IMPERIAL

Aether
A Language Agnostic CSE

Background & Progress Report

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

June 3, 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 assignement for instance, one could have $\{\mathtt{x} = x\}\, \mathtt{x} := 0\, \{\mathtt{x} = 0\}$.

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*:

FRAME

$$\frac{\{P\}\, S\, \{Q\}}{\{P * F\}\, S\, \{Q * F\}}$$

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 Bi-Abduction

Separation Logic is powerful in that a develop can define a precondition and postcondition of a function, and automations can then verify that these assertions hold. However, writing such specification can be tedious, in particular for large pieces of code where there is a significant amount of state in the pre/post-condition. *Bi-Abduction* [7] is a method that introduces the concept of *anti-frame*, an inverse to the usual frame, which represents missing parts of the state that were encountered during symbolic execution. This allows the execution to carry on, by taking into account the generated anti-frame, which is then signalled to the user. More formally, the question bi-abduction answers is:

$$P * A \vdash Q$$

In other words, given a precondition P what is the antiframe A that is needed to reach the postcondition Q. While formally developped in [7], Infer [8] was the real-world tool to bring the idea to production.

2.1.2 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) [9], derived from Incorrectness Logic [10]. It is similar to SL, but *under-approximate* (UX), instead ensuring that the detected bugs actually exist.

Where SL uses triplets of the form {precondition} S {postcondition}, incorrectness separation logic uses [presumption] S [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 [11]. 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.

2.1.3 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 [12], 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 [13], [14]: the "points to" predicate is extended with a *permission*, a fraction q in the $(0; 1]$ range, written $a \overset{q}{\mapsto} x$. A permission of 1 gives read and write permission, while anything lower only gives read permission. This allows one to split permissions, for instance $a \overset{1}{\mapsto} x$ is equivalent to $a \overset{0.5}{\longmapsto} x * a \overset{0.5}{\longmapsto}$, 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 accomodate specific challenges, which in turn require new soundness proofs that aren't compatible with eachother. 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* [16], [17], which allow for the creation of complex state models from simpler elements. Because soudness is proven for these smaller elements, constructions made using them also carry this soundness, and aleviate users of complicated proofs.

Different authors used different definitions and axioms to define separation algebras. In [16] 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 [17], 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 [18], [19], 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 definiton, separation

algebras are thus a type of partially commutative semigroup.

In [18], 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 explicity, with the definition of the total function $\hat{\cdot}$, the core of a resource, which is its *duplicable part*.

Finally, Iris [19] is a state of the art "higher-order concurrent separation logic". It places itself as a solution to the problem mentionned in [15], 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 their separation logic is aimed at automation via Rocq, they need the composition function $\cdot$ to be total. As such, instead of defining invalid composition via partiality (by marking an invalid composition of states as undefined), 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 developping state models for Gillian, where having the core be a total function made some state models more complicated to make sound.

2.2 Program Verification

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 [20], 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 [21]. 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*.

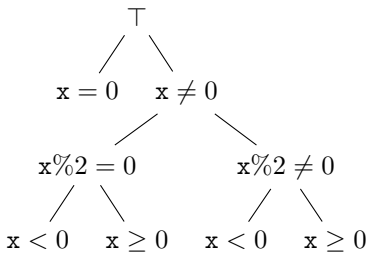
For a simple C program that checks if a given number is positive or negative and even or

[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

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: $x = 0$, $x\%2 = 0 \wedge x < 0$, $x\%2 = 0 \wedge x\ 0$, $x\%2 \neq 0 \wedge x < 0$ and $x\%2 \neq 0 \wedge x > 0$[3].

```
1  if (x == 0) {
2      print("zero");
3      return;
4  }
5  if (x % 2 == 0) {
6      print("even");
7  } else {
8      print("odd");
9  }
10 if (x < 0) {
11     print("negative");
12 } else {
13     print("positive");
14 }
```

(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

2.2.2 Compositional Symbolic Execution

An issue that arises from symbolic execution is that is scales quite poorly, because of path explosion [21], [22], as each branch can potentially multiply by two the total number of branches.

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, and minimises the effect of path explosion.

A great tool for 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.

