A Proposal for a Theory of Separation Logic Theory in SMT-LIB

SMT-LIB 4 SL Working Group

Draft of December 19, 2014

Abstract. Dynamic data structures are used in most programs. Separation Logic is an established Hoare logic for imperative, heap-manipulating programs. The program analysis tools dealing with programs annotated with Separation Logic specifications need to decide verification conditions over Separation Logic formulas. We propose a new theory for the SMT-Lib standard as the standard format for such formulae.

Keywords: Separation Logic, SMT-LIB, SAT Modulo Theory

1 Motivation

Separation Logic (SL) is an established and fairly popular Hoare logic for imperative, heap-manipulating programs, introduced nearly fifteen years ago by Reynolds [8,7,9]. Its high expressivity, its ability to generate compact proofs, and its support for local reasoning motivated the development of tools for automatic reasoning about programs using SL. For a rather exhaustive list of the past and present tools, see the web site [6].

These tools seek to establish memory safety properties and/or infer shape properties of the heap at a scale of millions of lines of code. They intensively use (semi-)decision procedures for checking satisfiability and entailment problems in SL. In the last five years, several papers reported on the design and implementation of such (semi-)decision procedures and compared publicly available tools. Moreover, the first competition of SL solvers, SL-COMP'14 [10], has been held in 2014 as an "off" (unofficial) event [1] associated with the SMT-COMP 2014 competition [11], at the FLoC Olympic Games.

The benchmarks of SL-COMP'14 were collected in the input format submitted by the participants. This set of problems was translated into a common format designed like a theory of the SMT-LIB format[2]. That is, they used the

[1] That is, the competition was executed in conjunction with the games by the SMT-COMP organizing committee, SMT-COMP being an official participant in the games; the results of SL-COMP 2014 were reported at the SMT-2014 workshop at FLoC; however, SL-COMP 2014 was organized too late and was too experimental to be an official part of the FLoC Olympic Games.

[2] `www.smtlib.org`

syntax of SMT-LIBv2, although the SL theory underlying the syntax is not an official SMT-LIB theory or, at this point, even compatible with the theory underlying SMT-LIB. This format is presented and commented in this document (Section 3).

The standardisation of formats in logic has played a major role in accelerating research in the past. We think that having a standard format for SL will have a similar effect. For this reason, we propose a way to integrate an SL theory in the SMT-LIB format. The rationale for the choice of SMT-LIB as background format is that SL is combined with or translated into first-order theories that are or will be supported by the SMT-LIB format, e.g., [3]. Moreover, most of the verification tools are based on a multi-sorted version of SL with inductively defined predicates. SMT-LIB is multi-sorted and allows the definition of recursive predicates (in version 2.5).

TODO: Discuss FO constraint of SMT-LIB.
TODO: Discuss pro and cons for using the set theory and reachability constraints.

This document is structured as follows. Section 2 presents the abstract syntax and the semantics of a fragment of Separation Logic that is standardized by the new SMT-LIB theory. The theory used for SL-COMP'14 is presented in Section 3; we also discuss its advantages and drawbacks. Finally, Section 4 proposes a new theory for SL and provides some examples.

2 The Target Theory

The program analysis tools based on SL use different fragments or extensions of this logic. However, most of these fragments have as common factor for the specification of the heap the *symbolic heaps* fragment, also known as the Separation Logic with Recursive Definitions (SLRD) [5] or the positive flat SL fragment [1]. In the following, we call this fragment SLRD.

The fragment specifies configurations of programs manipulating variables that are references to record types. The records are defined by the user as a set of fields typed as reference or data. Such configurations are modeled by (i) a heap that is a set of records and (ii) a stack that maps program variables to record addresses. When the program and the record types include variables resp. fields in some numerical domain, the model is extended to represent such data.

The SLRD fragment includes four atoms to specify the heap, also called *spatial atoms*: (i) the empty heap, (ii) any heap (unspecified), (iii) a heap consisting of one allocated record, and (iv) an unbounded heap segment corresponding to a data structure whose shape is defined inductively using a *recursive definition*. Examples of such recursive definitions are provided in Table 1. These atoms are connected via a separating conjunction primitive $*$. Only the existential quantification is allowed and the use of disjunction and negation is restricted.

