SPEN: A Solver for Separation Logic

Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, Tomáš Vojnar

SPEN: A Solver for Separation Logic

Constantin Enea[1], Ondřej Lengál[2], Mihaela Sighireanu[1], and Tomáš Vojnar[2]

[1] Univ. Paris Diderot, IRIF CNRS UMR 7089, France
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. SPEN is a solver for a fragment of separation logic (SL) with inductively-defined predicates covering both (nested) list structures as well as various kinds of trees, possibly extended with data. The main functionalities of SPEN are deciding the satisfiability of a formula and the validity of an entailment between two formulas, which are essential for verification of heap manipulating programs. The solver also provides models for satisfiable formulas and diagnosis for invalid entailments. SPEN combines several concepts in a modular way, such as boolean abstractions of SL formulas, SAT and SMT solving, and tree automata membership testing. The solver has been successfully applied to a rather large benchmark of various problems issued from program verification tools.

1 Introduction

For analyzing programs with dynamic memory, *separation logic* (SL) is an established and fairly popular logic introduced by Reynolds [11]. The high expressivity of SL, its ability to generate compact proofs, and its support for local reasoning motivated development of many tools for automatic reasoning about programs with complex dynamic linked data structures. These tools aim at establishing memory safety properties and/or inferring shape properties of the heap. The tools often build on (semi-)decision procedures for checking *satisfiability* and *entailment* problems in SL.

Our tool SPEN[3] provides (semi-)decision procedures for the most commonly considered *symbolic heaps* fragment of SL, extended with user-defined inductive predicates to specify data structures of an unbounded size. Because unrestricted definitions of inductive predicates make the entailment problem for the fragment undecidable [3], only semi-decision procedures have been proposed, e.g., in [2,4]. Iosif *et al.* [10] identified a rather large class of inductive definitions for which the entailment problem is decidable, although with a high complexity. SPEN focuses on a smaller class of inductive definitions that is, however, expressive enough to specify complex dynamic data structures, such as skip lists, lists of circular lists, AVL trees, or binary search trees.

The chosen class of inductive definitions enables the design of efficient (semi-)decision procedures for satisfiability and entailment [6,8]. The key idea used for satisfiability checking in SPEN is to exploit the semantics of restricted inductive definitions and of separating conjunction to build an equisatisfiable boolean abstraction of the formula. For entailment checking, the idea is to reduce the problem of checking $\varphi \Rightarrow \psi$ to the problem of checking a set of *simple entailments* where the right-hand side is an inductive predicate atom. The compositionality of this reduction leads to high efficiency (the simple entailments can be checked independently) and to a capability to provide fine diagnosis for invalid entailments.

[3] `https://github.com/mihasighi/spen`

The current version of SPEN improves on the ones reported in [6,8] in several directions. First, we introduced caching of constructions and results obtained from checking simple entailments in order to increase its efficiency. Second, the wrappers calling the SAT and SMT solvers have been refined to generate smaller formulas and to exploit the incrementality feature of underlying solvers. Third, we improved the diagnosis produced by SPEN. For satisfiability checking, SPEN now provides either a model of a satisfiable formula or an unsatisfiable core; for entailment checking, SPEN provides a proof witness for valid entailments and a diagnostic information otherwise.

SPEN has been successfully tested on a quite large benchmark. The first version of SPEN participated in the SL-COMP'14 contest [15] where it won one of its divisions and was second in another one. The later extensions now allow SPEN to handle a richer fragment than those considered in the competition. Moreover, the improvements above lead to better execution times (e.g., by 10 % within the SL-COMP'14 division won by the first version of SPEN and by 30 % on the division where SPEN was the second).

SPEN is not the only solver for SL. The existing solvers differ in the fragment considered (CYCLIST [2], SLIDE [9]) and/or the techniques used (ASTERIX [12], DRYAD [14], GRASSHOPPER [13], SLEEK [4]). A detailed comparison with these solvers is beyond the scope of this paper—we refer the reader to the survey in [6,8,15].

2 Logic Fragment

SPEN deals with decision problems in a fragment of SL, denoted as $\mathsf{SL}^{\mathsf{ID}}$, that combines the *symbolic heaps* fragment of SL [1] with user-defined *inductive predicates* describing various kinds of *lists* (possibly nested, cyclic, or equipped with skip links) or *trees*, possibly extended with data constraints.

