IMPERIAL

Aether
A Language Agnostic CSE

Background & Progress Report

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

July 17, 2024

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing (Software Engineering)

Contents

Chapter 1

Introduction

As software becomes part of critical element, it needs to be verified formally to ensure it does not fault. To support such verification in a scalable fashion, *separation logic* [1], [2] was created, permitting compositional proofs on the behaviour of programs. Research has also been done in the field of separation algebra, to provide an abstraction over the state modelled within separation logic, to improve modularity and reduce the effort needed for proofs.

Thanks to separation logic, compositional verification via specifications has been automated, and allows automatic checking of properties via compositional symbolic execution (CSE). A wide range of tools exist, usually enabling the verification of a specific language they are tailored for. Gillian [3]–[5] is a CSE engine that is different, in that it is not made for one specific language but is instead *parametric on the state model*, allowing for it to be used to verify different language, like C, JavaScript or Rust.

Gillian has now been in development for several years, and the research landscape surrounding it has evolved in parallel, with for instance new techniques for state modelling and the creation of incorrectness separation logic.

In this report we will present Aether, a CSE engine that follows the steps of Gillian and that is both parametric in the state model and in the language, while being closely related to the theory underpinning it. It's goal is to serve as a simple, efficient and complete engine that supports multiple analysis methods such as OX verification and UX true bug finding, and that is easily extendable for different uses.

Chapter 2

Literature Review

2.1 Separation Logic

As the need to formally verify programs grew, methods needed to be built to provide a framework to do so. Hoare Logic [6] is a logic made specifically to allow proving properties of programs, by describing them axiomatically. Every statement can be expressed as having a precondition – the state before execution – and a postcondition – the state after *successful* execution – expressed as $\{P\}\, S\, \{Q\}$. For assignment for instance, one could have $\{\mathsf{x} = x\}\, \mathsf{x} := 0\, \{\mathsf{x} = 0\}$.

While already extremely helpful to reason about programs, an issue remains however, and it is to do with shared mutable state, like memory of a program – how does one describe that there is a list in memory of unknown length, that may be mutated at multiple places in the program, while ensuring properties hold at a specific point in time? Being able to describe the state and constraints upheld by global state is difficult, and past solutions scaled poorly.

Separation Logic (SL) [1], [2] is an extension of Hoare logic that permits this in a clear and scalable way. It's main addition is the separating conjunction $*$: $P * Q$ means not only that the heap satisfies P and Q, but that it can be split into two *disjoint* parts, such that one satisfies P and the other Q. This allows us to reason compositionally about the state, by not only stating what properties are upheld, but how they may be split for further proofs. For instance, given a *list* predicate, when calling a function that mutates the list one can simply substitute the part of the state corresponding to that list with the postcondition of the function, with the guarantee that the rest of the state is untouched, by using the *frame rule*:

$$\textsc{Frame}\quad \frac{\{P\}\, S\, \{Q\}}{\{P * F\}\, S\, \{Q * F\}}$$

Because we know that S only uses (either by reading or modifying) P, any disjoint frame F can be added to the state without altering the execution of S. This is very powerful: one can prove properties of smaller parts of code (like a function, or a loop), and these properties will be able to be carried to a different context that may have a more complex state. For instance, this can be used in a Continuous Integration (CI) setting, by only analysing functions whose code is modified, while reusing past analysis of unchanged code, allowing for incremental analysis.

SL also comes equipped with the *emp* predicate, representing an empty state (ie. $P * emp = P$), and the separating implication $-\!\!*$ (also called wand). $P_0 -\!\!* P_1$ states that if the current state is extended with a disjoint part satisfying P_0, then P_1 holds in the extended heap [2].

Further predicates can then be defined; for instance the "points to" predicate, $a \mapsto x$, stating that the address a stores values x. We may note that $a \mapsto x * a \mapsto x$ does not hold, since a heap with a pointing to x cannot be split into two disjoint heaps both satisfying $a \mapsto x$. Similarly, $a \mapsto x * b \mapsto y$ can only hold if $a \neq b$.

2.1.1 Incorrectness Separation Logic

Separation Logic is, by definition, *over-approximate* (OX): $\{P\}\, S\, \{Q\}$ means that given a precondition P, we are guaranteed to reach a state that satisfies Q. In other words, Q may encompass *more* than only the states reachable from P. This can become an issue when used for real-world code, where errors that don't actually exist may be flagged as such, hindering the use of a bug detection tool (for instance in a CI setting).

To solve this problem, a recent innovation in the field is Incorrectness Separation Logic (ISL) [7], derived from Incorrectness Logic [8]. It is similar to SL, but *under-approximate* (UX), instead ensuring that the detected bugs actually exist.

Where SL uses triplets of the form $\{\text{precondition}\}\, S\, \{\text{postcondition}\}$, incorrectness separation logic uses $[\text{presumption}]\, S\, [\text{result}]$, with the result being an under-approximation of the actual result of the code. The reasoning is thus flipped, where we start from a stronger assertion at the end of the function, and then step back and broaden our assumptions, until reaching the initial presumption. This means that all paths explored this way are guaranteed to exist, but may not encompass all possible paths.

Another way of comparing SL to ISL is with consequence: in SL, the precondition implies

the postcondition, whereas with ISL the result implies the precondition. This enables us to do *true bug-finding*. Furthermore, ISL triplets are extended with an *outcome* ϵ, resulting in $[P]\, S\, [\epsilon : Q]$, where ϵ is the outcome of the function, for instance *ok*, or an error.

While SL is over-approximate and ISL is under-approximate, we may call *exact* (EX) specifications that are both OX-sound and UX-sound [9]. Such triplets are then written $\langle\!\langle P \rangle\!\rangle\, S\, \langle\!\langle o : Q \rangle\!\rangle$, with o the outcome, and P and Q the pre and postcondition respectively.

