Encoding Separation Logic in SMT-LIB v2.5

Radu Iosif[2], Cristina Serban[2], Andrew Reynolds[1], and Mihaela Sighireanu[3]

[1] The University of Iowa
[2] Verimag/CNRS/Université de Grenoble Alpes
[3] IRIF/Université Paris Diderot

Abstract. We propose an encoding of Separation Logic using SMT-LIB v2.5. This format is currently supported by SMT solvers (CVC4) and inductive proof-theoretic solvers (SLIDE and SPEN). Moreover, we provide a library of benchmarks written using this format, which stems from the set of benchmarks used in SL-COMP'14 [**?**].

1 Preliminaries

We consider formulae in multi-sorted first-order logic. A *signature* Σ consists of a set Σ^{s} of sort symbols and a set Σ^{f} of *function symbols* $f^{\sigma_1 \cdots \sigma_n \sigma}$, where $n \geq 0$ and $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma^{\mathrm{s}}$. If $n = 0$, we call f^σ a *constant symbol*. We make the following assumptions:

1. all signatures Σ contain the Boolean sort B, where $\top$ and $\bot$ denote the Boolean constants *true* and *false*.
2. Σ^{f} contains a boolean equality function $\approx^{\sigma\sigma\mathsf{B}}$ for each sort symbol $\sigma \in \Sigma^{\mathrm{s}}$.

Let Vars be a countable set of first-order variables, each $x^\sigma \in \mathsf{Vars}$ having an associated sort σ. First-order terms and formulae over the signature Σ (called Σ-terms and Σ-formulae) are defined as usual. A first-order variable is *free* if it does not occur within the scope of a quantifier, and we write $\varphi(\mathbf{x})$ to denote that the free variables of the formula φ belong to the set $\mathbf{x}$.

A Σ*-interpretation* $\mathcal{I}$ maps:

- each sort symbol $\sigma \in \Sigma$ to a non-empty set $\sigma^{\mathcal{I}}$,
- each function symbol $f^{\sigma_1,\ldots,\sigma_n,\sigma} \in \Sigma$ to a total function $f^{\mathcal{I}} : \sigma_1^{\mathcal{I}} \times \ldots \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ where $n > 0$, and to an element of $\sigma^{\mathcal{I}}$ when $n = 0$, and
- each variable $x^\sigma \in \mathsf{Vars}$ to an element of $\sigma^{\mathcal{I}}$.

For an interpretation $\mathcal{I}$ a sort symbol σ and a variable x, we denote by $\mathcal{I}[\sigma \leftarrow S]$ and, respectively $\mathcal{I}[x \leftarrow v]$, the interpretation associating the set S to σ, respectively the value v to x, and which behaves like $\mathcal{I}$ in all other cases. By writing $\mathcal{I}[\sigma \leftarrow S]$ we ensure that all variables of sort σ are mapped by $\mathcal{I}$ to elements of S. For a Σ-term t, we write $t^{\mathcal{I}}$ to denote the interpretation of t in $\mathcal{I}$, defined inductively, as usual. A satisfiability relation between Σ-interpretations and Σ-formulas, written $\mathcal{I} \models \varphi$, is also defined inductively, as usual. In this case, we say that $\mathcal{I}$ is a *model* of φ.

A (multi-sorted first-order) *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a non-empty set of Σ-interpretations, the *models* of T. A Σ-formula φ is T*-satisfiable* if it is satisfied by some interpretation in $\mathbf{I}$.

2 Ground Separation Logic

Let $T = (\Sigma, \mathbf{I})$ be a theory and let Loc and Data be two sorts from Σ, with no restriction other than the fact that Loc is always interpreted as a countable set. Also, we consider that Σ has a designated constant symbol $\mathsf{nil}^{\mathsf{Loc}}$. We define the *Ground Separation Logic* $\mathsf{SL}(T)^{g}_{\mathsf{Loc},\mathsf{Data}}$ to be the set of formulae generated by the following syntax:

$$\varphi := \phi \mid \mathsf{emp} \mid \mathsf{t} \mapsto \mathsf{u} \mid \varphi_1 * \varphi_2 \mid \varphi_1 \twoheadrightarrow \varphi_2 \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \exists x^{\sigma} \,.\, \varphi_1(x)$$

where ϕ is a Σ-formula, and t, u are Σ-terms of sorts Loc and Data, respectively. As usual, we write $\forall x^{\sigma} \,.\, \varphi(x)$ for $\neg\exists x^{\sigma} \,.\, \neg\varphi(x)$. We omit specifying the sorts of variables and functions when they are clear from the context.

