Kalpa Publications in Computing

Volume 9, 2018, Pages 23–36

LPAR-22 Workshop and Short Paper Proceedings

Harrsh: A Tool for Unified Reasoning about Symbolic-Heap Separation Logic

Jens Katelaan[2], Christoph Matheja[1], Thomas Noll[1], and Florian Zuleger[2]

[1] RWTH Aachen University, Germany
[2] TU Wien, Austria

Abstract

In this tool paper we present Harrsh – a tool for unified reasoning about symbolic-heap separation logic. Harrsh supports the analysis of *robustness properties* of the symbolic heap fragment of separation logic with user-defined inductive predicates. Robustness properties, such as satisfiability, reachability, and acyclicity, are important for a wide range of reasoning tasks in automated program analysis and verification based on separation logic. Harrsh makes use of heap automata, which offer a generic approach to reasoning about robustness properties. We report on experimental results for several robustness properties taken from the literature and compare against satisfiability checkers participating in a recent competition. We conclude that a generic approach to checking robustness is feasible and promising for the extension to further properties of interest.

1 Introduction

Separation logic (SL) [18] is a popular formalism for Hoare-style verification of imperative, heap-manipulating programs. Its symbolic heap fragment serves as formal basis for a multitude of automated verification tools, such as SLAyer [1], VeriFast [14], Infer [5], Sleek [8], VCDryad [16], GRASSHopper [17], Songbird [20], Spen [9] and SLS [19]. These tools are capable of proving complex properties of a program's heap, such as memory safety, for large code bases [6, 7]. Many of these tools rely on *systems of inductive predicate definitions (SID)* to specify the shape of data structures employed in a program, such as trees and linked lists. Originally, separation logic tools implemented highly-specialized procedures for such fixed SIDs. As this limits their applicability, there is an ongoing trend to support custom SIDs that are either defined manually [14, 8, 4, 20, 9] or even automatically generated. The latter may, for example, be obtained from the tool Caber [2].

Robustness properties Allowing for arbitrary SIDs, however, raises various questions about their *robustness*. A user-defined or auto-generated SID might, for example, be *inconsistent*, introduce *unallocated logical variables*, specify data structures that contain undesired *cycles*, or produce *garbage*, i.e., parts of the heap that are unreachable from any program variable. Accidentally introducing such properties into specifications can have a negative impact on performance, completeness, and even soundness of the employed verification algorithms:

G. Barthe, K. Korovin, S. Schulz, M. Suda, G. Sutcliffe and M. Veanes (eds.), LPAR-IWIL 2018 (Kalpa Publications in Computing, vol. 9), pp. 23–36

- Brotherston et al. [4] point out that tools might waste time on inconsistent scenarios due to *unsatisfiability* of specifications.
- The absence of unallocated logical variables, also known as *establishment*, is required by the approach of Iosif et al. [12, 13] to obtain a decidable fragment of symbolic heaps.
- Other verification approaches, such as the one by Habermehl et al. [10, 11], assume that no garbage is introduced by data structure specifications.
- During program analysis, questions such as *reachability*, *acyclicity* and *garbage-freedom* arise depending on the properties of interest. For example, as argued by Zanardini and Genaim [21], acyclicity of the heap is crucial in automated termination proofs.

Checking *robustness properties* of custom SIDs is thus crucial (1) in debugging of separation logic specifications prior to program analysis and (2) in the program analyses themselves.

Heap Automata In [15] we have developed a unified approach to reasoning about robustness properties of symbolic-heap separation logic. We employ a novel automaton model, so-called *heap automata*, which works directly on the structure of symbolic heaps. A heap automaton examines an SID bottom-up, starting from the non-recursive base case. At each stage of this analysis, a heap automaton remembers a fixed amount of information. Heap automata enjoy a variety of closure properties (intersection, union and complementation). We have proposed the following approach for the analysis of SIDs with heap automata: given a heap automaton and an SID of interest, we compute the intersection of the heap automaton with the SID (a process called *automatic refinement*) and then check for emptiness (see Section 2 for an overview of the approach). We have shown that several robustness properties from the literature such as satisfiability, establishment, reachability, garbage-freedom, and acyclicity can be captured by a heap automaton. For these properties our heap automaton approach allows us to obtain decision procedures and counterexample generation in a unified framework. Further, we discuss in [15] that the heap automaton approach is promising for the *entailment problem*. We construct heap automata for several interesting SID instances such as trees with linked leaves, which allows us to obtain decision procedures for the entailment problem with regard to these SIDs.

Harrsh In this paper, we report on experiments with HARRSH—our tool for deciding robustness properties based on heap automata. HARRSH implements a generic approach which supports unified reasoning: a user can define a heap automaton for a property of interest. HARRSH then refines the SID with the automaton. Finally, the resulting SID can be checked for emptiness.

Contributions We have conducted the following experiments in order to evaluate the effectiveness of HARRSH: We have checked all of the aforementioned robustness properties on the SIDs generated by CABER [2]. HARRSH was able to solve all considered robustness properties on all of these SIDs in a matter of milliseconds. As the SIDs were derived from code, this demonstrates the practical feasibility of using heap automata for checking robustness properties. In a second experiment, we have compared the performance of satisfiability checking by HARRSH and several satisfiability checkers from the literature. Unlike HARRSH, these checkers implement dedicated decision procedures for checking the satisfiability problem. This experiment has shown that the generic heap automaton approach is competitive with state-of-the-art satisfiability checkers. We conclude that a generic approach to checking robustness properties is feasible and promising for the extension to further properties.