2.1.2 Separation Algebras

While the above examples of separation logic use a simple *abstract heap*, mapping locations to values, this is not sufficient to model most real models used by programming languages. For instance, the model used by the C language (in particular CompCert C [10], a verified C compiler) uses the notion of memory blocks, and offsets: memory isn't just an assortment of different cells. Furthermore, the basic "points to" predicate, while useful, has limited applications; for contexts such as in concurrency, one needs to have a more precise level of sharing that goes beyond the *exclusive ownership* of "points to".

An example of such extension is fractional permissions [11], [12]: the "points to" predicate is extended with a *permission*, a fraction q in the $(0; 1]$ range, written $a \underset{q}{\mapsto} x$. A permission of 1 gives read and write permission, while anything lower only gives read permission. This allows one to split permissions, for instance $a \underset{1}{\mapsto} x$ is equivalent to $a \underset{0.5}{\longmapsto} x * a \underset{0.5}{\longmapsto}$, a program can thus concurrently execute two routines that read the same part of the state, while remaining sound – permissions can then be re-added as the routines exit, regaining full permissions over the cell.

Further changes and improvements to separation logic have been made, usually with the aim of adapting a particular language feature or mechanism that couldn't be expressed otherwise. "The Next 700 Separation Logics" argues that indeed too many subtly different separation logics are being created to accommodate specific challenges, which in turn require new soundness proofs that aren't compatible with each other. To tackle this, research has emerged around the idea of providing one sound metatheory, that would provide the tools necessary to building more complex abstractions *within* that logic, rather than in parallel to it.

A first step in this quest towards a single core logic is the definition of an abstraction over the state. The main concept introduced for this is that of *Separation Algebras* [14], [15], which allow for the creation of complex state models from simpler elements. Because soundness is proven for these smaller elements, constructions made using them also carry this soundness, and alleviate users of complicated proofs.

Different authors used different definitions and axioms to define separation algebras.

In [14] they are initially defined as a cancellative partially commutative monoid (PCM) $(\Sigma, \bullet, 0)$. This definition is however too strong: for instance, the cancellativity property implies $\sigma \bullet \sigma' = \sigma \implies \sigma' = 0$. While this is true for some cases such as the points to predicate, this would invalidate useful constructions such as that of an *agreement*, where knowledge can be duplicated. For an agreement separation algebra, we thus have $\sigma \bullet \sigma = \sigma$, which of course dissatisfies cancellativity.

This approach is also taken in [15], where separation algebras are defined with a functional ternary relation $J(x, y, z)$ written $x \oplus y = z$ – the choice of using a relation rather than a partial function being due to the fact the authors wanted to construct their models in Rocq, which only supports computable total functions. While the distinction is mostly syntactical and both relational and functional approaches are equivalent, the former makes proof-work less practical [16], [17], whereas the functional approach allows one to reason equationally. This paper also introduces the notion of *multi-unit separation algebras*, separation algebras that don't have a single 0 unit, but rather enforce that every state has *a* unit, that can be different from other states' units. Formally, this means that instead of having $\exists u.\ \forall x.\ x \oplus u = x$, we have $\forall x.\ \exists u_x.\ x \oplus u_x = x$ – this allows one to properly define the disjoint union (or sum) of separation algebras, with both sides of the sum having a distinct unit. We may note that this is a first distinction from PCMs, as a partially commutative monoid cannot have two distinct units; according to the above definition, separation algebras are thus a type of partially commutative semigroup.

In [16], further improvements are done to the axiomatisation of separation algebras. Most notably, the property of cancellativity is removed, as it is too strong and unpractical for some cases, as shown previously. Furthermore, the idea of unit, or *core*, is made explicit, with the definition of the total function $\hat{\cdot}$, the core of a resource, which is its *duplicable part*.

Finally, Iris [17] is a state of the art "higher-order concurrent separation logic". It places itself as a solution to the problem mentioned in [13], and aims to provide a sound base logic that can be reused and extended to fit all needs. Its *resource algebras* (RAs), similar to separation algebras. Because Iris supports "higher-order ghost state", they need more sophisticated mechanisms to model state, and as such their composition function $\cdot$ needs to be total. To then rule out invalid compositions, they instead add a validity function $\overline{\mathcal{V}}$, which returns whether a given state is valid. They also keep the unit function, written $|\cdot|$, that they make *partial*, rather than total – this, they argue, makes constructing state models from smaller components easier[1]. This was confirmed when developing state models for Gillian, where having the core be a total function made some state models more

[1]The author of this report found it surprising that they removed partiality from composition to then re-add it via the core, as this lacks consistency

complicated to make sound.

2.2 Program Verification with Gillian

While having a logic to prove programs is a great step towards verifying codebases, doing it manually is tedious and time-consuming. Tools have been developed to automate this, and *Gillian* [3]–[5], the main inspiration of this project, is an example of such tool. It is a *compositional symbolic execution engine*, with the added property of being *parametric on the memory model*, and that allows reasoning about *correctness and incorrectness*. We may now go over exactly what this means.

2.2.1 Symbolic Execution

Traditionally, software is tested by calling the code with a predetermined or randomly generated input (for instance with fuzzing), and verifying that the output is as expected. These approaches, where the code is executed "directly", are of the realm of *concrete* execution, as the code is run with concrete – as in real, existing – values. While this method is straightforward to execute, it comes with flaws: it is limited by the imagination of the person responsible for writing the tests (when written manually), or by the probability of a given input to reach a specific part of the codebase. With fuzzing, methods exist to improve the odds of finding new paths [18], but it all amounts to luck nevertheless.

A solution to this is symbolic execution: rather than running the code with concrete values, *symbolic* values are used [19]. These values are abstract, and are then restricted as an interpreter steps through the code and conditionals are encountered. Once a branch of the code terminates, a constraint solver can be used against the accumulated constraints to obtain a possible concrete value, called an *interpretation*. This method appeared in the 70s, notably with the Select [20] and EFFIGY [21] systems.

