Implementing Separation Logic using an SMT-backed Frame Rule

Kirill Golubev
Alcides Fonseca
alcides@ciencias.ulisboa.pt
LASIGE, Faculdade de Ciências da Universidade de Lisboa
Lisboa, Portugal

Abstract

Symbolic execution is a technique frequently used to reason about code. In symbolic execution, the analyzer keeps track of a logical state representation, and correctness verification are SMT queries. Separation logic is frequently used to express and verify the properties of programs with pointers or references. However, most SMT solvers (like the popular z3) do not support Separation Logic natively. CVC5 has introduced partial support for separation logic, which has not yet been integrated into more high-level tools.

This work aims to address this gap, by providing a proof of concept for implementing the Frame Rule using SMT queries in the Symbolic Heap fragment of separation logic, supported by CVC5. We conclude that this encoding can simplify the machinery dealing with separation logic, such as that present in Viper, Smallfoot, and others.

Todo ►*Viper is a debatable example, as it does not use separation logic internally. Instead, it relies on a more powerful mechanism of Implicit Dynamic Frames.*◄

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

ACM Reference Format:
Kirill Golubev and Alcides Fonseca. 2018. Implementing Separation Logic using an SMT-backed Frame Rule. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

There are few known ways to ensure that a computer program contains the least amount of errors. One of them is formal verification. The idea is to prove that a given program satisfies a specification provided in advance. The language of program specification is usually some logic that fits to describe program behavior. There are many ways to achieve this with varying resource requirements and automation degrees.

One technique employed to verify imperative programs is symbolic execution. Usually, the engine for it is implemented separately for each tool. SMT solvers are used as oracles to verify that each step of the symbolic execution engine is correct.

Most programs in imperative languages are written in terms of heap manipulation and the logic that is geared towards describing such programs is separation logic. Separation logic is an extension of Haor logic. It introduces some new operations and constants to it alongside the new inference rule called "frame rule". In general separation logic is proved to be undecidable, but there are decidable subsets. The one that is of interest to this paper is the "symbolic heap" fragment. It enables a significant degree of automation by being supported in the CVC5 SMT solver.

A symbolic heap requires the logical context to be split into a pure boolean part and a pure spatial part. This significantly restricts the expressive power of separation logic but permits encoding some interesting properties of programs. We chose it as a starting point, CVC5, for example, supports a larger fragment called GRASS. In contrast, Z3 does not have any support for separation logic.

The goal of the present work is to provide an opportunity to shift some heavy lifting related to separation logic from a symbolic execution engine to an SMT solver.

This is done by means of providing an algorithm to encode the frame rule through SMT solver queries.

Todo ►*Link to Smallfoot*◄

2 Frame Rule

The algorithm uses the notion of SMT query that is denoted as follows.

```
isUNSAT(¬query) = true,
```

iff an SMT solver gives the UNSAT result on the negation of the query. This means that for all free variables negation of

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

the query does not hold, which is equivalent to the situation when the query holds for all free variables.

$$\frac{\{pre\}\ code\ \{post\}}{\{pre * frame\}\ code\ \{post * frame\}}\ \text{Frame rule}$$

The algorithm itself is split into two phases.

- Check if the current context satisfies the precondition
- Apply postcondition to the larger context

Pseudocode for the first phase is quite simple. It exploits the idea, that it is possible to encode a heap containing any given one by adding `* true` to it.

The outline is that during the first step, it checks if the boolean context, defining pointer equivalence, and the heap imply precondition.

```
BCtx | H ⊩ pre
 iff isUNSAT(¬(BCtx ∧ H ⟹ pre * true))
```

The second phase exploits the same idea. The frame is inferred by checking each pointer for belonging in a frame, by precondition invalidation.

```
H = h₀ * ... * hₙ₋₁ //larger context
frame = H.map(λ h. pre * h * true)
          .filter(λ c. BCtx | H ⊩ c)
          .fold(emp, *)
```

The unfortunate consequence of this approach is a performance hit. Usually, the systems that are using SMT solvers need only `O(1)` SMT queries to make the symbolic execution step, but this algorithm does it in `O(size(H))` queries.

If examined more closely, this algorithm is not exactly doing frame rule application, but rather heap reconstruction. It will discard every heaplet that invalidates precondition and the remainder will be the result. This makes it possible to use it for other purposes with slight variations. For example, for merging heaps after branching.

3 Conclusions

One big advantage of this approach is that the SMT solver algorithm for separation logic is decidable. In contrast to Viper which is a notable alternative to writing a symbolic execution engine from scratch.

The simplicity and decidability come with the cost of features that are possible to support. Viper is a much more mature and rich backend for the language, while the presented approach is capped by the capabilities of separation logic support in SMT solver. Said capabilities are defined by GRASS fragment of separation logic which looks like "propositional" separation logic. The main features that are kept unreachable by this limitation are recursive predicates and fractional permissions.

The primary target of this algorithm was Liquid Java, but it is general enough to be used in other projects relying on SMT solvers to verify symbolic execution steps. The primary benefit of this algorithm is simplicity and delegation of responsibility for separation logic handling to the SMT solver instead of a symbolic execution engine which is usually implemented separately for each tool.

We implemented and tested this algorithm for Liquid Java, preliminary results are encouraging feature- and performance-wise. The prototype supports function calls, conditional branching, and assignments.

The performance degradation for synthetic benchmarks is around 30% relative to the pure boolean version of Liquid Java.

References

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009