Organization of the paper We give an overview on the heap automaton approach in Section 2. We describe the implementation of HARRSH in Section 3. We discuss our experiments in Section 4 and conclude in Section 5.

2 Overview of the Approach

We briefly discuss the main ideas underlying our approach to reasoning about robustness properties. A detailed formalization is found in [15]. Our goal is to reason about symbolic heaps, i.e. a fragment of separation logic in which every formula is of the form

$$\varphi(x_1, \dots, x_n) \;=\; \exists z_1, \dots, z_m \,.\, \Sigma : \Pi \;,$$

where Σ is a collection of spatial assertions, i.e. the empty heap **emp**, points-to assertions $u \mapsto v$, and *predicate calls* $P(u_1, \dots, u_k)$, connected by the separating conjunction $*$. Moreover, Π is a conjunction of finitely many pure formulas, i.e. equalities and disequalities between variables. For example, the symbolic heap

$$\varphi(x, y) = \exists z \,.\, x \mapsto z * z \mapsto y : \{x = y\}$$

specifies that $x = y$ lies on a cyclic list of length two. In particular, specifications in symbolic-heap separation logic are tight in the sense that a heap containing anything but a cyclic list of length two would violate the above formula. The meaning of predicate calls is usually defined by *systems of inductive definitions* (SID), i.e. a finite set of rules mapping symbolic heaps to predicates. Consider, for instance, the following SID Φ:

$$\begin{array}{lcl} \mathsf{sll}(x, y) & \Leftarrow & \mathbf{emp} : \{x = y\} \\ \mathsf{sll}(x, y) & \Leftarrow & \exists u \,.\, x \mapsto u * \mathsf{sll}(u, y) : \{x \neq y\} \end{array}$$

The SID Φ specifies a singly-linked list segment between variables x and y by means of two rules: Either both variables coincide and the list segment is empty or x has a successor u (specified by the *points-to assertion* $x \mapsto u$), which in turn is at the head of a (shorter) singly-linked list segment, $\mathtt{sll}(u, y)$. The inequality in the second rule guarantees that there is no cyclic model.

The semantics of a predicate call then corresponds to replacing it by any exhaustive unfolding according to the SID's rules. Analogously to context-free grammars known from the theory of formal languages, we call the set of all such unfoldings the *language* of an SID for a given predicate (we adhere to this analogy throughout this section). For example, the language $L(\Phi)$ obtained by unfolding the singly-linked list predicate $\mathsf{sll}(x, y)$ consists of the following formulas:

$$\begin{array}{l} \mathbf{emp} : \{x = y\} \\ \exists u_1 \,.\, x \mapsto u_1 * \mathbf{emp} : \{x \neq y, u_1 = y\} \\ \exists u_1, u_2 \,.\, x \mapsto u_1 * u_1 \mapsto u_2 * \mathbf{emp} : \{x \neq y, u_1 \neq y, u_2 = y\} \\ \vdots \end{array}$$

In total, $L(\Phi)$ consists of all (possibly empty) singly-linked list segments between x and y.

We make use of the analogy between SIDs and context-free grammars to illustrate the approach we implemented in HARRSH. To this end, recall the following well-known result from the theory of formal languages:

Proposition 2.1. *Given a context-free grammar $\mathcal{G}$ and a finite automaton $\mathcal{A}$, one can effectively construct a context-free grammar $\mathcal{G}'$ with*

$$L(\mathcal{G}') = L(\mathcal{G}) \cap L(\mathcal{A}),$$

where $L(\cdot)$ denotes the language of finite words generated by the context-free grammar resp. accepted by the finite automaton.

It is thus possible to "refine" a context-free grammar such that only words satisfying a property given by automaton $\mathcal{A}$ can be derived. Moreover, since the emptiness problem for context-free grammars can be discharged in polynomial time and finite-state automata are closed under Boolean operations, the above proposition immediately yields two decision procedures:

1. We can decide whether there exists a word in $L(G)$ that satisfies a property given by automaton $\mathcal{A}$, i.e. $L(G) \cap L(\mathcal{A}) \neq \emptyset$.
2. We can decide whether all words in $L(G)$ satisfy a property given by automaton $\mathcal{A}$, i.e.

$$L(G) \subseteq L(\mathcal{A}) \quad \text{iff} \quad L(G) \cap \overline{L(\mathcal{A})} = \emptyset.$$

Both the refinement of context-free grammars and the decision procedures can be lifted to reason about SIDs. In our setting, SIDs play the role of context-free grammars. However, instead of finite-state word automata, we use *heap automata*. Intuitively, these automata correspond to a certain class of finite-state tree automata (running on the unfolding trees of an SID) with additional side conditions to ensure that one can reason compositionally about the symbolic heap underlying an unfolding. As shown in [15], the above properties required to derive both decision procedures, i.e. intersection between context-free and regular languages, closure of languages of finite automata under Boolean operations, and efficient decidability of the emptiness problem, can be lifted to SIDs and heap automata. Hence, if a robustness property P is given as a language of a heap automaton, we can, for any SID Φ

1. construct a new SID Φ' such that Φ' specifies only the symbolic heaps specified by Φ that additionally satisfy P,
2. decide whether there exists an unfolding of Φ for a given predicate that satisfies P, and
3. decide whether all unfoldings of Φ for a given predicate satisfy P.