For a simple C program that checks if a given number is positive or negative and even or odd (see Figure 2.1), symbolic execution would thus branch thrice, once at each condition[2]. This would then result in 5 different branches, each with different constraints on the program variable x: $\mathsf{x} = 0$, $\mathsf{x}\%2 = 0 \land \mathsf{x} < 0$, $\mathsf{x}\%2 = 0 \land \mathsf{x} > 0$, $\mathsf{x}\%2 \neq 0 \land \mathsf{x} < 0$ and $\mathsf{x}\%2 \neq 0 \land \mathsf{x} > 0$[3].

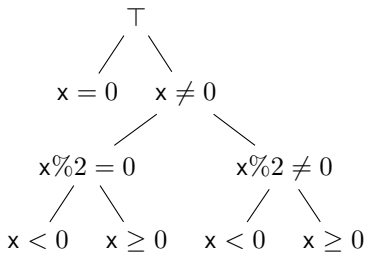
2.2.2 Compositional Symbolic Execution

An issue that arises from symbolic execution is that is scales quite poorly, because of path explosion [19], [22], as each branch can potentially multiply by two the total number of

[2]For simplicity, we omit the case where x is not an integer, which would lead to an error.
[3]The four last constraints also contain $\mathsf{x} = 0$, which is included in $\mathsf{x} > 0$ and $\mathsf{x} < 0$

```
if (x == 0) {
  print("zero");
  return;
}
if (x % 2 == 0) {
  print("even");
} else {
  print("odd");
}
if (x < 0) {
  print("negative");
} else {
  print("positive");
}
```

(a) Simple branching program



(b) Paths obtained from symbolically executing the code, with the added constraint at each node

Figure 2.1: Symbolic execution of a simple program

branches. Furthermore, it lacks compositionality, as the code must be tested in its entirety.

A solution to this is *compositional* symbolic execution (CSE), in which the code is tested compositionally, function by function. This allows for gradual adoption, as only parts of the code base can be tested, all while minimising the effect of path explosion.

A great tool to enable this is separation logic: one can specify the precondition and postcondition of a function, and the symbolic execution can then step through the code, by starting from a state satisfying the precondition and ensuring that once the branch terminates the postcondition is satisfied. Because of the frame rule, function specifications can then be re-used when calling functions, by symbolically executing the function specification on the corresponding state fragment. If the called function does not have a specification that can be used, it can instead be inlined, executing the code within the function, as would standard symbolic execution.

This approach was pioneered in Smallfoot [23], and was followed by later tools, such as JaVerT [24], [25], Infer [26], VeriFast [27], SpaceInvaders [28] and others – their individual contributions will be discussed in more detail in the next section.

2.2.3 Parametricity

An innovation of Gillian compared to other existing tools is the fact is is parametric on the memory model. Typical CSE engines usually come with a built-in notion of memory, ranging from simple heaps with memory chunks [27] to more complex state models tailored to a language, like in JaVerT [24], [25] to represent JavaScript objects.

While a fixed memory model is useful for specific languages or feature sets, this can become a limitation when porting the engine to other languages. The goal behind Gillian's parametricity is to enable any state model to be represented, enabling it to be used for virtually any language (of course as long as the language's memory model can be encoded

into a separation-logic equivalent). This enables it to verify programs written in JavaScript, C or Rust, despite their significant differences in terms of state representation and level of abstraction.

To define a memory model to use in Gillian, a developer must implement it in OCaml, defining a specific interface. A memory model is a partially commutative monoid, comprised of a set of states, which can be modified via *actions* and be represented in specifications via *core predicates*.

Actions serve as the interface for programs to interact with the state, for instance by storing or loading values into the state, or allocating a new block of memory and then freeing it. As such, while the language Gillian targets, GIL, provides the base semantics of an imperative goto-language (variables, errors, functions...) which do not affect the state – actions are the only way of manipulating it.

Core predicate are predicates which have in and out parameters, and represent fragments of the state. They are the encoding of the state into separation logic, and can be used within function specifications. For example, the separation logic $a \mapsto x$ predicate could be represented as a core predicate as $\langle \texttt{points\_to} \rangle (\texttt{a}; \texttt{x})$.

As mentioned in subsection 2.1.2, literature already exists to enable sound modelling of state, and Iris [17] is an example of the state of the art in this field, with a rich theory and existing automation in Rocq. However, this had never been ported to automatic program verification, instead remaining in the meta-theory.

2.2.4 Correctness, Incorrectness

Another key difference between Gillian and other CSE engines is that is allows reasoning about programs both with correctness, in separation logic [1], [2] and incorrectness separation logic [7], thus allowing OX program verification and UX true bug-finding respectively. It also supports whole-program testing, as a non-compositional engine.

This allows one to instantiate Gillian to their preferred target language, and automatically benefit from all three modes of symbolic execution. This is in part possible thanks to the exact (EX) [9] semantics of GIL, enabling automated proofs to be done both with left-to-right and right-to-left implications, representing OX and UX respectively.

Thanks to this and the aforementioned parametricity of the memory model, Gillian is thus a unified CSE engine, that allows scalable verification of software in a unified setting, without requiring one to adapt to two different engines, which may be prohibitive and extremely time-consuming.

2.3 Existing Tools

A wide range of engines exist to verify the correctness of code, most targeted towards a specific language. This allows them to implement behaviour that is appropriate for that particular language. For instance, CBMC [29] is a bounded model checker, that allows for the verification of C programs, via whole program testing – that is, it is not compositional, and instead symbolically executes a given codebase, and verifies that properties hold. While clearly useful, as shown by its success, this solution doesn't scale particularly well; larger applications need to be split modularly to be verified, and manual stubbing is required to support testing parts of the code separately.

Slightly separate from symbolic execution, KLEE [30] allows for the *concolic* execution of LLVM code – thus supporting C, C++, Rust, and any other language that can be compiled to LLVM. Concolic execution is a hybrid of symbolic and concrete execution, allowing the use of symbolic values along a concrete execution path. Similarly to CBMC, it handles whole program testing, and as such doesn't scale with larger codebases. MACKE [31] is an adaptation of KLEE enabling compositional testing, proving that concolic execution can be extended to be more scalable, without losing accuracy in reported errors.