Given an interpretation $\mathcal{I}$, a *heap* is a finite partial mapping $h : \mathsf{Loc}^{\mathcal{I}} \rightharpoonup_{\text{fin}} \mathsf{Data}^{\mathcal{I}}$. For a heap h, we denote by $\mathrm{dom}(h)$ its domain. For two heaps h_1 and h_2, we write $h_1 \# h_2$ for $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$ and $h = h_1 \uplus h_2$ for $h_1 \# h_2$ and $h = h_1 \cup h_2$. We define the *satisfaction relation* $\mathcal{I}, h \models_{\mathsf{SL}} \phi$ inductively, as follows:

$$
\begin{array}{ll}
\mathcal{I}, h \models_{\mathsf{SL}} \phi & \iff \mathcal{I} \models \phi \text{ if } \phi \text{ is a } \Sigma\text{-formula} \\
\mathcal{I}, h \models_{\mathsf{SL}} \mathsf{emp} & \iff h = \emptyset \\
\mathcal{I}, h \models_{\mathsf{SL}} \mathsf{t} \mapsto \mathsf{u} & \iff h = \{(\mathsf{t}^{\mathcal{I}}, \mathsf{u}^{\mathcal{I}})\} \text{ and } \mathsf{t}^{\mathcal{I}} \not\approx \mathsf{nil}^{\mathcal{I}} \\
\mathcal{I}, h \models_{\mathsf{SL}} \phi_1 * \phi_2 & \iff \text{there exist heaps } h_1, h_2 \text{ s.t. } h = h_1 \uplus h_2 \text{ and } \mathcal{I}, h_i \models_{\mathsf{SL}} \phi_i, i = 1, 2 \\
\mathcal{I}, h \models_{\mathsf{SL}} \phi_1 \twoheadrightarrow \phi_2 & \iff \text{for all heaps } h' \text{ if } h' \# h \text{ and } \mathcal{I}, h' \models_{\mathsf{SL}} \phi_1 \text{ then } \mathcal{I}, h' \uplus h \models_{\mathsf{SL}} \phi_2 \\
\mathcal{I}, h \models_{\mathsf{SL}} \exists x^S . \varphi(x) & \iff \mathcal{I}[x \leftarrow s], h \models_{\mathsf{SL}} \varphi(x), \text{ for some } s \in S^{\mathcal{I}}
\end{array}
$$

The satisfaction relation for Σ-formulae, Boolean connectives $\wedge$, $\neg$, and linear arithmetic atoms, are the classical ones from first-order logic. Notice that the range of a quantified variable x^S is the interpretation of its associated sort $S^{\mathcal{I}}$. A formula φ is said to be *satisfiable* if there exists an interpretation $\mathcal{I}$ and a heap h such that $\mathcal{I}, h \models_{\mathsf{SL}} \varphi$. We say that φ *entails* ψ, written $\varphi \models_{\mathsf{SL}} \psi$, when every pair $(\mathcal{I}, h)$ which satisfies φ, also satisfies ψ.

2.1 SMT-LIB Encoding

We write ground SL formulae in SMT-LIB using the following functions:

```
(par (Loc Data) (emp Loc Data Bool))
(sep Bool Bool Bool :left-assoc)
(wand Bool Bool Bool :right-assoc)
(par (Loc Data) (pto Loc Data Bool))
(par (Loc) (nil Loc))
```

Observe that `emp`, `pto` and `nil` are polymorphic functions, with sort parameters `Loc` and `Data`. There is no restriction on the choice of `Loc` and `Data`, as shown below. However, in addition to the classical SMT-LIB typing constraints, the SL theories require that the heap models are well-typed.