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 [23] 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

[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 $x = 0$, which is included in $x > 0$ and $x < 0$

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 [24] 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 [25] 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 smallers 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.

jStar [26] for instance is a tool allowing automated verification of Java code, and 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 targetted at Java and a subset of C, while SpaceInvader [28] allows for the verification of C code in device drivers.

Infer [8] is a tool developped 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's approach, while initially stemming from separation logic, was used for finding bugs rather than proving correctness, which led to the creation of ISL [9] to provide a sound theory justifying this approach. Following this, Pulse [29] 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 [30], [31] 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 [32], 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.

An example of such is Viper [33], 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. This is precisely 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 and a compiler to GIL, it's intermediary language. Currently Gillian has been instantiated to the C language (via CompCert-C [12]) and JavaScript, as well as Rust [34]. 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
Infer	C, C++, Java, Objective-C	✓	✗	OX
Pulse	C, C++, Java, Objective-C	✓	✗	UX
Cosette	JavaScript	✓	✗	WPST
JaVerT	JavaScript	✓	✗	WPST, OX
Viper	IR (Viper) 0	✓	✗	OX
Gillian	IR (GIL)	✓	✓	WPST, OX, UX

WPST = Whole Program Symbolic Testing.

Table 2.1: Comparison of verification tools

2.4 Gillian

Gillian [3]–[5] is a CSE engine, that supports UX true bug-finding, OX full verification, and exact (EX) whole-program testing [11]. It's first significant characteristic is thus a unified platform, that supports all three modes of operation.

Secondly, unlike other CSE engines, it is not targetted towards a specific programming language, and is instead parametric on the memory model of the target language. One can thus instantiate Gillian to their preferred language, and enjoy a complete CSE engine without having to build one from scratch. This is done by both providing a memory model of the language – code written in OCaml – and a compiler from the target language to GIL, an intermediate language used by Gillian.

Gillian uses PCMs for state models, which, as seen above, comes with certain restrictions. GIL programs interact with the state with *actions*, that are implemented via the

memory model – because actions are the only way for the program to interact with state, having sound actions is sufficient to then have sound language semantics. A simple heap model for instance could expose actions such as `allocate`, `free`, `load` and `store`.

Gillian has been in development for several years now, and a gap has started to form between the implementation and the theory, that has seen many evolutions (such has the notion of EX testing [11]) that didn't exist at the time of its original creation. Furthermore, currently unpublished papers aim to improve its existing theory – and while some of these improvements have already been ported to the source code, it still carries much of its past decisions.

Chapter 3

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 progresses made in two unpublished papers of the group, the so-called CSE 1 [35] and 2, as well as an upcoming PhD dissertation. 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 [19]. This attempt will feature OX verification.

The initial attempt will already 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 already existing in Gillian.

Later, UX true bug finding with bi-abduction will be added, again following the approach in the unreleased papers and existing research, notably with ISL [9] and Infer-Pulse [29].

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.

In parallel, a toolkit of separation algebras constructs will be developed, to allow new developers to easily construct complex state models from simpler elements.

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

Some of the innovations of the project include:

- Providing a mechanisation of program verification using Iris-like state models
- Providing the first (to the knowledge of the author) CSE that is parametric on the

semantics of the target language – existing CSE engines usually have one target language or intermediary language.

- Making the above mentionned engine's implementation close to theory, and implemented in a way to easily allow future mechanised proof of soundness.

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.-É. 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.-É. 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, Oct. 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[7] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional Shape Analysis by Means of Bi-Abduction," *J. ACM*, vol. 58, no. 6, Dec. 2011, ISSN: 0004-5411. DOI: 10.1145/2049697.2049700. [Online]. Available: https://doi.org/10.1145/2049697.2049700.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] 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, Aug. 2010, pp. 169–182. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-next-700-separation-logics/.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[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] D. Kroening, P. Schrammel, and M. Tautschnig, *CBMC: The C Bounded Model Checker*, 2023. arXiv: 2302.02384 [cs.SE].

[24] 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.

[25] 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.

[26] 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.

[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, *et al.*, "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] 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, Apr. 2022. DOI: 10.1145/3527325. [Online]. Available: https://doi.org/10.1145/3527325.

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

[31] 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, Jan. 2019. DOI: 10.1145/3290379. [Online]. Available: https://doi.org/10.1145/3290379.

[32] 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.

[33] 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.

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

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