As codebases grow, there comes a need for compositional tools that can verify smaller parts of the code, thus enable greater scaling and integration in continuous integration (CI) pipelines, which permit giving rapid feedback to developers. Separation logic enables this, by allowing one to reason about a specific function and ignoring the rest of the program's state.

While Smallfoot [23] was the first tool to enable compositional symbolic execution, jStar [32] was the first to allow automated verification of a real-world language, Java. It is in particular tailored towards handling design patterns that are commonly found in object-oriented programming and that may be hard to reason about. It uses an intermediate representation of Java called Jimple, taken from the Java optimisation framework Soot. Similarly, VeriFast [27] is a CSE tool that is targeted at Java and a subset of C, while SpaceInvader [28] allows for the verification of C code in device drivers.

Infer [26] is a tool developed by Meta that uses separation logic and bi-abduction to find bugs in C, C++, Java and Objective-C programs, by attempting to prove correctness and extracting bugs from failures in the proof. It is the tool that pioneered *bi-abduction*, a technique enabling execution to proceed despite resources missing, via the construction of an anti-frame. It's approach, while initially stemming from separation logic, was used for finding bugs rather than proving correctness, which led to the creation of ISL [7] to provide a sound theory justifying this approach. Following this, Pulse [33] followed and

fully exploited the advances made in ISL to show that this new theory served as more than a justification to past tools, and to prove the existing of true bugs.

Also using separation logic, JaVerT [24], [25] is a CSE engine aimed to verifying JavaScript, with support for whole program testing, verification, and bi-abduction in the same fashion as Infer – it followed from Cosette [34], which already supported whole program testing.

While the above examples are targeted at specific target languages, some tools have also been designed with modularity in mind. To do such, they instead support a general intermediate language – one that isn't designed specifically for a language, like Jimple for Java – that a target language must be compiled to. This allows for greater flexibility in regards to what languages can be analysed by the engine, as all it needs to be capable of verifying a new language is a compiler.

Another example is Viper [35], an infrastructure capable of program verification with separation logic, with existing frontends that allow for the verification of Scala, Java and OpenCL. It's intermediate language is a rich object-oriented imperative typed language, that operates on a built-in heap.

One can still take genericity further, by also abstracting the state model the engine operations on. A tool that does this is coreStar [36], the CSE engine backing jStar with all the Java-related parts removed and replaced. It does not target Java, but instead a generic intermediate language. Furthermore, instead of coming with predicates pre-defined, it requires user-defined predicates to be provided, which are then used throughout the proof. They are defined in a language specified by coreStar, and are thus however quite limited in expressivity. Still, this is to the author's knowledge the first example of a tool that allows some flexibility in the underlying logic theory.

This is also what Gillian [3]–[5] does, a CSE engine parametric on the state model, and the main work upon which this project this project is based. It doesn't support any one language, but instead can target any language that provides a state model implementation in OCaml and a compiler to GIL, it's intermediary language. Currently Gillian has been instantiated to the C language (via CompCert-C [10]) and JavaScript, as well as Rust [37]. It is a generalisation of JaVerT, which only targeted JavaScript.

	Target Language	Compositional	Parametric State Model	Mode
CBMC	C	✗	✗	WPST
KLEE	LLVM	✗	✗	Concolic
MACKE	LLVM	✓	✗	Concolic
jStar*	Java	✓	✗	OX
VeriFast	C, Java	✓	✗	OX
coreStar*	IR (coreStarIL)	✓	✓	OX
Infer	C, C++, Java, Objective-C	✓	✗	OX
Cosette*	JavaScript	✓	✗	WPST
JaVerT	JavaScript	✓	✗	WPST, OX
Viper	IR (Viper)	✓	✗	OX
Pulse	C, C++, Java, Objective-C	✓	✗	UX
Gillian	IR (GIL)	✓	✓	WPST, OX, UX

WPST = Whole Program Symbolic Testing.

*Unmaintained

Table 2.1: Comparison of verification tools

Chapter 3

Current Work

3.1 Initial Thoughts

Two separate groups of work stemmed from Gillian [3]–[5] and are in the process of being written or published: CSE1 [38] and CSE2, and the thesis of Sacha-Élie Ayoun [39]. CSE1 is a formalisation of a unified compositional symbolic execution engine, aimed at providing a sound and generalisable theory for a CSE engine. CSE2 then extends it by making the engine parametric on the memory model, while improving and mechanising the theory. Sacha's thesis on the other hand is the complete theory behind the Gillian framework, as well as that of the instantiation of Gillian to C and Rust. Both CSE2 and Sacha's thesis are thus about a CSE parametric on the memory model.

As these latter two works are being developed in parallel, they have taken different approaches for several parts of their respective engine – part of the current project is evaluating these decisions, and deciding which is the most appropriate. We may also consider outside works that contribute to these questions.

3.1.1 State models: PCMs or RAs

Gillian represents its state model as a partially commutative monoid (PCM). As discussed previously, this comes with certain limitations – most notably, the existence of a single 0 element, whereas other instances of separation algebras in the literature (such as the Resource Algebras (RAs) in Iris [17]) permit several 0s (via the unit function), as well as the absence of a unit.

This is in particular useful for some particular state models, such as the sum, written $A \oplus B$, with $A \stackrel{\text{def}}{=} (A, 0_A, \cdot)$ and $B \stackrel{\text{def}}{=} (B, 0_B, \cdot)$ two state models. Initial attempts at defining a sum model in Sacha's thesis modelled it as $A \oplus B \stackrel{\text{def}}{=} 0_\oplus \mid \mathtt{L}\ A \mid \mathtt{R}\ B$, with the composition defined as:

$$0_\oplus \cdot \sigma = \sigma \cdot 0_\oplus = \sigma$$

$$\texttt{L}\ \sigma \cdot \texttt{L}\ \sigma' = \texttt{L}\ (\sigma \cdot \sigma')$$

$$\texttt{R}\ \sigma \cdot \texttt{R}\ \sigma' = \texttt{R}\ (\sigma \cdot \sigma')$$

In particular this poses a problem when one wants to be able to *switch* from one side of the sum to the other, as the existence of the states $\texttt{L}\ 0_A$ and $\texttt{R}\ 0_B$ is not frame preserving, requiring the sum to be rewritten as $0_\oplus\ \texttt{|}\ \texttt{L}\ A\backslash\{0_A\}\ \texttt{|}\ \texttt{R}\ B\backslash\{0_B\}$ hinting at something being wrong about the 0s.