Syntax: We write X, Y, Z to denote *location variables*, d to denote *data variables*, and x, y, z for both kinds of variables. We use the vector notation $\vec{x}$ to abbreviate tuples. We denote by ρ the tuples built from pairs of *field labels* and variables that specify structured values. We assume a finite set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of *predicate symbols*, each with an associated arity, and a special location variable nil. A *symbolic heap formula* ψ is a formula of the form $\exists \vec{x} \cdot \Pi \wedge \Sigma$ where Π is a *pure formula* and Σ is a *spatial formula* with the following syntax:

$$\Pi \ ::= \ X = Y \mid X \neq Y \mid \Delta \mid \Pi \wedge \Pi \qquad\qquad \Sigma \ ::= \ \mathtt{emp} \mid X \mapsto \rho \mid P(X, \vec{x}) \mid \Sigma * \Sigma$$

Here, Δ is a constraint over data variables. We let it unspecified, though SPEN presently supports the first-order theory over multisets of integers with integer linear constraints. The spatial atoms (i.e., the empty heap, the heap cell allocated at X, resp. the heap region shaped by some predicate $P \in \mathcal{P}$) are composed by the separating conjunction "$*$". An $\mathsf{SL}^{\mathsf{ID}}$ formula φ is a set of symbolic heaps interpreted as a disjunction $\vee_i \psi_i$.

Predicates $P \in \mathcal{P}$ are defined by a set of *inductive rules* of the form $\psi \Rightarrow P(X, \vec{x})$ where $(X, \vec{x})$ is a tuple of distinct variables including all free variables in the symbolic heap ψ (the rule body). X is called the root node of the heap segment defined by P. A rule is called a *base rule* if its spatial part is $\mathtt{emp}$, i.e., an empty heap; otherwise, it is an *inductive rule*.

Fragments: SPEN considers a restricted class of inductive rules such that the defined predicates specify (possibly empty) heap segments connecting (by location fields) the

root location X with all locations in the heap or nil. The restrictions have been defined formally in [6,8]. They mainly require, for each inductive predicate P, the presence of a unique base rule and inductive rules where the root X points to a memory cell that contains at least one field from which another heap specified by P starts. The fragment defined in [6], called $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$, can describe various kinds of lists that can be singly- or doubly-linked, cyclic, nested, and can have skip links. It does not permit data constraints and inductive tree structures. On the other hand, the fragment $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$ defined in [8] permits data constraints and can describe tree structures of bounded width, such as sorted list segments, AVL trees, binary search trees, but not nested cyclic lists.

Decision problems: For both fragments above, SPEN considers the problems of checking satisfiability of a formula, i.e., checking whether $\models \varphi$ holds, and the validity of an entailment $\varphi \Rightarrow \varphi'$ where the symbolic heaps of φ' can be quantified only over data variables. A simple example of an entailment problem in $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$ considered by SPEN is:

$$\exists Y, W.\, \mathtt{X} \neq \mathtt{Z} \wedge \mathtt{X} \mapsto \{(\mathtt{next}, Y)\} * \mathtt{sll}(Y, W) * W \mapsto \{(\mathtt{next}, \mathtt{Z})\} \quad \stackrel{?}{\Rightarrow} \quad \mathtt{sll}(\mathtt{X}, \mathtt{Z}),$$

which, intuitively, checks whether a composition of two memory cells specified by the *points-to atoms* $\mathtt{X} \mapsto \{(\mathtt{next}, Y)\}$ and $W \mapsto \{(\mathtt{next}, \mathtt{Z})\}$ and the predicate atom $\mathtt{sll}(Y, W)$ describes a set of heaps that are all also models of the predicate $\mathtt{sll}(\mathtt{X}, \mathtt{Z})$ defining an acyclic singly-linked list segment between X and Z.

3 Satisfiability Checking

Given a set of inductive definitions $\mathcal{P}$ and a symbolic heap ψ, the procedures for checking satisfiability in SPEN follow the workflow given in Fig. 1. The satisfiability checking of an $\mathsf{SL}^{\mathsf{ID}}$ formula φ makes a classic use of this basic procedure. The crux of the procedures for both fragments is the definition of a boolean formula $B[\psi]$, called *boolean abstraction*, such that the data-free part of ψ is satisfiable iff $B[\psi]$ is satisfiable [6,7].