The type of heap models is fixed using a special command, not included in SMT-LIB, `declare-heap`. For example, assume that `Loc` is an uninterpreted sort U and `Data` is the integer sort Int. The following declarations fix the type of the heap model and some constant names:

```
(declare-sort U 0)

(declare-heap (U Int))

(declare-const x U)
(declare-const y U)
(declare-const a Int)
(declare-const b Int)
```

We write the SL formula $\mathsf{emp} \wedge ((x \mapsto a * y \mapsto b) \mathrel{-\!\!*} (x \mapsto \mathsf{nil} * \top))$ in SMT-LIB as follows:

```
(and (as emp U Int)
     (wand (sep (pto x a) (pto y b)) (sep (pto x (as nil Int)) true))
)
```

With the declarations above, a separation constraint of the form:

```
(sep (pto x y) (pto a b))
```

results in a typing error, because `(pto x y)` requires the heap to be of type $\mathsf{U} \rightharpoonup \mathsf{U}$, whereas `(pto a b)` requires the heap to be of type $\mathsf{Int} \rightharpoonup \mathsf{Int}$, and combining heaps of different types is not allowed.

This heap typing restriction is not a limitation of the expressive power of the SMT-LIB encoding and can be easily overcome by using datatypes (available in SMT-LIB v2.5). Suppose, for instance that we would like to specify a heap consisting of cells containing both integer and boolean data. The idea is to declare a union type:

```
(declare-datatype BoolInt ((cons_bool (d Bool)) (cons_int (d Int))))

(declare-heap (U BoolInt))
```

and use it to describe a mixed data heap, as in:

```
(sep (pto x (cons_bool false)) (pto y (cons_int 0)))
```

The extension of the heap typing with typed locations is presented in Section **??**.

2.2 Separation Logic with Inductive Definitions

Let Pred be a set of second-order variables, each $P^{\sigma_1 \ldots \sigma_n} \in \mathsf{Pred}$ having an associated tuple of parameter sorts $\sigma_1, \ldots, \sigma_n \in \Sigma^{\mathrm{s}}$. In addition to the first-order terms built using variables from Vars and function symbols from Σ^{f}, we enrich the language of SL with the boolean terms $P^{\sigma_1 \ldots \sigma_n}(t_1, \ldots, t_n)$, where each t_i is a first-order term of sort σ_i, for $i = 1, \ldots, n$. Each second-order variable $P^{\sigma_1 \ldots \sigma_n} \in \mathsf{Pred}$ is provided with an inductive definition $P(x_1, \ldots, x_n) \leftarrow \phi_P(x_1, \ldots, x_n)$, where ϕ_P is a formula in the extended language, possibly containing occurrences of P. The satisfaction relation is then extended as follows:

$$\mathcal{I}, h \models_{\mathsf{SL}} P^{\sigma_1 \ldots \sigma_n}(t_1, \ldots, t_n) \iff \mathcal{I}, h \models_{\mathsf{SL}} \phi_P(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$$

where ϕ_P is the inductive definition of $P^{\sigma_1 \ldots \sigma_n}$. Observe that, given a set of inductive definitions, the set of possible models for each second-order variable is the least fixed point of a monotonic and continuous function mapping tuples of sets of models to a set of models.

2.3 SMT-LIB Encoding

An inductive definition $P(x_1, \ldots, x_n) \leftarrow \phi_P(x_1, \ldots, x_n)$ is written in SMT-LIB using a recursive function definition. For instance, the inductive definition of a doubly-linked list segment:

$$\begin{aligned} \mathsf{dllseg}(h, p, t, n) \leftarrow & (\mathsf{emp} \wedge h \approx n \wedge p \approx t) \vee \\ & (\exists x^{\mathsf{Loc}} \,.\, h \mapsto (x, p) * \mathsf{dllseg}(x, h, t, n)) \end{aligned}$$

is written into SMT-LIB as follows:

```
(declare-datatype Node ((node (next Loc) (prev Loc))))

(declare-heap (Loc Node))

(define-fun-rec dllseg ((h Loc) (p Loc) (t Loc) (n Loc)) Bool
    (or (and emp (= h n) (= p t))
        (exists ((x Loc)) (sep (pto h (node x p)) (dllseg x h t n)))
    )
)
```