Let us illustrate this approach for the analysis of the following SID Φ:

$$\begin{array}{lcl} \mathsf{sll}(x,y) & \Leftarrow & \mathbf{emp} : \{x = y\} \\ \mathsf{sll}(x,y) & \Leftarrow & \exists u \,.\, x \mapsto u * \mathsf{sll}(u,y) : \{x \neq y\} \\ \mathsf{top}(x,y) & \Leftarrow & \mathsf{sll}(x,y) : \{x \neq y\} \end{array}$$

We note that Φ extends the SID for singly-linked lists by a top-level predicate $\mathsf{top}(x,y)$ that unfolds to a singly linked list with $x \neq y$. We are interested in deciding whether $\mathsf{top}(x,y)$ is satisfiable, i.e., whether there is an unfolding of $\mathsf{top}(x,y)$ such that the resulting formula is satisfiable. All formulas of Φ have free variables in the set $\{x,y\}$.

In [15] we showed that there is a heap automaton $\mathcal{A}_{SAT}$ that accepts all satisfiable formulas without predicates calls whose free variables belong to some fixed finite set of variables. Intuitively, the heap automaton $\mathcal{A}_{SAT}$ needs to track for each free variable x_i whether the variable is definitely allocated and for each pair of variables x_i, x_j whether definitely $x_i \neq x_j$ or $x_i = x_j$; for a precise definition of $\mathcal{A}_{SAT}$ we refer the reader to [15].

In the above examples, the free variables to track are $\{x, y\}$. Applying the refinement theorem with Φ and $\mathcal{A}_{SAT}$ thus results in an SID Φ' that has the following rules:

$$\begin{array}{lcl} \mathsf{sll}^{\emptyset}_{x=y}(x,y) & \Leftarrow & \mathbf{emp} : \{x = y\} \\ \mathsf{sll}^{\{x\}}_{x\neq y}(x,y) & \Leftarrow & \exists u \,.\, x \mapsto u * \mathsf{sll}^{\{x\}}_{x\neq y}(u,y) : \{x \neq y\} \\ \mathsf{sll}^{\{x\}}_{x\neq y}(x,y) & \Leftarrow & \exists u \,.\, x \mapsto u * \mathsf{sll}^{\emptyset}_{x=y}(u,y) : \{x \neq y\} \\ \mathsf{top}(x,y)^{\{x\}}_{x\neq y} & \Leftarrow & \mathsf{sll}^{\{x\}}_{x\neq y}(x,y) : \{x \neq y\} \end{array}$$

Note that each predicate has been extended by the information which variable is definitely allocated (superscript) and which (dis)equality is definitely valid (subscript). This information becomes part of the predicate identifier, i.e., it is part of the syntax of the refined SID. The SID Φ', resulting from the refinement of Φ with $\mathcal{A}_{SAT}$, has the property that every formula in $L(\Phi')$ is satisfiable. This is not the case for the original SID Φ: $L(\Phi)$ contains the unsatisfiable unfolding $\mathbf{emp} : \{x = y, x \neq y\}$. The property that every formula in $L(\Phi')$ is satisfiable can be used to check the satisfiability of $\mathsf{top}(x, y)$ (here the fact that there is a refinement $\mathsf{top}(x, y)^{x}_{x\neq y}$ of $\mathsf{top}(x, y)$ proves the satisfiability). Moreover, the property that every unrolling is satisfiable is an important building block in the analysis of further robustness properties.

3 Implementation

We developed a prototype of our framework—called HARRSH[1]—that implements the refinement of SIDs with heap automata

HARRSH is licensed under the MIT license and available on GitHub.[2] The implementation currently consists of about 7000 lines of Scala code, excluding tests, comments and blank lines.

HARRSH implements two variants of refinement:

- **Full refinement.** Given an SID Φ and a heap automaton $\mathcal{A}$, compute a refined SID Φ' with $L(\Phi') = L(\Phi) \cap L(\mathcal{A})$.

  Using this algorithm, any SID can be transformed into a *robust* SID whose unfoldings are all in the language of automaton $\mathcal{A}$. Put differently, the algorithm filters out those unfoldings of the SID that do not satisfy the robustness property specified by the automaton.

- **Decision procedures using on-the-fly refinement.** Given an SID Φ and a heap automaton $\mathcal{A}$, decide whether there exists an unfolding $\varphi \in L(\Phi) \cap L(\mathcal{A})$ [15, Alg. 1].

  We can, for example, pass the automaton $\mathcal{A}_{SAT}$ introduced in the previous section to this algorithm to decide whether Φ is satisfiable.

  It is often possible to determine whether an unfolding $\varphi \in L(\Phi) \cap L(\mathcal{A})$ exists without computing the entire refined SID Φ'. While this does not change the worst-case performance, it often improves performance in practice.

