluated on the Collections-C library [11], and Gillian's compositional OX verification, previously evaluated on AWS

Table 1 Aggregated results of synthesising function specifications for the Collections-C library (commit `584e113`). Results were obtained by setting the loop and recursive call unrolling limit to 3, on a MacBook Pro 2019 laptop with 16 GB memory and a 2.3 GHz Intel Core i9 CPU.

Library	Functions	GIL Inst.	Succ. Specs	Err. Specs	Time (s)
array	45	1784	251	260	1.36
deque	47	2312	271	210	2.25
hashset	14	160	7	112	9.43
hashtable	28	1527	31	147	15.67
list	66	2977	454	615	5.59
pqueue	10	557	90	51	3.96
queue	16	85	133	67	1.36
rbuf	9	181	9	17	0.07
slist	52	2269	292	1873	24.49
stack	16	85	136	88	0.50
treeset	17	214	28	106	0.36
treetable	36	1601	144	276	1.55
other	8	139	14	11	0.03
Total	**364**	**13891**	**1860**	**3833**	**66.62**

code [21]. In addition, we have implemented UX bi-abduction in Gillian, following the fixes-from-errors approach presented in §7, where functions are evaluated bottom-up along the call graph and previously generated specifications are used at call sites.

To evaluate Gillian's new support for UX reasoning, we have tested its new UX bi-abduction analysis on real-world code, specifically, the Collections-C [25] data-structure library for C. As discussed in §8, specification synthesis using UX bi-abduction constitutes the back-end of Pulse-style bug-finding and is its most time-consuming part. The Collections-C library has 2.6K stars on GitHub and approximately 5.2K lines of code, and it uses many C constructs and idioms such as structures and pointer arithmetic. The data structures it provides include, e.g., dynamic arrays, linked lists, and hash tables. To carry out the evaluation, we extended previous work where the Gillian platform has been instantiated to the C programming language, called Gillian-C. Tbl. 1 presents the results of our new UX bi-abduction analysis, grouped by the data structures of the library: the numbers of associated functions; number of corresponding GIL instructions (GIL is the intermediate language used by Gillian); the number of success and error specifications; and the analysis time. Since one specification is synthesised per execution path, the number of specifications reflect the number of execution paths the Gillian engine was able to construct using bi-abduction. In summary, Gillian-C synthesises specifications for 364 functions of the Collections-C library, producing 5693 specifications in 66.92 seconds. We believe the results are promising both in terms of performance and number of specifications synthesised. One anomaly is that 58% of the execution time is spent on 3 of the 343 functions, leading to the creation of 1640 specifications. This anomaly arises because of the memory model currently in use by Gillian-C and not from a limitation of our formalisation or of the Gillian engine. More detailed analysis and a selection of generated specifications are available in [18].

10 Related Work

First-order Compositional Symbolic Execution. Static symbolic execution tools and frameworks based on first-order logic, such as CBMC [16] and Rosette [32, 27], can be made functionally compositional with respect to the variable store but not with respect to arbitrary state because they are not able to specify functions that manipulate memory in a way that would make the reasoning scalable.

Compositional Symbolic Execution. We work with a CSE engine with consume and produce operations, as found in, e.g., the OX tools VeriFast [14], Viper [23], and Gillian [11, 21]. In contrast, an alternative approach is to describe CSE inside a separation logic using proof search, as found in, e.g., Smallfoot [2, 3], Infer [4], and Infer-Pulse [17].

Here, we focus on formalising a CSE engine with consumers and producers, which more accurately models tool implementations. All the current consume-produce tools are based on OX reasoning. Some have detailed work on formalisation: Featherweight VeriFast [15] provides a Coq mechanisation inspired by VeriFast; Schwerhoff's PhD thesis [30] and Zimmerman et al. [35] provide detailed accounts of Viper's symbolic execution backend. Previous work has not, like us, introduced an axiomatic interface for their consume and produce operations. Because of the interface, our results are established using function specifications whose meaning is defined in standard SL/ISL-style, in particular, using the standard satisfaction relation for assertions defined independent of the choice of our CSE engine. This means that we can use specifications developed outside of our engine, e.g., using theorem provers, and vice versa. In contrast, the work on Featherweight VeriFast does not define an assertion satisfaction relation independent of their consume and produce operations. Schwerhoff does not give a soundness theorem at all. Lastly, Zimmerman et al. give a standard satisfaction relation (for implicit dynamic frames [31], a variant of SL [26]) but only embed this relation inside their concrete semantics instead of working with the standard definitions of function specifications (see Fig. 11 of their paper). Finally, we have demonstrated that our engine semantics provides a common foundation for OX and UX reasoning, with the difference in the underlying engine only amounting to the choice to use satisfiability or validity. This allows a straightforward extension of Gillian to support UX reasoning.

Bi-abduction. Bi-abduction was originally introduced for OX reasoning [5, 6] and led to Meta's automatic Infer tool for bug-finding [4]. Recently, it was reworked for UX reasoning and led to Meta's Infer-Pulse for true bug-finding [28, 17]. In these works, compositional symbolic execution is formalised using proof search, in the style of the Smallfoot description of symbolic execution [2, 3], with bi-abduction embedded into that proof search. In contrast, our UX bi-abduction is formulated as a separate layer on top of our CSE engine, establishing fixes from missing-resource errors using an idea introduced for OX bi-abduction by JaVerT 2.0 [10].