2.4 A Detailed Example

Let us go through an example step by step. First of all, the preamble of and SMT-LIB file describing a SL satisfiability query must contain (at least):

```
(set-logic SEPLOG)
```

The fragments of this theory are defined in Section **??**. If SL is used in combination with other theories, it is customary to start with:

```
(set-logic ALL_SUPPORTED)
```

We consider the slightly modified version of the dllseg definition above, which describes a doubly-linked list segment with ordered integer data:

$$\begin{aligned} \mathsf{dllseg}_{ord}(h, p, t, n, min) \leftarrow & (\mathsf{emp} \wedge h \approx n \wedge p \approx t) \vee \\ & (\exists x^{\mathsf{Loc}} \exists d^{\mathsf{Int}} \,.\, h \mapsto (d, x, min) * \mathsf{dllseg}_{ord}(x, h, t, n, d)) \wedge min \leq d \end{aligned}$$

Since we do not perform any pointer arithmetic reasoning, we can declare Loc to be an uninterpreted sort:

```
(declare-sort Loc 0)
```

We encode the definition of dllseg_{ord} as:

```
(declare-datatype Node ((node (data Int) (next Loc) (prev Loc))))

(declare-heap (Loc Node))

(define-fun-rec dllseg_ord ((h Loc) (p Loc) (t Loc) (n Loc) (min Int)) Bool
    (or (and (as emp Loc Data) (= h n) (= p t))
        (exists ((x Loc) (d Int))
                (and
                     (sep (pto h (node x p)) (dllseg_ord x h t n))
                     (<= min d)
                )
        )
    )
)
```

Let us consider the problem of proving that a dllseg_{ord} to which a node is appended is again a dllseg_{ord}, provided that the data of the new node it smaller than the minimal element of the first dllseg_{ord}:

$$x \mapsto (m, u, v) * \mathsf{dllseg}_{ord}(u, x, z, t, n) \land m \leq n \models_{\mathsf{SL}} \mathsf{dllseg}_{ord}(x, y, z, t, m)$$

We encode this entailment problem as an assertion asking whether the negated problem is satisfiable:

```
(declare-const x U)
(declare-const y U)
(declare-const z U)
(declare-const u U)
(declare-const v U)
(declare-const t U)
(declare-const m Int)
(declare-const n Int)

(assert (not (implies
              (and (sep (pto x (node m u v)) (dllseg_ord u x z t n)) (<= m n))
              (dllseg_ord x y z t m)
             )
        )
)
```

The entailment holds when the assertion is unsatisfiable, which can be checked in the standard way, using `(check-sat)`. However, the dual problem:

```
(assert (not (implies
              (dllseg_ord x y z t m)
              (and (sep (pto x (node m u v)) (dllseg_ord u x z t n)) (<= m n))
)       )    )
```

is satisfiable, and the counter-model can be obtained in the standard way, using (get-model). Observe that the model of a satisfiable SL query consists of an interpretation of the constants and a specification of the heap.

To comply with the format of SL-COMP'14 [?], the entailment problems may also be encoded using two separate assertions:

```
(assert (dllseg_ord x y z t m))
(assert (not (and (sep (pto x (node m u v)) (dllseg_ord u x z t n)) (<= m n))
)       )
(check-sat)
```

3 Multi-Sorted Separation Logic

Until now, we considered only problems with one type of locations. However, the heap typing declaration allows to declare a union type by listing the pair of types for locations and the corresponding heap cells.

For example, consider a heap storing a nested list. Locations in the inner lists are typed by RefList and the heap cells at these locations, typed by List, are linked by one field:

```
(declare-sort RefList 0)
(declare-datatype List ((c_list (next RefList))))
```

The heap cells of the upper list are typed by Nll and store a pair of locations, one of type RefList to the inner list, and a location of a same type of cell, typed by RefNll:

```
(declare-sort RefNll 0)
(declare-datatype Nll ((c_nll (next RefNll) (down RefList))))

(declare-heap (RefNll Nll) (RefList List))
```

A heap containing two cell is specified by:

```
(declare-const x RefNll)
(declare-const y RefList)

(assert (sep (pto x (c_nll (as nil RefNll) y))
             (pto z (c_list (as nil RefList)))
             (_ emp RefList List))
)
```

The empty heap is typed by one of the pairs of the union type declared for the heap.