When data fields and variables are used in the program, the recursive definitions are extended with data parameters or collections (sets, multisets) over

such data. The data parameters are constrained inside the recursive definitions using a catamorphic schema [12].

An important feature of this fragment is its *flatness*: a formula has the following form:

$$\bigvee_i \exists \boldsymbol{X}_i.\, \Sigma_i \wedge \Pi_i \wedge \Delta_i$$

where the disjuncts are built from a the *spatial part* Σ_i, combining by $*$ the spatial atoms, the *pure part* Π_i, built as a conjunction of equalities and disqualities between reference variables, and the data constraints Δ_i. This flat form is not mandatory, but it usually facilitates the decision procedures.

Syntax: More formally, the syntax of formulas in the SLRD fragment of Separation Logic is given by the following grammar:

$f \in \mathbb{F}$ field names	$P \in \mathbb{P}$ recursive definition name	
$x, y \in \mathit{Vars}$ reference program vars	$X, Y \in \mathit{LVars}$ reference logical vars	
$d, D \in \mathit{DVars}$ data variables	Δ data constraints	
E, F	$::=\ x \mid X$	reference variables
ρ	$::=\ \{(f, E)\} \mid \{(f, D)\} \mid \rho \cup \rho$	set of field references
Π	$::=\ E = F \mid E \neq F \mid \Pi \wedge \Pi$	pure formulas
Σ	$::=\ \mathit{emp} \mid \mathit{junk} \mid E \mapsto \rho \mid P(\boldsymbol{E}, \boldsymbol{D}) \mid \Sigma * \Sigma$	spatial formulas
A, B	$\triangleq\ \exists \boldsymbol{X}, \boldsymbol{D}.\, \Pi \wedge \Sigma \wedge \Delta$	formulas

The program variable $\texttt{nil} \in \mathit{Vars}$ has a fixed meaning representing an undefined (not allocated) reference.

The fragment is parameterized by a set $\mathbb{P}$ of *recursive definitions* defined using the following syntax:

$$P(\boldsymbol{E}, \boldsymbol{D}) \triangleq \bigvee_i \exists \boldsymbol{X}_i, \boldsymbol{D}_i.\, \Pi_i \wedge \Sigma_i \wedge \Delta_i \tag{1}$$

where the spatial formulas Σ_i may call P or other predicates from $\mathbb{P}$. Table 1 gives several common examples of recursive data structures (without data constraints) definable using the syntax above.

Semantics: Let Loc be a set of locations and Val a set of data values. A stack $S : (\mathit{Vars} \cup \mathit{LVars} \to \mathit{Loc}) \cup (\mathit{DVars} \to \mathit{Val})$ maps reference variables to locations and data variables to values. A heap $H : \mathit{Loc} \times \mathbb{F} \rightharpoonup \mathit{Loc} \cup \mathit{Val}$ is a partial function that defines values of fields for some of the locations in Loc. The domain of H is denoted by $\mathit{dom}(H)$ and the set of locations in the domain of H is denoted by $\mathit{ldom}(H)$. As expected, $\texttt{nil}$ is interpreted to a location $S(\texttt{nil}) \notin \mathit{ldom}(H)$.

The set of configurations satisfying a formula φ is defined by the relation $(S, H) \models \varphi$ defined in Table 2 ($\uplus$ denotes the disjoint union of sets and $S[X \leftarrow \ell]$ denotes the function S' s.t. $S'(X) = \ell$ and $S'(Y) = S(Y)$ for any $Y \neq X$). Note that a configuration (S, H) satisfies a predicate atom $P(\boldsymbol{E})$ if it belongs to the

Comment
[MS1]: DNF

Comment
[MS2]: Give examples of translation to flat form.

Comment
[MS3]: Add loop?