Crucially, both algorithms are parametric in the heap automaton $\mathcal{A}$: Adding a new decision procedure to HARRSH is as simple as subclassing the `HeapAutomaton` interface, providing

1. A `type State`, specifying the state space of the heap automaton.

[1] Heap Automata for Reasoning about Robustness of Symbolic Heaps
[2] https://github.com/katelaan/harrsh/

2. An implementation of `def getTargetsFor(src: Seq[State], sh: SymbolicHeap): Set[State]`. Intuitively, this function encodes the transition relation of the heap automaton: It computes the set of all target states that the automaton can reach in a single step from the sequence of source states `src` when reading the given symbolic heap `sh`.

Supported properties. HARRSH currently defines heap automata for the following properties.

1. Satisfiability and unsatisfiability.
2. Establishment checking for proving absence of dangling pointers and
3. Non-establishment checking for detecting the possibility of dangling pointers.
4. Garbage freedom, i.e., absence of unreachable allocated memory locations.
5. Presence of garbage in the heap.
6. Definite reachability between pairs of variables in the heap.
7. Strong cyclicity, i.e., guaranteed presence of a cycle in the heap.
8. Weak acyclicity, i.e., acyclicity of all paths involving only non-dangling pointers.

At first sight, it may seem redundant to have dedicated automata for both satisfiability and unsatisfiability. Since heap automata are closed under complementation, this is true in theory. In general, complementation incurs an exponential blow up, however, whereas the complement automata implemented in HARRSH have the same size as the non-complemented automata.

Having complement automata in HARRSH also enables checking whether *all* unfoldings of a symbolic heap have a given property: Recall that to check whether a symbolic heap *has* a satisfiable unfolding, the heap automaton for satisfiability can be passed to the on-the-fly refinement algorithm. Conversely, to check whether *all* unfoldings of a symbolic heap are satisfiable, you can use the heap automaton for *unsatisfiability* and negate the result of on-the-fly refinement: If there does *not* exist an unsatisfiable unfolding than all unfoldings are satisfiable.

HARRSH also contains an experimental entailment checker based on heap automata that is currently unpublished.

Additional features. HARRSH also has support for several features that are useful for or derived from the refinement algorithm. Notably,

1. **Witness generation.** Given an SID Φ and an automaton $\mathcal{A}$, generate an unfolding of Φ that has the property specified by $\mathcal{A}$. This can also be used for **counterexample generation** by using the complement automaton: For example, to show that not all unfoldings of Φ are satisfiable, compute a witness for unsatisfiability of Φ.
2. **Model generation.** Given a satisfiable formula, HARRSH can compute a model of the formula, i.e., a stack and heap that satisfy the formula. Models are generated by computing a witness for satisfiability and converting this witness to a model.
3. **Partial target computation.** Refinement tends to perform badly for predicates and symbolic heaps with many recursive calls: Assuming two or more automaton states are reachable for each predicate, the number of transitions considered in the refinement algorithm is exponential in the maximum number of predicate calls in the SID (see [15] for details).

In many cases, this exponential blowup can be avoided. For example, it is often sufficient to consider only a subset of the recursive calls in an unsatisfiable symbolic heap to prove its unsatisfiability. When defining a heap automaton in HARRSH, it is possible to define a *partial target computation*. In this approach, the target states are constructed incrementally, processing the source states `src` of the automaton transitions one at a time. This makes it possible to short-circuit the computation of the target states as soon as the result (e.g. unsatisfiability) is determined. The implementation of this optimization for (un)satisfiability checking led to a substantial speedup on many instances from the SL-COMP'18 competition.

Input formats. HARRSH currently supports three different input formats.

- The CYCLIST [3] input format.
- The subset of the SL-COMP'18 input format used in the categories `qf_shls_sat` and `qf_shid_sat`.
- Our own custom format for specifying SIDs. For example, lasso-shaped singly-linked lists can be specified as follows using our format.[3]

```
# Lasso-shaped list structure with head x1
lasso <= x1 -> y * lasso(y);
lasso <= x1 -> y * sll(y,x1);

# Underlying singly-linked list segments from x1 to x2
sll <= emp : {x1 = x2};
sll <= x1 -> y * sll(y,x2)
```

Output. When HARRSH performs full refinement, it outputs the refined SID in HARRSH's own format. For example, refining the above SID for lasso-shaped lists with the automaton for satisfiability yields the following result.

```
lasso <= lasso2(x1) ;
sll1 <= emp : {x1 = x2} ;
sll0 <= x1 -> y1 * sll1(y1,x2) ;
sll0 <= x1 -> y1 * sll0(y1,x2) ;
lasso2 <= x1 -> y1 * sll1(y1,x1) ;
lasso2 <= x1 -> y1 * sll0(y1,x1) ;
lasso2 <= x1 -> y1 * lasso2(y1)
```

As described in the previous section, the predicate identifiers in the refined SID encode the reached state of the heap automaton. In this example, three states of the heap automaton were reached and mapped to the suffixes 0, 1, and 2 in the output.

