Encoding Separation Logic in SMT-LIB v2.5

Radu Iosif\textsuperscript{2}, Cristina Serban\textsuperscript{2}, Andrew Reynolds\textsuperscript{1}, and Mihaela Sighireanu\textsuperscript{3}

\textsuperscript{1} The University of Iowa
\textsuperscript{2} Verimag/CNRS/Universit\'{e} de Grenoble Alpes
\textsuperscript{3} IRIF/Universit\'{e} 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 [7].

1 Preliminaries

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

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

Let $\text{Vars}$ be a countable set of first-order variables, each $x^\sigma \in \text{Vars}$ having an associated sort $\sigma$. First-order terms and formulae over the signature $\Sigma$ (called $\Sigma$-terms and $\Sigma$-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 $\varphi(x)$ to denote that the free variables of the formula $\varphi$ belong to the set $x$.

A $\Sigma$-interpretation $I$ maps:

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

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

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

2 Ground Separation Logic

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

\[ \varphi ::= \phi \mid \text{emp} \mid t \mapsto u \mid \varphi_1 \ast \varphi_2 \mid \varphi_1 \# \varphi_2 \mid \neg \varphi_1 \mid \varphi_1 \land \varphi_2 \mid \exists x^\tau . \varphi_1(x) \]

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

Given an interpretation \( I \), a heap is a finite partial mapping \( h : \text{Loc} \rightarrow \text{Data} \).

For a heap \( h \), we denote by \( \text{dom}(h) \) its domain. For two heaps \( h_1 \) and \( h_2 \), we write \( h_1 \# h_2 \) for \( \text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset \) and \( h = h_1 \uplus h_2 \) for \( h_1 \# h_2 \) and \( h = h_1 \cup h_2 \). We define the satisfaction relation \( I, h \models_{\text{sl}} \varphi \) inductively, as follows:

\[
\begin{align*}
I, h \models_{\text{sl}} \varphi & \iff I \models \varphi \text{ if } \varphi \text{ is a } \Sigma \text{-formula} \\
I, h \models_{\text{sl}} \text{emp} & \iff h = \emptyset \\
I, h \models_{\text{sl}} t \mapsto u & \iff h = (t^f, u^f) \text{ and } t^f \neq \text{nil}^f \\
I, h \models_{\text{sl}} \varphi_1 \ast \varphi_2 & \iff \exists h_1, h_2 \text{ s.t. } h = h_1 \uplus h_2 \text{ and } I, h_1 \models_{\text{sl}} \varphi_i, i = 1, 2 \\
I, h \models_{\text{sl}} \varphi_1 \# \varphi_2 & \iff \forall h' \text{ if } h' \# h \text{ and } I, h' \models_{\text{sl}} \varphi_1 \text{ then } I, h' \uplus h \models_{\text{sl}} \varphi_2 \\
I, h \models_{\text{sl}} \exists x^S . \varphi(x) & \iff I[x \leftarrow s], h \models_{\text{sl}} \varphi(x), \text{ for some } s \in S^f
\end{align*}
\]

The satisfaction relation for \( \Sigma \)-formulae, Boolean connectives \( \land, \neg \), 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 \( S^f \). A formula \( \varphi \) is said to be satisfiable if there exists an interpretation \( I \) and a heap \( h \) such that \( I, h \models_{\text{sl}} \varphi \). We say that \( \varphi \) entails \( \psi \), written \( \varphi \models_{\text{sl}} \psi \), when every pair \( (I, h) \) which satisfies \( \varphi \), also satisfies \( \psi \).

2.1 SMT-LIB Encoding

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

\[
\begin{align*}
\text{(par (Loc Data) (emp Loc Data Bool))} \\
\text{(sep Bool Bool Bool :left-associ) } \\
\text{(wand Bool Bool Bool :left-associ) } \\
\text{(par (Loc Data) (pto Loc Data Bool))} \\
\text{(par (Loc) (nil Loc))}
\end{align*}
\]

Observe that \text{emp}, \text{pto} and \text{nil} are polymorphic functions, with sort parameters \text{Loc} and \text{Data}. There is no restriction on the choice of \text{Loc} and \text{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 `emp ∧ ((x ↦ a ⇨ y) ↨ (x ↦ nil ∧ T))` 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 `U ↨ U`, whereas `(pto a b)` requires the heap to be of type `Int ↨ 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 3.

### 2.2 Separation Logic with Inductive Definitions

Let `Pred` be a set of second-order variables, each `P^σ_1...σ_n ∈ Pred` having an associated tuple of parameter sorts `σ_1,...,σ_n ∈ Σ`. In addition to the first-order terms built using variables from `Vars` and function symbols from `Σ`, we enrich the language of SL with the boolean terms `P^σ_1...σ_n(t_1,...,t_n)`, where each `t_i` is a first-order term of sort `σ_i`, for `i = 1,...,n`. Each second-order variable `P^σ_1...σ_n ∈ Pred` is provided with an inductive
definition \( P(x_1, \ldots, x_n) \leftarrow \phi_P(x_1, \ldots, x_n) \), where \( \phi_P \) is a formula in the extended language, possibly containing occurrences of \( P \). The satisfaction relation is then extended as follows:

\[
I, h \models_{\mathit{SL}} P^\tau_1, \ldots, ^\tau_n(t_1, \ldots, t_n) \iff I, h \models_{\mathit{SL}} \phi_P(t_1^I, \ldots, t_n^I)
\]

where \( \phi_P \) is the inductive definition of \( P^\tau_1, \ldots, ^\tau_n \). 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:

\[
dllseg(h, p, t, n) \leftarrow (\text{emp} \land h \approx n \land p \approx t) \lor
\]

\[
(\exists \text{Loc}. \ h \mapsto (x, p) \ast dllseg(x, h, t, n))
\]

is written into SMT-LIB as follows:

```smt-lib
(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 \( \mathit{SL} \) satisfiability query must contain (at least):