Similarly, a frame-preserving switch requires that either the state switched from or switched to (in OX and UX respectively) be *exclusively owned*. This can be defined as $\texttt{is\_exclusively\_owned}\ \sigma \stackrel{\text{def}}{=} \forall \sigma'.\ \sigma' \neq 0 \implies \neg(\sigma \# \sigma')$, where $\#$ signifies that both states are disjoint and can be composed. While this definition works, having a core function would make it more elegant, and easier to prove for a given state; Iris defines this same predicate as $\text{exclusive}(a) \stackrel{\text{def}}{=} \forall c. \neg\overline{\mathcal{V}}(a \cdot c)$, allowing them to then prove sides of a sum can be switched from one to the other via the following rule:

$$\textsc{Exclusive-Update}\quad \frac{\text{exclusive}(a) \qquad \overline{\mathcal{V}}(b)}{a \rightsquigarrow b}$$

The insight from this is that again, the requirement that a PCM must have a unique 0 element is too strong – it then requires one to remove this 0 in higher state model transformations, which is both inelegant and tricky to notice.

Another argument in favour of RAs is the strength of the Iris theory, that provides flexible constructs and has been successfully reused several times. As such, creating an engine that uses Iris' RAs for its state models allows one to benefit for free from the existing infrastructure surrounding it, and in particular its Rocq implementation.

Iris being extremely sophisticated and supporting complex "higher-order ghost state", their composition needs to be total, and accompanied by a validity function $\overline{\mathcal{V}}$. Because the CSE engine developed for this project will not require such sophistication, we will make the choice of diverging from Iris on the totality of composition, instead keeping it partial. And because the validity function was used to rule out compositions that should not be allowed, we may also get rid of it, and instead rely on the fact an invalid composition is simply not defined. This comes with the pleasant side-effect that "error states" need not be defined

for state models, as is the case with Iris where they must always define a fallback $\lightning$ state to be used for invalid composition, for instance as it is the case for the exclusive state model:

$$\mathrm{Ex}(X) \stackrel{\text{def}}{=} \mathrm{ex}(x : X) \mid \lightning$$
$$\overline{\mathcal{V}}(a) \stackrel{\text{def}}{=} a \neq \lightning$$
$$|ex(x)| \stackrel{\text{def}}{=} \bot$$

The above parts marked in red could thus be ignored by making composition partial; instead of always being $\lightning$, it always is undefined. One can go from an Iris-like RA to this new type of partial RA with a simple equivalence: $\overline{\mathcal{V}}(\sigma \cdot \sigma')$ is true in the former if and only if $\sigma \cdot \sigma'$ is defined in the latter (this can be further proven using Iris' rule Ra-Valid-Op, which states $\forall a, b.\ \overline{\mathcal{V}}(a \cdot b) \implies \overline{\mathcal{V}}(a)$).

While further work could be about attempting to port Iris' higher-order ghost states to this framework, this will be left out of scope for now, noting that the current divergence from Iris is minimal and easy to revert with the above equivalence.

It is also worth noting that Iris enforces that the *global RA* (so the final RA, composed of smaller elements) must be unital, ie. that it has a unique unit element. This is equivalent to the unique 0 element of PCMs – as such, the global state model of the engine will remain very similar to a PCM; the outlined distinctions (multiple or no units, a core function) are relevant mostly for the construction of the global state model. Once constructed, it can be trivially turned into a unital RA by extending it with a unit.

3.1.2 Well-formedness: Implicit or Explicit

Another distinction between the two works is their handling of well-formedness. The CSE1 paper *explicitly* defines well-formedness $\mathcal{W}f$, defined for a state $(\hat{s}, \hat{h}, \hat{\pi})$ comprised of a symbolic store, symbolic heap and path condition as:

$$\hat{\pi}_1 \vDash \hat{\pi}_2 \stackrel{\text{def}}{=} \forall \epsilon.\ \epsilon(\hat{\pi}_1) = \text{true} \implies \epsilon(\hat{\pi}_2) = \text{true}$$
$$\mathcal{W}fc(\hat{s}) \stackrel{\text{def}}{=} \mathrm{codom}(\hat{s}) \subseteq \mathrm{Val}$$
$$\mathcal{W}fc(\hat{h}) \stackrel{\text{def}}{=} \mathrm{dom}(\hat{h}) \subseteq \mathrm{Nat} \wedge \mathrm{codom}(\hat{h}) \subseteq \mathrm{Val} \wedge (\forall \hat{v}_i, \hat{v}_j \in \mathrm{dom}(\hat{h}).\ i \neq j \implies \hat{v}_i \neq \hat{v}_j)$$
$$\mathcal{W}f((\hat{s}, \hat{h}, \hat{\pi})) \stackrel{\text{def}}{=} \mathsf{SAT}(\hat{\pi}) \wedge (\mathrm{sv}(\hat{s}) \cup \mathrm{sv}(\hat{h}) \subseteq \mathsf{sv}(\hat{\pi})) \wedge \hat{\pi} \vDash (\mathcal{W}fc(\hat{s}) \wedge \mathcal{W}fc(\hat{h}))$$

It then enforces in the axiomatic interface of `consume` and `produce` for well-formedness of the resulting state to be preserved. One may also note that, quoting CSE1, "we only work with well-formed states $\hat{\sigma}$, denoted $\mathcal{W}f(\hat{\sigma})$", making it similar to Iris' approach to

validity with $\overline{\mathcal{V}}$.