Comment
[MS4]: Defs with data constraints.

singly linked lists

$$\texttt{ls}(E,F) \triangleq (E = F \wedge \mathit{emp}) \vee (E \neq F \wedge \exists X.\, E \mapsto \{(f,X)\} * \texttt{ls}(X,F)) \quad (2)$$

nested linked lists

$$\texttt{nll}(E,F,B) \triangleq (E = F \wedge \mathit{emp}) \vee (E \neq \{F,B\} \wedge \exists X, Z.\, E \mapsto \{(s,X),(h,Z)\} * \texttt{ls}(Z,B) * \texttt{nll}(X,F,B)) \quad (3)$$

doubly linked lists

$$\texttt{dll}(E,L,P,F) \triangleq (E = F \wedge L = P \wedge \mathit{emp}) \vee \big(E \neq F \wedge L \neq P \wedge \exists X.\, E \mapsto \{(n,X),(p,P)\} * \texttt{dll}(X,L,E,F)\ \big) \quad (4)$$

binary tree

$$\texttt{btree}(E) \triangleq (E = \texttt{nil} \wedge \mathit{emp}) \vee \big(E \neq \texttt{nil} \wedge \exists X, Y.\, E \mapsto \{(r,X),(l,Y)\} * \texttt{btree}(X) * \texttt{btree}(X)\big) \quad (5)$$

tree with linked leaves

$$\texttt{tll}(R,P,E,F) \triangleq (R = E \wedge R \mapsto \{(l,\texttt{nil}),(r,\texttt{nil}),(p,P),(n,F)\}) \vee \big(R \neq E \wedge \exists X, Y, Z.\, R \mapsto \{(l,X),(r,Y),(p,P),(n,Z)\} * \texttt{tll}(X,R,E,Z) * \texttt{tll}(Y,R,Z,F)\big) \quad (6)$$

Table 1. Examples of recursive definitions used in the benchmark

least fixed point of the set of recursive definitions $\mathbb{P}$ for the actual parameters $\boldsymbol{E}$ of P. The set of models of a formula φ is denoted by $[\![\varphi]\!]$. Given two formulas φ_1 and φ_2, we say that φ_1 entails φ_2, denoted by $\varphi_1 \Rightarrow \varphi_2$, iff $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$.

Notice that this semantics is a *precise* semantics. It was chosen because it is the most used in tools and in the verification of concurrent programs.

Comment [MS5]: Usually not commented in SMT-LIB theories.

Decidability and complexity properties:

3 The Theory at SL-COMP'14

The theory defined for SL-COMP'14 considers only the fragment of SLRD without data.

Name for the theory: The name `QF_S` has been used for the 2014 edition. It shall be reconsidered in the presence of a SMT-LIB theory of sets.

Records: The user has to declare the sorts corresponding to types *reference to record*. For example, consider the following C code defining a binary tree type:

```
typedef struct btree_s {
  struct btree_s* lson;
  struct btree_s* rson;
  }* btree_t;
```

$$
\begin{array}{ll}
(S,H) \models E = F & \text{iff } S(E) = S(F) \\
(S,H) \models E \neq F & \text{iff } S(E) \neq S(F) \\
(S,H) \models \varphi \wedge \psi & \text{iff } (S,H) \models \varphi \text{ and } (S,H) \models \psi \\
(S,H) \models \mathit{emp} & \text{iff } \mathit{dom}(H) = \emptyset \\
(S,H) \models \mathit{junk} & \text{always} \\
(S,H) \models E \mapsto \{\rho\} & \text{iff } \mathit{dom}(H) = \{(S(E), f_i) \mid (f_i, E_i) \in \{\rho\}\} \text{ and} \\
 & \quad \text{for every } (f_i, E_i) \in \{\rho\},\ H(S(E), f_i) = S(E_i) \\
(S,H) \models \Sigma_1 * \Sigma_2 & \text{iff } \exists H_1, H_2 \text{ s.t. } \mathit{ldom}(H) = \mathit{ldom}(H_1) \uplus \mathit{ldom}(H_2), \\
 & \quad (S,H_1) \models \Sigma_1, \text{ and } (S,H_2) \models \Sigma_2 \\
(S,H) \models P(\boldsymbol{E}) & \text{iff } (S,H) \in [\![\mathbb{P}]\!]\,(P(\boldsymbol{E})) \\
(S,H) \models \exists X.\, \varphi & \text{iff there exists } \ell \in \mathit{Loc} \text{ s.t. } (S[X \leftarrow \ell], H) \models \varphi
\end{array}
$$