```smt-lib
(set-logic SEPLOG)
```

The fragments of this theory are defined in Section 5. If \( \mathit{SL} \) is used in combination with other theories, it is customary to start with:

```smt-lib
(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:

\[
dllseg_{\text{ord}}(h, p, t, n, \text{min}) \leftarrow (\text{emp} \land h \approx n \land p \approx t) \lor
\]

\[
(\exists \text{Loc} \exists \text{Int}. \ h \mapsto (d, x, \text{min}) \ast dllseg_{\text{ord}}(x, h, t, n, d)) \land \text{min} \leq d
\]

Since we do not perform any pointer arithmetic reasoning, we can declare \( \text{Loc} \) to be an uninterpreted sort:
We encode the definition of \( \text{dllseg}_\text{ord} \) as:

\[
\begin{align*}
\text{(declare-datatype Node ((node (data Int) (next) (prev)) (prev Loc) (next Loc)))} \\
\text{(declare-heap (Loc Node))} \\
\text{(define-fun-rec dllseg_ord ((h Loc) (p Loc) (t Loc) (n Loc) (min Int)) Bool} \\
\text{ (or (and (as emp Loc Data) (= h n) (= p t))} \\
\text{ (exists ((x Loc) (d Int))} \\
\text{ (and} \\
\text{ (sep (pto h (node x p))) (dllseg_ord x h t n))} \\
\text{ (<= min d))} \\
\text{))} \\
\end{align*}
\]

Let us consider the problem of proving that a \( \text{dllseg}_\text{ord} \) to which a node is appended is again a \( \text{dllseg}_\text{ord} \), provided that the data of the new node is smaller than the minimal element of the first \( \text{dllseg}_\text{ord} \):

\[
x \mapsto (m, u, v) \ast \text{dllseg}_\text{ord}(u, x, z, t, n) \land m \leq n \models \text{dllseg}_\text{ord}(x, y, z, t, m)
\]

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

\[
\begin{align*}
\text{(declare-const x U)} \\
\text{(declare-const y U)} \\
\text{(declare-const z U)} \\
\text{(declare-const u U)} \\
\text{(declare-const v U)} \\
\text{(declare-const t U)} \\
\text{(declare-const m Int)} \\
\text{(declare-const n Int)} \\
\text{(assert (not (implies} \\
\text{ (and (sep (pto x (node m u v))) (dllseg_ord u x z t n)) (<= m n))} \\
\text{ (dllseg_ord x y z t m))}) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{(assert (not (implies} \\
\text{ (dllseg_ord x y z t m))} \\
\text{ (and (sep (pto x (node m u v))) (dllseg_ord u x z t n)) (<= m n))}) \\
\end{align*}
\]
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 [7], 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 \( \varphi(x) \) and \( \phi(y) \), and second-order variables \( X(x, y) \) and \( Y(x, y) \), we consider the following synthesis problems:

1. The **abduction problem** asks for a satisfiable definition of a \( X \) such that \( \varphi(x) \models_{\text{SL}} X(x, y) =_{\text{SL}} \psi(y) \). 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 \( \varphi(x) \models_{\text{SL}} \exists z. \psi(y) + Y(x, y) \), where \( z = y \setminus x \).
3. The **bi-abduction problem** asks for both a satisfiable definition of \( X \) and a definition of \( Y \) such that \( \varphi(x) \models_{\text{SL}} X(x, y) =_{\text{SL}} \psi(y) + Y(x, y) \).

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 \( \phi \) and magic wand;
- **LS** where the only inductively defined predicate is the acyclic list segment, \( \text{lsl} \);
- **ID** for the fragment with user defined predicates;
- **LID** for the fragment of linear user defined predicates, i.e., inly one recursive call by definition;
- **BI** for the fragment with magic wand atoms.

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 [2], 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 [7], which adopted the \( \text{QF}_S \) format, described in [6]. The current proposal is inspired by \( \text{QF}_S \), 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 [3] – a description of the SL format of CVC4 is provided in [2] (a slightly modified version of the current proposal)
- SLIDE (under construction) – uses the encoding from the current proposal.
- SPEN [1] – a description of the SL format of SPEN (\( \text{QF}_S \)) is available in [6].

Other tools that participated to SL-COMP’14 have been adapted to \( \text{QF}_S \) by means of a specialized front-end [5]. 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 [4] that can be used to parse and type check SL inputs encoded in SMT-LIB using the current specification.

References