This approach is different in CSE2, where well-formedness is part of the definition of the models relation $\vDash$ itself: a concrete state (θ, s, h) models a symbolic state $(\hat{s}, \hat{h}, \hat{\mathcal{P}}, \hat{\pi})$ if there exists an interpretation of the symbolic state resulting in the concrete state. Because well-formedness is part of this relation, one remains in separation logic longer, as conversion to first order logic is done only when the SMT check is needed and not before.

In comparison, Sacha's thesis does not explicitly enforce such well-formedness, and instead keeps it *implicit*, by assuming a reached is already valid and well-formed, as enforced by the path condition – a state that is not well-formed would have already raised an error, and would not have continued. Well-formedness is then kept throughout execution, as soundness is preserved.

One of the core distinctions between the two approaches is thus at what point throughout execution is a state encoded into a first order logic formula and sent to an SMT solver; whereas CSE chooses to stay in separation logic longer and only encode the constraints to first order logic later for the SMT check, Sacha's thesis instead keeps accumulating the constraints in $\hat{\pi}$ directly, and then doesn't require further encoding into first order logic.

Both alternatives are mostly analogous, and there both seem to be equivalent – the difference may amount to performance, which would need to be measured. To remain aligned with the decision to remove the validity function of RAs however, the implicit alternative to well-formedness feels at present more justified.

3.1.3 Symbolic and Logical Variables: United or Separate

Another consideration to take into account is the distinction between *symbolic variables* and *logical variables*.

Symbolic variables arise from symbolic execution: program variables are mapped to symbolic values via a symbolic store $\hat{s} : \text{PVar} \xrightarrow{fin} \text{SVal}$, and a symbolic variable can be *interpreted* into a set of concrete values with $\epsilon : \text{SVar} \xrightarrow{fin} \text{Val}$ [38]. They are the symbolic counterpart to concrete variables.

Logical variables, on the other hand, are the classic variables one would encounter when reasoning about a program; they are existentially quantified. A substitution $\hat{\theta} : \text{LVar} \xrightarrow{fin} \text{SVal}$ maps logical variables to symbolic values, whereas a logical interpretation $\theta : \text{LVar} \xrightarrow{fin} \text{Val}$ maps logical variables to concrete values [38]. One can see logical variables as a syntactic construct to designate symbolic values. For instance, the in and out values of predicates are logical variables.

While they represent separate concepts, CSE2 has taken the approach of treating them as the same and having no distinction between the two, treating logical variables as symbolic

variables. A key motivation for this is that it avoids having to carry a substitution $\hat{\theta}$ into symbolic values, making the theory less verbose.

On the other hand, one could argue that they represent separate worlds and should not be mixed. Symbolic variables live in the symbolic state, and should not be "forgotten" or lost throughout execution, whereas logic variables are merely syntactic and can be renamed as needed. Furthermore, during matching the distinction between logic and symbolic variables is useful to easily indicate which variables can and cannot be changed – mixing both makes this less clear, and more prone to error.

3.2 Work So Far

We may now present some of the work that has been done so far, which may serve during the development of this project.

3.2.1 State Models

Following the PhD thesis of Sacha-Élie Ayoun [39], several state models and state model transformers have been implemented for the current version of Gillian. We distinguish state models, which are atoms of the construction of a larger state model, and state model *transformers*, which are functors that receive one or more state models (or other inputs), and create a new state model from these.

The full list of implemented state models is:

- `Exc`: the exclusive state model, which implements `load` and `store` actions and a single `points_to` predicate. This corresponds to the "default" points to predicate of separation logic, but without the address, which is added later via a transformer.
- `Ag`: the agreement state model, which implements a `load` action and a single `agree` predicate. This predicate is duplicable, and can be used to model the concept of shareable *knowledge*, which is immutable – this notion. has been studied several times in the literature [16], [17], [40], [41]. Note that it does not have a `store` action, as it would not be frame-preserving.
- `Frac`: the fractional state model, which similarly to `Exc` implements the `load` and `store` actions, but instead adds a `frac` predicate which is equivalent to the points to predicate equipped with a fraction, as discussed previously and taking inspiration from [11], [12].

To allow extending simpler state models, several state model transformers have also been implemented. These may extend their components by implementing additional ac-

tions, modifying the parameters or return values of actions, adding in or out parameters to predicates, or extending the possible states with additional information. The implemented transformers are:

- `PMap`: a partial map transformer, that allows mapping locations to states. It extends all predicates with an extra in value, the index, and similarly with the parameters of actions. It also allows for two different modes of operation. The first is *static* mode, which enables C-like behaviour, where new cells must be allocated via an `alloc` action, that returns a fresh location. The second is *dynamic* mode, which is what one would expect to see for JavaScript: the map is initially empty, but trying to write or load from anywhere in the map will simply add the index to the map if not already allocated – in this mode, the `alloc` action is thus forbidden.

- `List`: a list transformer, that maps integers between 0 and a given length to states. It is similar to `PMap`, but with the added guarantee of having all indices fall in a range. It is initialised with empty elements in all cells, and as such does not require an `alloc` action. It does however extend all actions and predicates with an "offset" parameter and in value respectively.

- `Freeable`: this transformer allows one to *free* a resource, that is to delete it and remember the cell was freed. It thus extends the states with a freed state $\varnothing$, and adds an action `free` that frees the state. The need for a $\varnothing$ state to mark a location as freed is needed for UX-soundness, and is one of the key insights of Incorrectness logic[8], as simply "forgetting" about a resource when freeing it would not be frame preserving in a UX setting. This transformer is similar to Iris' `OneShot`, except the $\varnothing$ state is *not duplicable*, which is also needed for UX-soundness – it is instead exclusively owned.

- `Product`: this represents the product of two state models. It doesn't add any actions or predicates, but modifies all predicate and action names by appending a prefix to them, to then be able to distinguish to which side of the product the action or predicate is destined.

Because Gillian has significantly grown in complexity over time, the interface of state models `MonadicSMemory` is quite complex and contains several required functions that have an unclear definition. To ease development of the above state models and transformers, a simpler interface `MyMonadicSMemory` was implemented, which adds defaults to some of these functions and allows some simplifications of the interface. While not extensively tested, these simplifications do not seem to be the source of any problem, at least in OX-mode.