When asked for a model, HARRSH prints a human-readable representation of a stack–heap model of the SID that satisfies the property. For example, for the `lasso` SID and the satisfiability automaton, HARRSH outputs

[3] For additional examples see https://github.com/katelaan/harrsh/.

```
Stack {
  x1 -> 1
}
Heap {
  1 -> 1
}
```

Note that HARRSH currently uses consecutive integers 1, 2, etc. as memory locations, rather than computing concrete memory addresses. In this example, there is a single heap-allocated location with a self-loop (`1 -> 1`) and pointed-to by the variable `x1`—the shortest possible lasso-shaped list with head `x1`.

Ongoing work. Our current efforts are focused on implementing heap automata for entailment checking of the bounded treewidth fragment of symbolic heaps proposed in [12]. We would also like to make improvements to the user interface; in particular, making it easier to explore models and countermodels of robustness properties.

4 Evaluation

We tested HARRSH on a large collection of benchmarks that is distributed with CYCLIST as well as on the benchmarks of this year's separation-logic competition, SL-COMP'18.

In particular, we evaluated the performance of HARRSH on:

1. 45945 problem instances that were automatically generated by the inference tool CABER [2].
2. A set of particularly hard problem instances that are derived from the SIDs used to prove lower complexity bounds for satisfiability [4].
3. The benchmarks from the SL-COMP'18 categories `qf_shid_sat` and `qf_shls_sat`.

The first two experiments were performed on an Intel Core i5-3317U at 1.70GHz with 4GB of RAM. The third experiment was conducted on the StarExec cluster.[4]

4.1 Performance on Inferred SIDs

To evaluate the performance of HARRSH on a realistic set of benchmarks, we ran both HARRSH and CYCLIST on all 45945 benchmarks generated by CABER [2]. For CYCLIST, we only checked satisfiability—the only of the robustness properties supported by CYCLIST; for HARRSH, we checked the whole range of the properties it supports.

Both tools were capable of proving (un)satisfiability of all of these problem instances within a set timeout of 30 seconds. All in all, the *cumulative* analysis time of HARRSH for these instances was 12460ms, while CYCLIST required 44856ms.[5] For a wide range of other properties, HARRSH also achieved *cumulative* analysis time below 20 seconds; see Table 1. These numbers demonstrate the applicability of our tool to problem instances that occur in practice.

[4] https://www.starexec.org/

[5] We ran HARRSH in batch mode to avoid the overhead of starting the JVM for each benchmark. For both tools we added up the analysis times of individual tasks, reported with millisecond precision. Consequently, we expect that rounding errors influence the accumulated time to a similar degree for both tools.

Property	**Analysis Time (ms)**
Satisfiability	12460
Complement of Satisfiability	11980
Establishment	18055
Complement of Establishment	17272
Reachability	14897
Garbage-Freedom	18192
Weak Acyclicity	18505

Table 1: Total analysis time for a subset of the properties supported by HARRSH on the 45945 SIDs that were inferred by CABER.

4.2 Performance on Worst-Case Instances

We ran HARRSH and CYCLIST on a set of particularly hard benchmarks that were handcrafted to illustrate exponential-time lower bounds for the satisfiability problem. These benchmarks are distributed with CYCLIST and were also part of the `qf_shid_sat` category of SL-COMP'18.

In our evaluation, we chose a timeout of 5 minutes for both tools. The measured runtimes are found in Table 2. Even though HARRSH has not been optimized for satisfiability checking, the runtimes of both tools are generally within the same order of magnitude. Both tools exhibit similar scalability. This further corroborates the feasibility of the unified reasoning approach implemented in HARRSH.

Note, however, that the runtimes of CYCLIST reported here deviate significantly from the ones observed in SL-COMP'18; we will further comment on this discrepancy below.

Benchmark	HARRSH	CYCLIST
succ-rec01.defs	3	0
succ-rec02.defs	10	8
succ-rec03.defs	24	12
succ-rec04.defs	106	20
succ-rec05.defs	496	128
succ-rec06.defs	2175	792
succ-rec07.defs	9692	4900
succ-rec08.defs	39408	31144
succ-rec09.defs	169129	164464
succ-rec10.defs	TO	TO
succ-circuit01.defs	80	4
succ-circuit02.defs	142	8
succ-circuit03.defs	699	48
succ-circuit04.defs	4059	832
succ-circuit05.defs	75110	28800
succ-circuit06.defs	TO	TO

Table 2: Comparison of HARRSH and CYCLIST for hard instances of the satisfiability problem. Provided times are in milliseconds. The timeout (TO) was 5 minutes (300000 milliseconds).

Solver	solved	time (s)	wrong	timeout	memout	unknown
CYCLIST [4]	90	5.1	9	0	0	0
HARRSH	72	246.6	0	27	0	0
S2S	71	706.3	0	28	0	0
SLEEK [8]	73	693.2	9	1	16	0
SONGBIRD [20]	63	300.5	0	22	14	0
SPEN [9]	3	0.1	2	0	0	94

Table 3: Results for SL-COMP'18 division qf_shid_sat. TO = 600s, MO = 4GB

4.3 SL-COMP 2018

HARRSH participated in the categories `qf_shls_sat` (satisfiability of quantifier-free symbolic heaps with singly-linked lists) and `qf_shid_sat` (satisfiability of quantifier-free symbolic heaps with arbitrary systems of inductive definitions) of this year's separation-logic competition.