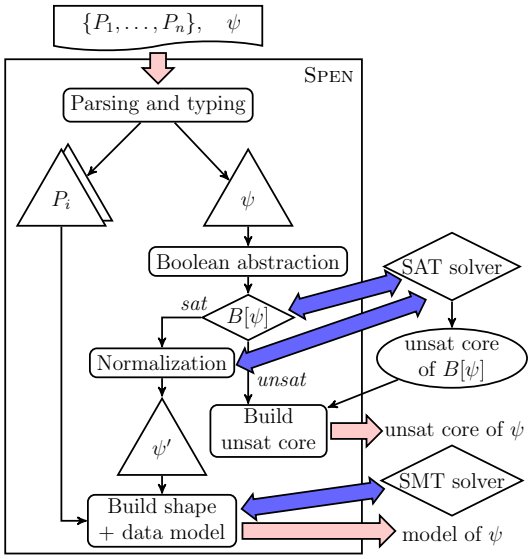

Fig. 1: SPEN workflow for satisfiability checking

Once the boolean abstraction $B[\psi]$ is computed, SPEN queries a SAT solver (currently, MINISAT[4]) for the satisfiability of $B[\psi]$. If $B[\psi]$ is unsatisfiable, SPEN can return an unsatisfiable core of ψ, deduced from an unsatisfiable core of $B[\psi]$. If $B[\psi]$ is satisfiable and $\psi \in \mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$, SPEN has the option of returning a model of ψ obtained from a model of $B[\psi]$ by unfolding predicate atoms corresponding to non-empty heap segments. The unfolding of predicate atoms is done twice to emphasize the non-emptiness

[4] Available at `http://minisat.se`.

of the segment. For $\psi \in \mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$, the satisfiability checking continues by constructing a formula Δ_ψ that conjuncts the data part of ψ with the data parts obtained by unfolding the non-empty heap segments given by the model of $B[\psi]$. To check the satisfiability of Δ_ψ, SPEN queries an SMT solver for the theory of multisets with integer data (currently, SPEN implements a wrapper for the UFLIA theory of Z3 [5]).

If the boolean abstraction $B[\psi]$ is satisfiable, it is then used to normalize the spatial part of ψ, which is a step used by entailment checking too. This process saturates the pure part of ψ with (dis-)equalities between locations variables and removes predicate atoms that correspond to empty heap segments, producing a normalized formula ψ'.

4 Entailment Checking

To check the validity of an entailment $\varphi_1 \Rightarrow \varphi_2$, SPEN uses a sound procedure to deal with disjunctive formulas: it checks that for every disjunct ψ_1 in φ_1, there is a disjunct ψ_2 of φ_2 such that $\psi_1 \Rightarrow \psi_2$. The procedure for deciding the validity of entailments between symbolic heaps follows the workflows given in Fig. 2 and Fig. 3 (the theoretical foundations were established in [6,8]). The two formulas are first checked for satisfiability and normalized using the procedures from Section 3. If one of the two formulas is unsatisfiable, then the validity of the entailment can be already determined, e.g., if ψ_1 is unsatisfiable then the entailment is valid. When both formulas are satisfiable, SPEN offers two different procedures tuned for each fragment of $\mathsf{SL}^{\mathsf{ID}}$.

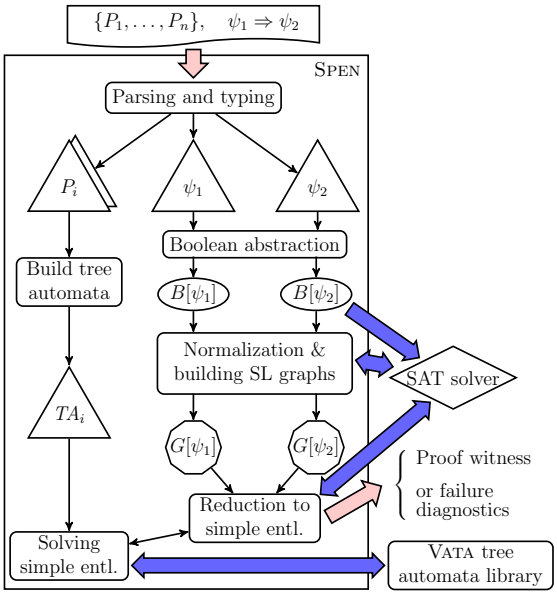

Fig. 2: SPEN workflow for entailment in $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$