3.2.2 Gillian Experiment: Outcomes

One of the insights and advances made in Sacha-Élie's thesis is the addition of an `LFail` error, to distinguish missing resource errors from logical failure due to a logical mismatch or consumer incompleteness. This means the possible outcomes for a symbolic execution branch are $\mathcal{O}_l^+ = \{\texttt{Ok}, \texttt{Err}, \texttt{Miss}, \texttt{LFail}\}$.

The current version of Gillian uses the `result` type for symbolic branches, defined as `('a, 'e) result = | Ok of 'a | Error of 'e`. To then distinguish between fixable `Miss` errors and other errors, state models must then implement a `can_fix` function that tells whether an error is fixable, which is to say whether it is a `Miss`. This is a significant difference between the implementation and the theory behind Gillian.

Redefining a custom state model interface wrapping Gillian's own interface allowed the author to easily experiment with core changes to the typings and interfaces, as these changes could then be converted back to a Gillian-compatible format within the wrapper, while giving access to the improvements to the state models. As such, an experiment was made to extend errors with a `Miss` and `LFail` outcomes, as seen in Figure 3.1. This then allowed the `can_fix` function to be removed, as a simple pattern match for `Miss` is sufficient to know if an outcome is fixable.

```
type 'a 'e 'f 'm sym_result =
| Ok of 'a
| Err of 'e
| LFail of 'f
| Miss of 'm
```

Figure 3.1: Symbolic result type

The outcome of the experiment proved quite promising – although it sometimes made handling symbolic branches more verbose (as more outcomes needed to be checked or wrapped), we consider the gain in parity with the theory and the expressiveness of the outcomes more than enough to make up for it. This change allowed to further simplify the implementation of all state models and transformers, at no extra cost. While this experiment was limited to the scope of state models, it may be of interest in future research and the current project to more deeply use this system, in particular in the core of the engine, as it could probably allow for further simplifications.

3.2.3 Sum Soundness

An unexpected side-effect of implementing the aforementioned state model and state model transformers is that it helped uncover unsoundness in a state model transformer: the sum. As explained in subsection 3.1.1, the initial definition and implementation of the sum was

both OX and UX unsound. Research was thus done to prove properties that must hold for a sum that can switch sides – this is for instance the case of `Freeable`, which while implemented as a standalone state model, could be represented as $X \oplus \Sigma_{\texttt{Freed}}$ with X the input state model, and $\Sigma_{\texttt{Freed}}$ a trivial monoid with a single `Freed` element and exclusive ownership.

For brevity, we will simply list the properties for a OX and UX-sound "swap" action that flips the sum from one side to the other. First, neither sides of the sum must allow a 0 element. Furthermore, both the states swapped from and to must be *exclusively owned*. This rules out some state models as valid elements of such sums, for instance `Ag`.

Chapter 4

Project Plan

During the duration of this project, the author will aim to implement a Compositional Symbolic Engine, that is both parametric in the memory model in the style of Gillian [3]–[5], and parametric on the language semantics.

An initial implementation will first be done, by following the progress made by unpublished papers of the group, CSE1 [38] and CSE2, as well as the upcoming PhD dissertation of Sacha-Élie Ayoun [39] – see section 3.1 for some key differences. Further refinements will be brought, from additions from the author as well as existing research, most notably in the field of separation algebras, attempting to line up with progress done by Iris [17]. This first version will feature OX and UX verification.

The initial version will also attempt to have the language semantics as separate from the rest of the engine as possible, while implementing them for a simplified version of the GIL intermediate language as seen in Gillian.

Along with this, a theory will be made to justify the choices made in the engine and proving its soudness. Some of these decisions include, as seen in section 3.1, deciding what form will state models take, where will well-formedness by formulated, how symbolic variables will be handled and matched, etc.

In further development, the built-in language semantics will be stripped out (or made optional), in favour of an interface to specify language semantics externally, and proving soundness of the approach by formulating axioms that must hold for the analyses to be sound.

In parallel, a toolkit of separation algebras constructs will be developed, to allow new developers to easily construct complex state models from simpler elements. These will be ported from the pre-existing work presented in subsection 3.2.1.

This project plan is structured in a way that if due to time constraints further steps cannot be completed, the existing version will remain functional and will still provide in-

novation. Some notable decisions that are yet to be taken, and that will distinguish

Some of the innovations of the project include:

- Making a CSE engine that uses Iris-like state models
- Providing the first (to the knowledge of the author) CSE engine that is parametric on the semantics of the target language – existing CSE engines usually have one target language or intermediary language.
- Providing axioms and proof for the soundness of parametric language semantics.

Finally, the project will be evaluated by comparing it's execution time in comparison to the existing Gillian implementation, as well as by trying to verify existing real-life code with it.

Bibliography

[1] P. O'Hearn, J. Reynolds, and H. Yang, "Local Reasoning about Programs that Alter Data Structures," in *Computer Science Logic*, L. Fribourg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–19, ISBN: 978-3-540-44802-0.

[2] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[3] J. F. Santos, P. Maksimović, S.-É. Ayoun, and P. Gardner, *Gillian: Compositional Symbolic Execution for All*, 2020. arXiv: 2001.05059 [cs.PL].

[4] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner, "Gillian, Part I: A Multi-Language Platform for Symbolic Execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 927–942, ISBN: 9781450376136. DOI: 10.1145/3385412.3386014. [Online]. Available: https://doi.org/10.1145/3385412.3386014.

[5] P. Maksimović, S.-E. Ayoun, J. F. Santos, and P. Gardner, "Gillian, Part II: Real-World Verification for JavaScript and C," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*, Berlin, Heidelberg: Springer-Verlag, 2021, pp. 827–850, ISBN: 978-3-030-81687-2. DOI: 10.1007/978-3-030-81688-9_38. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_38.

[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[7] A. Raad, J. Berdine, H.-H. Dang, D. Dreyer, P. O'Hearn, and J. Villard, "Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds., Cham: Springer International Publishing, 2020, pp. 225–252, ISBN: 978-3-030-53291-8.