Table 3 contains an edited version of the SL-COMP results for the category `qf_shid_sat`.[6][7] The reported time is the cumulative analysis time on solved instances.

Correctness. Both SLEEK and CYCLIST returned wrong results on several benchmarks.[8] For example, consider the formula R defined follows.

$$\begin{array}{lcl} \mathsf{dll}(x_1,x_2,x_3,x_4) & \Leftarrow & \mathbf{emp} : \{x_1 = x_3, x_2 = x_4\} \\ \mathsf{dll}(x_1,x_2,x_3,x_4) & \Leftarrow & \exists u \,.\, x_1 \to (u,x_2) * \mathsf{dll}(u,x_1,x_3,x_4) : \{x_1 \neq x_3, x_2 \neq x_4\} \\ \mathsf{R}(x,y) & = & \mathsf{dll}(x,\mathbf{null},\mathbf{null},y) * (y \to (\mathbf{null},\mathbf{null})) : \{x \neq y\} \end{array}$$

R appeared as benchmark `dll-01`[9] in the `qf_shid_sat` category. Both SLEEK and CYCLIST concluded that R is satisfiable. R is unsatisfiable, however:

1. $y \neq \mathbf{null}$ is implied by the points-to assertion $y \to (\mathbf{null}, \mathbf{null})$, whereas applying the non-recursive rule to unfold $\mathsf{dll}(x, \mathbf{null}, \mathbf{null}, y)$ would enforce $y = \mathbf{null}$. Consequently, we have to apply the recursive dll rule at least once.
2. In dll unfoldings that contain the recursive rule at least once, the fourth parameter is always equal to the last location that is allocated within the list. (The parameter that is allocated in the recursive rule is passed as second argument to the recursive call and the base rule enforces $x_2 = x_4$.)
3. y is the fourth parameter of the predicate call.
4. Hence y is allocated twice, once within the unfolding of dll and once in the points-to assertion $y \to (\mathbf{null}, \mathbf{null})$, proving that R is unsatisfiable.

[6] Adapted from https://www.irif.fr/~sighirea/sl-comp/18/qf_shid_sat.html. Some tools were run with a timeout of 2400s, some with a timeout of 600s, but no tool solved any benchmark in more than 600s. We can therefore assume a uniform timeout of 600s.

[7] At the time of writing, https://www.irif.fr/~sighirea/sl-comp/18/qf_shid_sat.html wrongly reports the number of instances solved by SONGBIRD to be 60 instead of 63. Additionally, the classification into timeout, memout and unknown is incorrect for multiple tools. We took the liberty to correct these data for this paper.

[8] In all these cases, the tools wrongly classified unsatisfiable benchmarks as satisfiable. SLEEK: atll-02, atll-03, dll-01, dll-02, dll-04, dll-06, lss-03-03, lss-04-03, tll-02; CYCLIST: atll-02, atll-03, dll-01, inconsistent-ls-of-ls.defs, lss-03-01, lss-03-02, lss-03-03, lss-04-03, tll-02.

[9] At the time of writing, this benchmark is available at https://github.com/sl-comp/SL-COMP18/blob/master/bench/qf_shid_sat/dll-01.smt2 in the official SL-COMP18 repository.

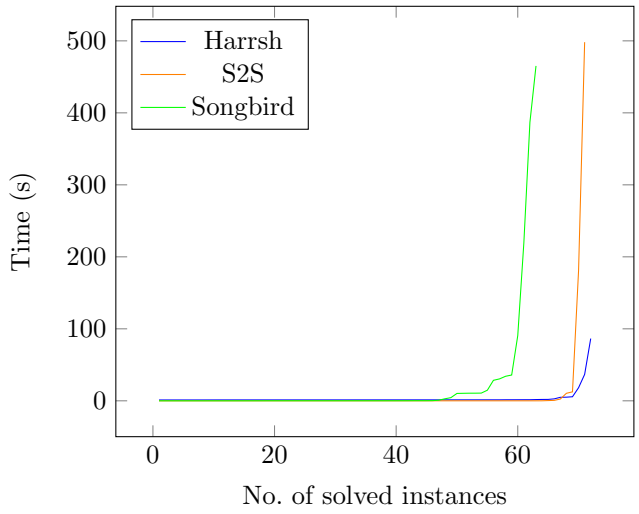

Figure 1: Cactus plot for SL-COMP'18 division `qf_shid_sat`, comparing the three best tools without wrong results. Timeout = 600s.

Crucially, at least in the case of CYCLIST, the soundness issues appear to have had a significant impact on the performance of the tool. Purportedly, the version of CYCLIST participating in SL-COMP was able to prove the satisfiability of all 40 worst-case instances (the easiest 16 of which are listed in Table 2) in a matter of milliseconds—despite the proof of exponential lower bounds for these benchmarks in [4] and despite exhibiting exponential behavior on our machine (cf. Table 2). We thus conjecture that the soundness issues of the CYCLIST version that participated in SL-COMP are the main cause of the difference in number of solved instances between CYCLIST on the one hand and SLEEK, S2S, and HARRSH on the other hand.