Alternating between OX and UX Reasoning. Smash by Godefroid et al. [13] is the most well-known tool to combine OX and UX reasoning. It is a first-order tool which "alternates" between OX and UX reasoning to speed up the program analysis implemented by the tool. However, citing Le et al. [17], who in turn report on personal communication with Godefroid, Smash-style analyses seem to have faced obstacles when put into practice in that they were "used in production at Microsoft, but are not used by default widely in their deployments, because other techniques were found which were better for fighting path explosion." Our CSE engine, in contrast, has a completely different motivation in that its purpose is to host different types of OX and UX analyses.

Program Correctness and Incorrectness. We know of two program logics that work with both program correctness and incorrectness. Exact separation logic (ESL) [20] combines the guarantees of both SL and ISL, providing exact function specifications compatible with both OX verification and UX true bug-finding. Exact specifications are compatible with our CSE engine, e.g., in UX mode the engine can call exact unbounded function specifications of list algorithms and still preserve true bug-finding. Outcome logic (OL) [34] is based on OX

Hoare logic with a different approach to handling incorrectness based on the reachability of sets of states. No tool is currently based on OL; in the future, we hope to be able to extend straightforwardly our unified approach to incorporate OL.

11 Conclusions

We have introduced a compositional symbolic execution engine capable of creating and using function specifications arising from an underlying separation logic. Our engine is formally defined using a novel axiomatic interface which ensures a sound link between the execution engine and the function specifications using consume and produce operations. Thus, our engine creates function specifications usable by other tools, and uses function specifications from various sources, including theorem provers and pen-and-paper proofs. Additionally, we have captured the essence of the Gillian consume and produce implementations both operationally, using inference rules, and via an accompanying Haskell implementation, and shown that our operational description satisfies the properties of the axiomatic interface. In this way, we offer a degree of assurance that the real-world, heavily-optimised Gillian implementation is correct.

A surprising property of our semantics is that it provides a common foundation for both OX reasoning based on SL, and UX reasoning based on ISL. By leveraging the minimal differences between the OX and UX engines, we have extended the OX Gillian platform to support UX reasoning. This extension includes function specifications underpinned by ISL, enabling automatic true bug-finding using UX bi-abduction which our engine incorporates by creating fixes from missing-resource errors. We evaluate our extension using the Gillian instantiation to C, the first real-world tool to support both compositional correctness and incorrectness reasoning, grounded on a common formal compositional symbolic execution engine. Our instantiation preserves the previous OX verification evaluated on AWS code [21] and now automatically synthesises UX function specifications for the real-world Collections-C library using our UX bi-abduction technique.

We believe that our axiomatic interface and formalisation of UX bi-abduction serve as re-usable techniques, which we hope will provide valuable guidance for the implementation of the next-generation compositional symbolic execution engines.

References

1 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018. doi:10.1145/3182657.

2 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Conference on Formal Methods for Components and Objects*, 2005. doi:10.1007/11804192_6.

3 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, 2005. doi:10.1007/11575467_5.

4 Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_33.

5 Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages*, 2009. doi:10.1145/1480881.1480917.

6 Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), 2011. doi:10.1145/2049697.2049700.

7 David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017. doi:10.1145/3110256.

8 José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *Principles and Practice of Declarative Programming*, 2018. doi:10.1145/3236950.3236956.

9 José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: Javascript verification toolchain. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158138.

10 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290379.

11 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part I: A multi-language platform for symbolic execution. In *Programming Language Design and Implementation*, 2020. doi:10.1145/3385412.3386014.

12 Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Principles of Programming Languages*, 2012. doi:10.1145/2103656.2103663.

13 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Principles of Programming Languages*, 2010. doi:10.1145/1706299.1706307.

14 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_4.

15 Bart Jacobs, Frédéric Vogels, and Frank Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11, 2015. doi:10.2168/LMCS-11(3:19)2015.

16 Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2014. doi:10.1007/978-3-642-54862-8_26.

17 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022. doi:10.1145/3527325.

18 Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. Compositional symbolic execution for correctness and incorrectness reasoning (extended version), 2024. doi:10.48550/arXiv.2407.10838.

19 Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Petar Maksimović, and Philippa Gardner. Matching plans for frame inference in compositional reasoning. In *European Conference on Object-Oriented Programming*, 2024. doi:10.4230/LIPIcs.ECOOP.2024.26.

20 Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. Exact separation logic. In *European Conference on Object-Oriented Programming*, 2023. doi:10.4230/LIPIcs.ECOOP.2023.19.

21 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: Real-world verification for JavaScript and C. In *Computer Aided Verification*, 2021. doi:10.1007/978-3-030-81688-9_38.

22 Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. From definitional interpreter to symbolic executor. In *International Workshop on Meta-Programming Techniques and Reflection*, 2019. doi:10.1145/3358502.3361269.

23 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, 2016. doi:10.1007/978-3-662-49122-5_2.

24 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, 2001. doi:10.1007/3-540-44802-0_1.

25 Srdja Panić. Collections-C: A library of generic data structures. https://github.com/srdja/Collections-C, 2014.

26 Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *European Symposium on Programming*, 2011. doi:10.1007/978-3-642-19718-5_23.

27 Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. A formal foundation for symbolic evaluation with merging. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3498709.

28 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification*, 2020. doi:10.1007/978-3-030-53291-8_14.

29 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, 2002. doi:10.1109/LICS.2002.1029817.

30 Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zürich, 2016.

31 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009. doi:10.1007/978-3-642-03013-0_8.

32 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Conference on Programming Language Design and Implementation*, 2014. doi:10.1145/2594291.2594340.

33 Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois Urbana-Champaign, 2001.

34 Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 2023. doi:10.1145/3586045.

35 Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. Sound gradual verification with symbolic execution. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024. doi:10.1145/3632927.