Table 2. Semantics of the Separation Logic fragment

The sort for this type is declared as follows:

```
(declare-sort Btree_t () 0)
```

The SL theory defines the sort `Void` to denote the reference to any user record.

Remark 1. A solution that follows closely the C definition is (i) to consider that user declared sorts correspond to record types, and (ii) to define a parameterized sort in the theory corresponding to a reference type, e.g.:

```
(declare-sort Ref () 1)
```

This solution requires to use the `Ref` in most places where record sorts appear.

Fields: The set of field names, $\mathbb{F}$, is defined using the `Field` sort of arity 2 defined in the theory. Each field is declared as a function symbol of arity 0 and result type `Field A B` where `A` is the sort corresponding to the record type declaring the field and `B` is the sort typing the field. For example, the following code declares the fields of a binary tree with an integer field `data`:

```
(declare-fun lson () (Field Btree_t Btree_t))
(declare-fun rson () (Field Btree_t Btree_t))
(declare-fun data () (Field Btree_t Int))
```

Variables: The set of reference variables (program or existentially quantified) are declared in a classic way. For example:

```
(declare-fun root () Btree_t)
```

declares the variable `root` to be a reference to a binary tree.

The theory declares `nil` to be a special variable typed by the `Void` sort.

Flat formulas: The flatness of formulas is not ensured by the 2014 format. The format discourages the nesting of spatial and pure formulas by requiring heavy type casting. Indeed, the spatial atoms are typed in the theory by the `Space` sort (arity 0) and their combination with a pure (boolean) formula requires to cast `Space` to `Bool`. The space atoms are built using the following theory operators:

Abstr. syntax	SMT-LIB notation	SMT-LIB typing
emp	`emp`	`Space`
junk	`junk`	`Space`
$\Sigma_1 * \ldots * \Sigma_n$	`(ssep` $\langle form \rangle^+$ `)`	`(Space`$^+$ `Space)`
$E \mapsto \rho$	`(pto` $\langle var \rangle$ ρ`)`	`(par (A) (pto A (SetRef A) Space)`
none	`(tobool` $\langle form \rangle$ $\langle form \rangle$`)`	`(Space Bool)`
none	`(tospace` $\langle form \rangle$ $\langle form \rangle$`)`	`(Bool Space)`
$\{(f, E)\}$	`(ref` $\langle$`f`$\rangle$ $\langle var \rangle$`)`	`(par (A B) (ref (Field A B) B (SetRef A)))`
$\rho \cup \rho$	`(sref` $\langle \rho \rangle^+$`)`	`(par (A) (sref (SetRef A) (SetRef A) (SetRef A))`

For example, the following SLRD formula:

$$X \mapsto \{(lson, Y), (rson, Z)\} \wedge X \neq Y \tag{7}$$

is encoded in the SL theory in SMT-LIB as follows:

```
(and (tobool (pto (sref (ref lson Y) (ref rson Z))))
     (distinct X Y)
)
```

Remark 2. A solution to simplify the syntax is to replace the `Space` sort by the `Bool` sort in the above typing. This solution eliminates the cast operators `tobool` and `tospace`. The flatness of formulas could be then ensured by some transformations or by (sound) syntactic checks.

Recursive definitions: These definitions are introduced by the `define-fun-rec` operator.

Remark 3. The version 2.0 of SMT-LIB forbids the recursive calls in the term defining a function (`efine-fun`). The new version allows recursive calls introduced by `define-fun-rec` which defines a set of (mutually recursive) functions.