Performance on symbolic heaps with user-defined SIDs. Because of their unsoundness, we ignore both CYCLIST and SLEEK in the evaluation of the SL-COMP results. Figure 1 shows a cactus plot that compares the performance in the category `qf_shid_sat` of the three best tools that did not return any wrong results. The curves in the plot are to be interpreted as follows. If the curve for a tool goes through the point (x, y), the tool was able to solve x of the benchmarks using at most y seconds for each of these x benchmarks.

The plot reveals that the runtime is dominated by the handful of worst-case instances that the tools were able to solve within the time limit of 600s.

Figure 2 zooms in on the left region of the graph by assuming a timeout of 20s instead of 600s. HARRSH, as the only tool that runs on the JVM rather than natively, is much slower than the other tools on simple benchmarks. This is because merely starting the JVM takes approximately 1.2 seconds and thus dominates the runtime on simple benchmarks. This constant overhead can be easily avoided in practice by running HARRSH as a server rather than running one instance of HARRSH per benchmark. As the complexity of the benchmarks increases, HARRSH significantly outperforms SONGBIRD and performs slightly better than S2S.

In summary, Figures 1 and 2 thus demonstrate that HARRSH can compete with other state-of-the satisfiability checkers on benchmarks with user-defined SIDs.

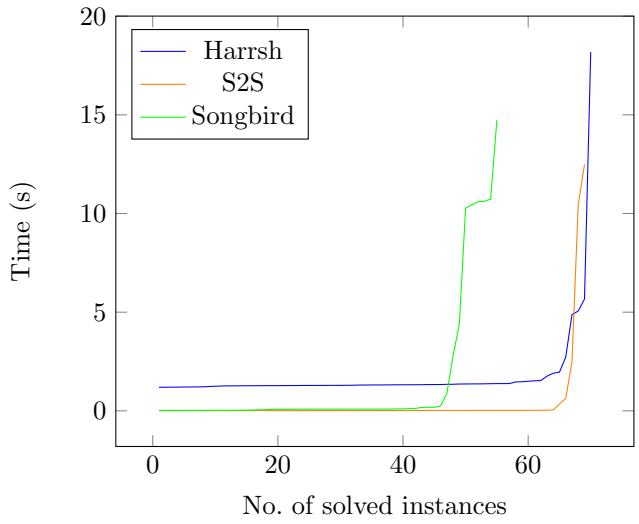

Figure 2: Cactus plot for SL-COMP'18 division `qf_shid_sat`, comparing the three best tools without wrong results. Timeout = 20s.

Performance on benchmarks with singly-linked lists. Every tool that participated in the category `qf_shls_sat` was able to correctly solve all 110 benchmarks in that category.

Overall, HARRSH did not perform as well on this category as most other participating tools. This is not surprising: Unlike most of the other tools, HARRSH does *not* implement any list-specific reasoning; additionally, refinement is not optimized for formulas with many recursive calls, which dominate the `qf_shls_sat` category. We are thus not discouraged by the fact that with the exception of CYCLIST (which does not implement list-specific reasoning either), all other tools completed this category one to two orders of magnitude faster than HARRSH. Because of the vast differences in performance, we do not present a more detailed comparison to other tools.

The cactus plot in Figure 3 shows in detail the performance of HARRSH in the category `qf_shls_sat`. It demonstrates that HARRSH is able to solve most benchmarks reasonably fast: 79 of 110 instances were solved in less than 2 seconds and 106 of 110 instances were solved in less than 5 seconds. Performance tends to be much worse on unsatisfiable instances, where on-the-fly refinement does not have any performance advantages over full refinement. (Only 4 of the 31 instances that took more than 2 seconds are satisfiable, whereas in total, exactly 50% of the instances are satisfiable.)

5 Conclusion

We presented HARRSH, a tool that implements *refinement* of inductive predicate definitions in symbolic-heap separation logic with a wide range of *robustness properties* encoded by means of a novel automaton model, *heap automata*. Refinement with a robustness property changes the set of formulas defined by the inductive definitions in such a way that all formulas that violate the property are filtered out. Supported properties include satisfiability, establishment, garbage freedom and (a)cyclicity checking. In addition, support for entailment checking using the same approach is currently under development.

HARRSH also leverages our approach—refinement with heap automata—to obtain decision

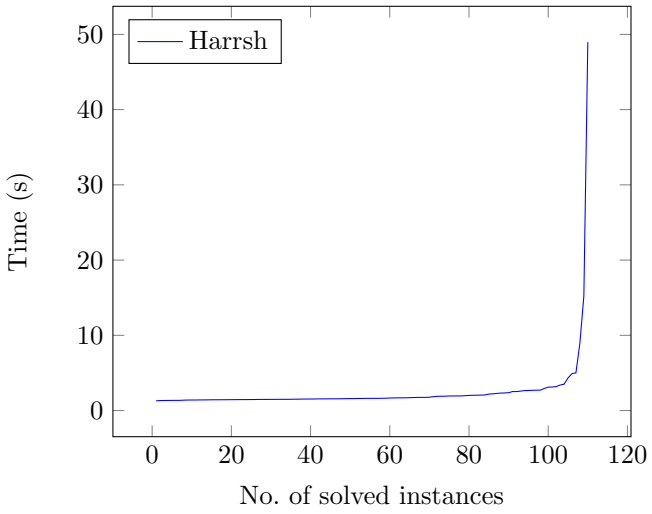

Figure 3: Cactus plot showing the performance of Harrsh in the SL-COMP'18 division `qf_shls_sat`.

procedures for the robustness properties (including satisfiability checking), find counterexamples to the properties and generate models that exhibit the properties.