4 Abduction and Frame Inference

Abduction and frame inference (or bi-abduction for both) are problems that occur in the context of program verification. In this case, the solver is not only required to give a yes/no answer to a satisfiability query, but to infer SL formulae that ensure the validity of a given entailment. Given SL formulae $\varphi(\mathbf{x})$ and $\phi(\mathbf{y})$, and second-order variables $X(\mathbf{x},\mathbf{y})$ and $Y(\mathbf{x},\mathbf{y})$, we consider the following synthesis problems:

1. The *abduction problem* asks for a satisfiable definition of a X such that $\varphi(\mathbf{x}) * X(\mathbf{x},\mathbf{y}) \models_{\mathsf{SL}} \psi(\mathbf{y})$. Sometimes X is called an *anti-frame*. Observe that $X \leftarrow \bot$ is always a solution, but not a very interesting one.
2. The *frame inference problem* asks for a definition of Y such that $\varphi(\mathbf{x}) \models_{\mathsf{SL}} \exists \mathbf{z} \,.\, \psi(\mathbf{y}) * Y(\mathbf{x},\mathbf{y})$, where $\mathbf{z} = \mathbf{y} \setminus \mathbf{x}$.
3. The *bi-abduction problem* asks for both a satisfiable definition of X and a definition of Y such that $\varphi(\mathbf{x}) * X(\mathbf{x},\mathbf{y}) \models_{\mathsf{SL}} \psi(\mathbf{y}) * Y(\mathbf{x},\mathbf{y})$.

The capability of solving the above problems is key to using a given SL solver for practical program verification purposes. For this reason, we aim at finding a standard way of specifying these problems in SMT-LIB.

5 Logics

The benchmarks of SL-COMP refer to one of the sub-logics of the many-sorted Separation Logic. These sub-logics identify fragments of the main logic for which have been identified efficient techniques for checking satisfiability and entailment.

The sub-logics are named using groups of letters, in a similar way that SMT-LIB. These letters have been chosen to evoke the restrictions used by the sub-logics:

- `QF` for the restriction to quantifier free formulas;
- `SH` for the symbolic heap fragment where formulas are conjunction of atoms and don't constraints ϕ and magic wand;
- `LS` where the only inductively defined predicate is the acyclic list segment, `ls`;
- `BI` for the fragment with magic wand atoms;
- `ID` for the fragment with user defined predicates.

The following logics are used in the SL-COMP benchmark:

- `QF_SHLS` is the logic for the divisions `sll0a_sat` and `sll0a_entl` of SL-COMP'14. A formula in these scripts is a conjunction of pure and spatial atoms except magic wand and including list segment predicate atoms.
- `QF_SHID` is the logic for the divisions `UDP_sat`, `UDP_entl`, `FDP_sat` and `FDP_entl` of SL-COMP'14. The scripts include inductive definitions of predicates and formulas that are conjunctions of aliasing, points-to and predicate atoms.
- `QF_BI` corresponds mainly to the logic defined in CVC4 [**?**], where formulas are quantifier free and boolean combinations of pure and spatial including magic wand; the scripts do not include inductive definitions and the heap type is only one pair of location and data sorts.

6 Additional Resources

The quest for a suitable format for SL solvers started with SL-COMP'14 [**?**], which adopted the `QF_S` format, described in [**?**]. The current proposal is inspired by `QF_S`, and relies on the datatypes introduced SMT-LIB v2.5 for an elegant treatment of union and record types. The tools supporting SMT-LIB as a native language are:

- CVC4 [**?**] – a description of the SL format of CVC4 is provided in [**?**] (a slightly modified version of the current proposal)
- SLIDE (under construction) – uses the encoding from the current proposal.
- SPEN [**?**] – a description of the SL format of SPEN (QF_S) is available in [**?**].

Other tools that participated to SL-COMP'14 have been adapted to QF_S by means of a specialized front-end [**?**]. It is our goal to convince the developers of SL solvers to adopt SMT-LIB as the native input language of their tools, rather than use a translator from SMT-LIB. For this purpose, we provide a C++ front-end [**?**] that can be used to parse and type check SL inputs encoded in SMT-LIB using the current specification.