Because recursive definitions are spatial atoms, their result type shall be `Space`. The syntax of such definitions looks heavy because conversions are needed between space and boolean formula to be able to use the built-in boolean operators in SMT-LIB (exists, and, or). For example, the binary tree definition from Table 1 is encoded as follows:

```
(define-fun btree ((?root Btree_t)) Space (tospace
   (or (and (= ?root nil) (tobool emp))
```

```
       (exists ((?X Btree_t) (?Y Btree_t))
          (and (distinct ?root nil)
               (tobool (ssep (pto ?root (sref (ref lson ?Y) (ref rson ?Z)))
                             (btree ?X) (btree ?Y)
       ))))
)))
```

Remark 4. If the `Space` sort is eliminated, the recursive definitions are typed as boolean predicates. The definition above becomes:

```
(define-fun-rec btree ((?root Btree_t)) Bool
   (or (and (= ?root nil) emp)
       (exists ((?X Btree_t) (?Y Btree_t))
          (and (distinct ?root nil)
               (ssep (pto ?root (sref (ref lson ?Y) (ref rson ?Z)))
                     (btree ?X) (btree ?Y)
       )))
))
```

Notice the absence of two '`)`'!

4 Proposal for a Simplified Theory

This section defines an SMT-LIB theory for SLRD which changes as follows the SL-COMP'14 format:

- The unary sort `Ref` is introduced for reference types (cf. Remark 1).
- The sort `Space` is removed from the theory (cf. Remark 2).

Records: The user has to declare the sorts corresponding to *record* types. Then, a sort encoding a reference to record type is declared using the `Ref` sort. For example, the declaration of a binary tree variable is given by:

```
(declare-sort Btree_s () 0)
(declare-fun root () (Ref Btree_s))
```

The theory keeps the sort `Void` to denote the reference to any user record.

Fields: The set of field names, $\mathbb{F}$, is defined using the `Field` sort of arity 2. A field is declared as a 0-arity function of result typed by `Field A B` because `A` and `B` will be some `(Ref R)`. For example:

Comment
[MS6]: Impact on pto?

```
(declare-fun lson () (Field (Ref Btree_s) (Ref Btree_s)))
(declare-fun rson () (Field (Ref Btree_s) (Ref Btree_s)))
(declare-fun data () (Field (Ref Btree_s) Int))
```

Variables: The set of reference variable are declared as before but the type is always `(Ref A)` where `A` is some sort.

Flat formulas: Formulas are typed by `Bool`. The following operators are proposed:

Abstr. syntax	SMT-LIB notation	SMT-LIB typing
emp	`emp`	`Bool`
junk	`junk`	`Bool`
$\Sigma_1 * \ldots * \Sigma_n$	`(ssep` $\langle form \rangle^+$ `)`	`(Bool`$^+$ `Bool)`
$E \mapsto \rho$	`(pto` $\langle var \rangle$ ρ`)`	`(par (A) (pto A (SetRef A) Bool)`
$\{(f, E)\}$	`(ref` $\langle$`f`$\rangle$ $\langle var \rangle$`)`	`(par (A B) (ref (Field A B) B (SetRef A)))`
$\rho \cup \rho$	`(sref` $\langle \rho \rangle^+$`)`	`(par (A) (sref (SetRef A) (SetRef A) (SetRef A))`

For example, the formula (**??**) is encoded as follows:

```
(and (pto (sref (ref lson Y) (ref rson Z)))
     (distinct X Y)
)
```

TODO: discuss extension with a data theory.

References

1. Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *FOSSACS*, volume 8412 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2014.
2. James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. Technical Report RN/13/15, University College London, 2013.
3. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.
4. Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2011.
5. Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2013.
6. Peter O'Hearn. Separation logic. `www0.cs.ucl.ac.uk/staff/p.ohearn/SeparationLogic/Separation_Logic/SL_Home.html`.
7. Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
8. John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*. Palgrave Macmillan, 1999. Publication date November 2000.
9. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

10. SL-COMP'14. www.liafa.univ-paris-diderot.fr/~sighirea/slcomp14/.
11. SMT-COMP. smtcomp.sourceforge.org.
12. Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL*, pages 199–210. ACM, 2010.