For the fragment $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$, SPEN reduces the entailment problem $\psi_1 \Rightarrow \psi_2$ to a set of entailment queries of the form $\psi_1[a] \Rightarrow a$, called *simple entailments*, where $\psi_1[a]$ is a sub-formula of ψ_1 and a is a (points-to or inductive) spatial atom of ψ_2 (there will be one such entailment for each spatial atom a in ψ_2). Intuitively, the sub-formula $\psi_1[a]$ describes the region of a heap modelled by ψ_1 that should satisfy a. The procedures for computing $\psi_1[a]$ and testing simple entailments use an intermediary graph representation of symbolic heap formulas, called an *SL-graph* and denoted $G[\psi]$. Basically, nodes of $G[\psi]$ represent sets of aliased variables according to the pure part of ψ, and edges represent dis-equalities and spatial atoms of ψ, e.g., a spatial atom $P(X, Y, \vec{x})$ is represented by a directed edge from X to Y labeled by $P(\vec{x})$. Thus, when a is a predicate atom $P(X, Y, \vec{x})$, $\psi_1[a]$ is obtained from the SL-graph of ψ_1 by selecting the edges reachable from X and co-reachable from Y. The graph selected for $\psi_1[a]$ is transformed

Table 1: Experimental results on an Intel(R) Core(TM) i7-2600 CPU at 1.60GHz

Fragments				*Time [s]*		*SL-COMP'14 results*
$\mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$	$\mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$	*Benchmark*	*Size*	SPEN$_{\mathsf{L}}$	SPEN$_{\mathsf{D}}$	*Time [s] StarExec/solver*
✓	✓	`sll0_sat`	110	11.20	11.28	(I) 1.06 / Asterix, (II) 3.27 / SPEN
✓	✓	`sll0_entl`	292	34.45	34.94	(I) 2.98 / Asterix, (II) 7.58 / SPEN
✓	✓	`FDB_entl`	43	1.08	1.00	(I) 0.61 / SPEN, (II) 43.65 / SLEEK
✓		`FDB_entl`$^+$	55	0.65	—	—

into a tree t_1, which is tested for membership in the language of a tree automaton built from the rules defining P for the atom $a = P(X, Y, \vec{x})$.

For the fragment $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$, SPEN implements a proof search strategy for the entailment problem $\psi_1 \Rightarrow \exists \vec{d}.\ \psi_2$. The strategy computes a sequence of formulas $\exists \vec{d}^1.\ \psi_1^1, \ldots, \exists \vec{d}^n.\ \psi_1^n$ such that (1) $\exists \vec{d}^i.\ \psi_1^i \Rightarrow \exists \vec{d}^{i+1}.\ \psi_1^{i+1}$ and (2) $\exists \vec{d}^n.\ \psi_1^n$ is syntactically equivalent to $\exists \vec{d}.\ \psi_2$. The entailments in point (1) are obtained by applying the inductive rules and lemmas obtained automatically thanks to restriction required on inductive definitions. The procedure requires to check entailments between data constraints, which is done using the previously mentioned wrapper to the SMT solver.

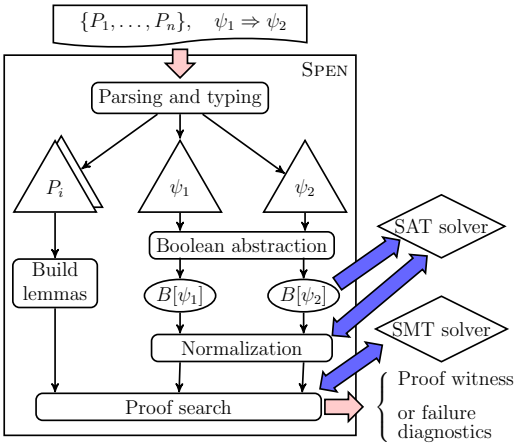

Fig. 3: SPEN workflow for entailment in $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$

For both procedures, when the input entailment $\psi_1 \Rightarrow \exists \vec{d}.\ \psi_2$ holds, SPEN has the option of providing a proof witness that either indicates the fact that ψ_1 is unsatisfiable or it consists of the normalized forms of ψ_1 and ψ_2 and the mapping of sub-fomulas in ψ_1 to atoms of ψ_2. When the input entailment is not valid and the procedure terminates, SPEN provides a diagnosis that explains why the entailment fails.