[8] P. W. O'Hearn, "Incorrectness logic," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, 2019. DOI: 10.1145/3371078. [Online]. Available: https://doi.org/10.1145/3371078.

[9] P. Maksimović, C. Cronjäger, A. Lööw, J. Sutherland, and P. Gardner, "Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding," en, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: 10.4230/LIPICS.ECOOP.2023.19. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.19.

[10] X. Leroy, A. Appel, S. Blazy, and G. Stewart, "The CompCert Memory Model, Version 2," Jun. 2012.

[11] R. Bornat, C. Calcagno, P. Hearn, and H. Yang, "Fractional and counting permissions in separation logic,"

[12] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *SIGPLAN Not.*, vol. 40, no. 1, pp. 259–270, 2005, ISSN: 0362-1340. DOI: 10.1145/1047659.1040327. [Online]. Available: https://doi.org/10.1145/1047659.1040327.

[13] M. J. Parkinson, "The next 700 separation logics," in *2010 Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, vol. 6217, Springer Berlin / Heidelberg, 2010, pp. 169–182. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-next-700-separation-logics/.

[14] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local Action and Abstract Separation Logic," in *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, 2007, pp. 366–378. DOI: 10.1109/LICS.2007.30.

[15] R. Dockins, A. Hobor, and A. W. Appel, "A Fresh Look at Separation Algebras and Share Accounting," in *Programming Languages and Systems*, Z. Hu, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 161–177, ISBN: 978-3-642-10672-9.

[16] F. Pottier, "Syntactic soundness proof of a type-and-capability system with hidden state," *Journal of Functional Programming*, vol. 23, no. 1, pp. 38–144, 2013. DOI: 10.1017/S0956796812000366.

[17] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, e20, 2018. DOI: 10.1017/S0956796818000151.

[18] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding Software Vulnerabilities by Smart Fuzzing," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 427–430. DOI: 10.1109/ICST.2011.48.

[19] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[20] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software*, Los Angeles, California: Association for Computing Machinery, 1975, pp. 234–245, ISBN: 9781450373852. DOI: 10.1145/800027.808445. [Online]. Available: https://doi.org/10.1145/800027.808445.

[21] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[22] S. Anand, P. Godefroid, and N. Tillmann, "Demand-Driven Compositional Symbolic Execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381, ISBN: 978-3-540-78800-3.

[23] J. Berdine, C. Calcagno, and P. O'Hearn, "Modular automatic assertion checking with separation logic," vol. 4111, Nov. 2005, pp. 115–137, ISBN: 978-3-540-36749-9. DOI: 10.1007/11804192_6.

[24] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienundefined, T. Wood, and P. Gardner, "JaVerT: JavaScript verification toolchain," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 2017. DOI: 10.1145/3158138. [Online]. Available: https://doi.org/10.1145/3158138.

[25] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "JaVerT 2.0: compositional symbolic execution for JavaScript," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019. DOI: 10.1145/3290379. [Online]. Available: https://doi.org/10.1145/3290379.

[26] C. Calcagno and D. Distefano, "Infer: an automatic program verifier for memory safety of C programs," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11, Pasadena, CA: Springer-Verlag, 2011, pp. 459–465, ISBN: 9783642203978.

[27] B. Jacobs, J. Smans, and F. Piessens, "A Quick Tour of the VeriFast Program Verifier," vol. 6461, Nov. 2010, pp. 304–311, ISBN: 978-3-642-17163-5. DOI: 10.1007/978-3-642-17164-2_21.

[28] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn, "Scalable shape analysis for systems code," Jul. 2008, pp. 385–398, ISBN: 978-3-540-70543-7. DOI: 10.1007/978-3-540-70545-1_36.

[29] D. Kroening, P. Schrammel, and M. Tautschnig, *CBMC: The C Bounded Model Checker*, 2014. arXiv: 2302.02384 [cs.SE].

[30] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[31] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "Macke: Compositional analysis of low-level vulnerabilities with symbolic execution," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 780–785, ISBN: 9781450338455. DOI: 10.1145/2970276.2970281. [Online]. Available: https://doi.org/10.1145/2970276.2970281.

[32] D. Distefano and M. J. Parkinson J, "jStar: towards practical verification for java," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '08, Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 213–226, ISBN: 9781605582153. DOI: 10.1145/1449764.1449782. [Online]. Available: https://doi.org/10.1145/1449764.1449782.

[33] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O'Hearn, "Finding real bugs in big programs with incorrectness logic," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022. DOI: 10.1145/3527325. [Online]. Available: https://doi.org/10.1145/3527325.

[34] J. Fragoso Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic Execution for JavaScript," Sep. 2018, pp. 1–14. DOI: 10.1145/3236950.3236956.

[35] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62, ISBN: 978-3-662-49122-5.

[36] M. Botincan, D. Distefano, M. Dodds, R. Grigore, D. E, and M. Parkinson, "Corestar: The core of jstar," May 2012.

[37] S.-É. Ayoun, X. Denis, P. Maksimović, and P. Gardner, *A hybrid approach to semi-automated Rust verification*, 2024. arXiv: 2403.15122 [cs.PL].

[38] A. Lööw, D. Nantes Sobrinho, S.-E. Ayoun, C. Cronjäger, P. Maksimović, and P. Gardner, "Compositional Symbolic Execution for Correctness and Incorrectness Reasoning."

[39] S.-E. Ayoun, "Gillian: Foundations, Implementation and Applications of Compositional Symbolic Execution."

[40] A. Bizjak and L. Birkedal, "On Models of Higher-Order Separation Logic," *Electronic Notes in Theoretical Computer Science*, vol. 336, pp. 57–78, 2018, The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2018.03.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066118300197.

[41] A. Pilkiewicz and F. Pottier, "The essence of monotonic state," in *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, ser. TLDI '11, Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 73–86, ISBN: 9781450304849. DOI: 10.1145/1929553.1929565. [Online]. Available: https://doi.org/10.1145/1929553.1929565.