The evaluation of HARRSH on both handwritten and generated benchmarks showed good performance across all implemented robustness properties. In particular, despite using a uniform approach for solving all robustness properties, HARRSH was able to compete with state-of-the-art satisfiability checkers for symbolic-heap separation logic in this year's separation-logic competition, SL-COMP'18.

References

[1] Josh Berdine, Byron Cook, and Samin Ishtiaq. "SLAyer: Memory Safety for Systems-Level Code". In: *CAV 2011*. Vol. 6806. LNCS. Springer, 2011, pp. 178–183 (cit. on p. 23).

[2] James Brotherston and Nikos Gorogiannis. "Cyclic Abduction of Inductively Defined Safety and Termination Preconditions". In: *SAS 2014*. Vol. 8723. LNCS. Springer, 2014, pp. 68–84 (cit. on pp. 23, 24, 30).

[3] James Brotherston, Nikos Gorogiannis, and Rasmus L Petersen. "A generic cyclic theorem prover". In: *APLAS 2012*. Springer. 2012, pp. 350–367 (cit. on p. 29).

[4] James Brotherston et al. "A decision procedure for satisfiability in separation logic with inductive predicates". In: *CSL-LICS 2014*. ACM, 2014, 25:1–25:10 (cit. on pp. 23, 24, 30, 32, 33).

[5] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *NFM 2011*. Vol. 6617. LNCS. Springer, 2011, pp. 459–465 (cit. on p. 23).

[6] Cristiano Calcagno et al. "Compositional shape analysis by means of bi-abduction". In: *POPL 2009*. ACM, 2009, pp. 289–300 (cit. on p. 23).

[7] Cristiano Calcagno et al. "Moving Fast with Software Verification". In: *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Cham: Springer International Publishing, 2015, pp. 3–11 (cit. on p. 23).

[8] Wei-Ngan Chin et al. "Automated verification of shape, size and bag properties via user-defined predicates in separation logic". In: *Sci. Comput. Program.* 77.9 (2012), pp. 1006–1036 (cit. on pp. 23, 32).

[9] Constantin Enea et al. "SPEN: A Solver for Separation Logic". In: *NASA Formal Methods*. Ed. by Clark Barrett, Misty Davies, and Temesghen Kahsai. Cham: Springer International Publishing, 2017, pp. 302–309 (cit. on pp. 23, 32).

[10] Peter Habermehl et al. "Forest automata for verification of heap manipulation". In: *CAV 2011*. Vol. 6806. LNCS. Springer, 2011, pp. 424–440 (cit. on p. 24).

[11] Peter Habermehl et al. "Forest automata for verification of heap manipulation". In: *Formal Methods in System Design* 41.1 (2012), pp. 83–106 (cit. on p. 24).

[12] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. "The Tree Width of Separation Logic with Recursive Definitions". In: *CADE-24*. Vol. 7898. LNCS. Springer, 2013, pp. 21–38 (cit. on pp. 24, 30).

[13] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. "Deciding Entailments in Inductive Separation Logic with Tree Automata". In: *ATVA 2014*. Vol. 8837. LNCS. Springer, 2014, pp. 201–218 (cit. on p. 24).

[14] Bart Jacobs et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NFM 2011*. Vol. 6617. LNCS. Springer, 2011, pp. 41–55 (cit. on p. 23).

[15] Christina Jansen et al. "Unified Reasoning About Robustness Properties of Symbolic-Heap Separation Logic". In: *Proceedings of ESOP, Uppsala, Sweden*. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 611–638 (cit. on pp. 24–28).

[16] Edgar Pek, Xiaokang Qiu, and P Madhusudan. "Natural Proofs for Data Structure Manipulation in C using Separation Logic". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 440–451 (cit. on p. 23).

[17] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "GRASShopper. Complete Heap Verification with Mixed Specifications". In: *Proceedings of TACAS, Grenoble, France*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 124–139 (cit. on p. 23).

[18] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *LICS 2002*. IEEE, 2002, pp. 55–74 (cit. on p. 23).

[19] Quang-Trung Ta et al. "Automated Lemma Synthesis in Symbolic-heap Separation Logic". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 9:1–9:29. ISSN: 2475-1421 (cit. on p. 23).

[20] Quang-Trung Ta et al. "Automated Mutual Explicit Induction Proof in Separation Logic". In: *FM 2016: Formal Methods*. Ed. by John Fitzgerald et al. Cham: Springer International Publishing, 2016, pp. 659–676 (cit. on pp. 23, 32).

[21] Damiano Zanardini and Samir Genaim. "Inference of Field-Sensitive Reachability and Cyclicity". In: *ACM Trans. Comput. Log.* 15.4 (2014), 33:1–33:41 (cit. on p. 24).