IMPERIAL

Gillian 2--
A Language Agnostic CSE

Background & Progress Report

Author: Opale Sjöstedt

Supervisor: Philippa Gardner

May 31, 2024

Contents

Chapter 1

Introduction

Intro

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 **hoarelogic** 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 the statement, expressed as $\{P\}\, S\, \{Q\}$. For assignement for instance, one could have $\{\mathtt{x} = x\}\, \mathtt{x} := 0\, \{\mathtt{x} = 0\}$.

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) **seplogic1**, **seplogic2** is an extension of Hoare logic that permits this in a clear and scalable way. It's main addition is $*$, a separating conjunction: $P * Q$ means not only that the heap satisfies P and Q, but that it can be split in two *disjoint* parts, such that one satisfies P and the other Q. This allows us to reason about state, by stating what properties are upheld, and how they may be split for further proofs. For instance, given a *list* predicate, when calling a function that mutates the list we 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*:

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

Because we know that the code 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. 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 **seplogic2**.

From these we can then build more complex predicates. A traditional example is 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 can not be split into two disjoint heaps both satisfying $a \mapsto x$. Similarly, $a \mapsto x * b \mapsto y$ can only hold if $a \neq b$.

Equipped with composition symbolic execution and separation logic, we are capable of automating verification of code, and ensuring that no bug exists in a given program, according to a manually written specification.

2.1.1 Bi-Abduction

Explain

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) **isl**, derived from Incorrectness Logic **incorrectnesslogic**. It is similar to SL, but *under-approximate* (UX), 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.

2.1.3 State Models

To support sound verification engines, research also exists around describing the state used by different languages, in a modular fashion – although broadly related to SL, this research also has roots in category theory and abstract algebra. Some litterature around this includes:

- **fracpermissions** proposes an axtension of the traditional "points to" predicate, to allow for fractional permissions that can be used in divide-and-conquer algorithms or in concurrent applications.
- **abstractseplogic** gives a formal definition of separation algebra and predicates that satisfy abstract heaps.
- **sepalgebra** defines separation algebra, a way of modelling the state of programs in a composable way. It uses relations and defines axioms a relation must respect to be used as a separation algebra.
- **higherorderseplogic** uses partial commutative monoids and shows constructions that can be used in higher-order separation logic, as seen in Iris.
- **statesoundness** further researches the idea of state models, and in particular properties of separation algebras, such as having multiple units, and the active and passive execution pre-order.

2.2 Gillian

Gillian **gillian0**, **gillian1**, **gillian2** is a CSE engine, that supports UX true bug-finding, OX full verification, and exact (EX) whole-program testing **exactsl**. 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 has been in development for several years now, and a gap has started to form between the theory, that has seen many evolutions (such has the notion of EX testing **exactsl**) 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 has further diverged from a theory. We will thus aim to write Gillian 2.0--, a rework of the original Gillian, taking into account advanced made in the past years as well as additional innovations in the field and ideas from the author of this report.

2.3 Program Verification

2.3.1 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 and simple to realize, 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 **smartfuzzing**, 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 **surveysymex**. These values are abstract, and are then restricted as an interpreter steps through the code. 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 odd (see Figure 2.1), symbolic execution would thus branch thrice, once at each condition[1]. This would then result in 5 different branches, each with different constraints on the program variable $\mathtt{x}$: $\mathtt{x} = 0$, $\mathtt{x}\%2 = 0 \land \mathtt{x} < 0$, $\mathtt{x}\%2 = 0 \land \mathtt{x}\, 0$, $\mathtt{x}\%2 \neq 0 \land \mathtt{x} < 0$ and $\mathtt{x}\%2 \neq 0 \land \mathtt{x} > 0$[2].

2.3.2 Compositional Symbolic Execution

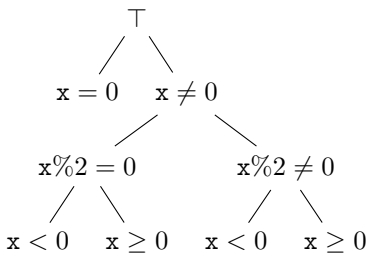
An issue that arises from symbolic execution is that is scales quite poorly, because of path explosion **pathexplo**, **surveysymex**, as each branch can potentially multiply by two the total number of branches.

A solution to this is *compositional* symbolic execution (CSE), in which the code is tested compositionally, function by function **compositionalsymtesting**. This allows for gradual

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

```
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

adoption, as only parts of the code base can be tested, and minimises the effect of path explosion, as only the conditionals in the treated function will cause branching.

To support functions calls, functions are enhanced with *function summaries*, that specify its preconditions and postconditions and that are used when it is called. These summaries, or specifications, express the state at the start and end of the function with first order logic, as follows **compositionalsymtesting**:

$$\begin{aligned}\phi_f = &(s[0] = c \wedge \texttt{ret} = 0)\\ &\vee (s[0] \neq c \wedge s[0] = 0 \wedge \texttt{ret} = -1)\\ &\vee (s[0] \neq c \wedge s[0] \neq 0 \wedge s[1] = c \wedge \texttt{ret} = 1)\\ &\vee \text{etc.}\end{aligned}$$

This approach uses first order logic, and while a good first step towards thinking compositionally, more tools are needed to allow for a more scalable reasoning.

2.4 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 **cbmc** 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 **klee** 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 **macke** 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 **jstar** 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 **verifast** is a CSE tool that is targetted at Java and a subset of C.

Infer **infer** 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 **isl** to provide a sound theory justifying this approach. Following this, Pulse **pulse** 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 **javert1**, **javert2** 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 **cosette**, 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 **viper**, 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 **gillian0**, **gillian1**, **gillian2** 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 **compcert**) and JavaScript, as well as Rust **gillianrust**. It is a generalisation of JaVerT, which only targeted JavaScript.

Name	Target Language	Compositional	Parametric State Model
CBMC	C	✗	✗
KLEE	LLVM	✗	✗
MACKE	LLVM	✓	✗
jStar	Java	✓	✗
VeriFast	C, Java	✓	✗
Infer	C, C++, Java, Objective-C	✓	✗
Pulse	C, C++, Java, Objective-C	✓	✗
Cosette	JavaScript	✓	✗
JaVerT	JavaScript	✓	✗
Viper	Any	✓	✗
Gillian	Any	✓	✓

Table 2.1: Comparison of verification tools

Chapter 3

Project Plan

Discuss the plan for the remainder of the project. Include a timeline. It is useful to define what success in your project will look like. You might be able to compare your system against existing systems.