5 Experimental Results

SPEN has been applied to a benchmark of 578 problems (available in the repository), 90 % obtained from verification conditions of iterative programs on complex dynamic data structures. The remaining problems are crafted to test the capabilities of the solver. Tables 1 and 2 provide an overview of results obtained by SPEN on this benchmark.

The benchmark of $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{L}}$ problems includes three divisions of SL-COMP'14: satisfiability and entailment problems for acyclic singly linked lists (`sll0_sat` resp. `sll0_entl`), and entailment checking for formulas describing more complicated types of linked lists, e.g., doubly-linked lists, skip lists, and nested lists (`FDB_entl`). SPEN spends less than 0.05 s on 90 % of the problems with the maximum time of 0.5 s; these times include calls to a SAT solver. The benchmark `FDB_entl`$^+$ includes the problems not in the SL-COMP'14 benchmark (e.g., formulas describing lists of cyclic lists). The reported times in the last column have been obtained in 2014 on the StarExec[5] platform.

[5] `www.starexec.org`, an Intel(R) Xeon(R) CPU E5-2609 at 2.40GHz of and 10 MB cache.

The benchmark of $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$ problems (see Table 2) includes verification conditions for proving the correctness of iterative procedures (delete, insert, search) over recursive data structures storing integer data: sorted lists, binary search trees, AVL trees, and red-black trees. SPEN spends less than 0.4 s on each problem, including calls to SAT and SMT solvers. The first three lines of Table 1 demonstrate that the two approaches implemented in SPEN (based on tree automata—column "SPEN_{L}"—and on proof search—column "SPEN_{D}") are not only complementary but also comparable on the common fragment. The improvements discussed in this paper reduce the execution times by 10 % within the division `sll0_entl` and by 30 % within `FDB_entl` w.r.t. the old version [6].

Table 2: Results for $\mathsf{SL}^{\mathsf{ID}}_{\mathsf{D}}$

Benchmark	*Size*	*Time [s]*
`sll0_sorted`	16	0.45
`BST`	45	1.67
`AVL`	22	1.21
`RBT`	21	3.61

Acknowledgement. This work was supported by the French ANR project Vecolib, the Czech Science Foundation (project 17-12465S), the BUT FIT project FIT-S-17-4014, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

References

1. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *Proc. FSTTCS'04*, LNCS 3328, Springer, 2004.
2. J. Brotherston, N. Gorogiannis, and R. L. Petersen. A Generic Cyclic Theorem Prover. In *Proc. of APLAS'12*, LNCS 7705, Springer, 2012.
3. C. Calcagno, H. Yang, P. O'Hearn. Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In *Proc. of FSTTCS'01*, LNCS 2245, Springer, 2001.
4. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Science of Computer Programming*, 77(9):1006–1036, Elsevier, 2012.
5. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS'08*, LNCS 4963, Springer, 2008.
6. C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. Compositional Entailment Checking for a Fragment of Separation Logic. In *Proc. of APLAS'14*, LNCS 8858, Springer, 2014.
7. C. Enea, V. Saveluc, and M. Sighireanu. Compositional Invariant Checking for Overlaid and Nested Linked Lists. In *Proc. of ESOP'13*, LNCS 7792, Springer, 2013.
8. C. Enea, M. Sighireanu, and Z. Wu. On Automated Lemma Generation for Separation Logic with Inductive Definitions. In *Proc. of ATVA'15*, LNCS 9364, Springer, 2015.
9. R. Iosif, A. Rogalewicz, and T. Vojnar. Deciding Entailments in Inductive Separation Logic with Tree Automata. In *Proc. of ATVA'14*, LNCS 8837, Springer, 2014.
10. R. Iosif, A. Rogalewicz, and J. Šimáček. The Tree Width of Separation Logic with Recursive Definitions. In *Proc. of CADE-24*, LNCS 7898, Springer, 2013.
11. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Proc. of CSL'01*, LNCS 2142, Springer, 2001.
12. J. A. N. Pérez and A. Rybalchenko. Separation Logic Modulo Theories. In *Proc. of APLAS'13*, LNCS 8301, Springer, 2013.
13. R. Piskac, T. Wies, and D. Zufferey. Automating Separation Logic Using SMT. In *Proc. of CAV'13*, LNCS 8044, Springer, 2013.
14. X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural Proofs for Structure, Data, and Separation. In *Proc. of PLDI'13*, ACM Press, 2013.
15. M. Sighireanu and D. Cok. Report on SL-COMP'14. *JSAT*, 9:173–186, 2014.