Journal on Satisfiability, Boolean Modeling and Computation 1 (2014) XX-YY

# Report on SL-COMP 2014

**Mihaela Sighireanu** sighirea@liafa.univ-paris-diderot.fr
*LIAFA, University Paris Diderot & CNRS*

**David R. Cok** dcok@grammatech.com
*GrammaTech, Inc.,*

## Abstract

A competition of solvers for Separation Logic was held in May 2014, as an unofficial satellite event of the FLoC Olympic Games. Six solvers participated in the competition; the success and performance of each solver was measured over an appropriate subset of a library of benchmarks accumulated for the purpose. The benchmarks consisted of satisfiability and entailment problems over formulas in the fragment of symbolic heaps with inductive definitions, which is the fragment of Separation Logic that is most used in program analysis and verification tools. We report in this paper on the competition rules, the participants, the results, the findings, and the future of this event.

KEYWORDS: *Separation Logic, SMT-LIB, SAT Modulo Theory*

*Submitted December 12th, 2014; revised December 15, 2015; published*

## 1. Introduction

Separation Logic (SL) is an established and fairly popular Hoare logic for imperative, heap-manipulating programs, introduced nearly fifteen years ago by Reynolds [19, 15, 20]. Its high expressivity, its ability to generate compact proofs, and its support for local reasoning have 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 [14].

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 [11].

The organization of a public competition of SL solvers was an opportunity to collect the existing benchmarks and to make available the binaries of these solvers on a common platform, i.e., StarExec [26]. The SL-COMP 2014 has been possible due to the support of the SMT-COMP organizing committee, although SL is not a theory of the SMT-LIB format. The competition has been held as an "off" (unofficial) event[1.] associated with the SMT-COMP 2014 competition [24], at the FLoC Olympic Games.

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. Input Theory

The competition focused on a fragment of SL that has been found to be the kernel of most tools, known as the *symbolic heaps* fragment of Separation Logic with Inductive Definitions (SLID) [12] or the positive flat SL fragment [1]. We provide here only an informal description of this fragment, its detailed description could be found in, e.g., [20, 12, 1].

This fragment specifies the configurations of programs manipulating variables that are references to some record types; the records are defined by the user as a set of reference fields. Such configurations are modeled by (i) a heap that is a set of records and (ii) a stack that maps program variables into *locations* that are record addresses. In the following, we denote by lower case letters $x, y, z$ the program variables and by upper case letters $X, Y, Z$ logical variables. The program variable `nil` has a fixed meaning representing an undefined (not allocated) location. Variables and formulas are typed based on the user-defined records. Record values are denoted by mappings of record fields to locations stored in variables. For example, $\{(\texttt{lson}, y), (\texttt{rson}, z)\}$ denotes a value for a record defining a binary tree node.

The (non-)aliasing relation between variables are specified using (dis-)equality atoms, e.g., $x = y$ or $x \neq z$. A conjunction of such atoms is called a *pure formula*. The heap is specified using three kinds of atomic propositions, called also *spatial atoms*: (i) the empty heap *emp*, (ii) a heap consisting of one allocated record which location is stored in some variable, e.g., $x \mapsto \{(\texttt{lson}, y), (\texttt{rson}, z)\}$, and (iii) an unbounded heap segment allocated for a data structure whose shape is defined by an *inductive definition*. Table 1 provides some examples of such inductive definitions. The spatial atoms are connected via the *separating conjunction operator* $*$ to form a *spatial formula*. A formula in SLID is the conjunction of a pure and a spatial formula.

For example, the following formula specifies a heap built from two cells of a singly linked list (with locations stored in $x$ resp. $y$) and a list segment (starting from $z$ and ending in $y$):

$$x \mapsto \{(\texttt{nxt}, y)\} \ * \ y \mapsto \{(\texttt{nxt}, \texttt{nil})\} \ * \ \texttt{ls}(z, y) \tag{1}$$

The definition of `ls` in Table 1 specifies acyclic and possibly empty singly linked list segments starting in $E$ and with last element pointing to $F$. The list is acyclic due to the non-aliasing pure constraint $E \neq F$ which prevents $F$ to be one of the cells inside the list segment. We explain in the following the inductive definitions in Table 1. The atom $\texttt{nll}(E, F, B)$ specifies a list segment (of successor field `nxtup`) starting from $E$ where each cell has a reference (in field `inner`) to a list segment ending in some common border location $B$. The definition $\texttt{dll}(E, L, P, F)$ specifies a doubly linked list segment rooted at $E$ and ending in $L$; $P$ denotes the previous location of $E$ (stored in field `prv`) and $F$ denotes the next location of $L$ (stored in field `nxt`). The definition $\texttt{btree}(E)$ specifies a binary tree with root $E$: the first disjunct defines the case of an empty tree, i.e., with `nil` root, while the second disjunct defines a tree with at least one cell allocated at $E$ separated from the trees rooted in its locations stored in the fields `lson` and `rson`. This simple definition of binary trees is extended in `tll` to keep track of the parent node in each cell and to link the leaves of the tree into a singly linked list (between $E$ and $F$) using the `nxt` field. The general syntax of the inductive definitions is the following:

$$P(\vec{E}) \triangleq \bigvee_i \exists \vec{X}_i . \ \Pi_i \wedge \Sigma_i \tag{2}$$

*singly linked lists*

$$\texttt{ls}(E,F) \triangleq (E = F \wedge emp) \vee (E \neq F \wedge \exists X.\, E \mapsto \{(\texttt{nxt}, X)\} \ * \ \texttt{ls}(X,F)) \quad (3)$$

*nested linked lists*

$$\texttt{nll}(E,F,B) \triangleq (E = F \wedge emp) \vee (E \neq F \wedge E \neq B \wedge \exists X, Z.\, E \mapsto \{(\texttt{nxtup}, X), (\texttt{inner}, Z)\} \ * \ \texttt{ls}(Z,B) \ * \ \texttt{nll}(X,F,B)) \quad (4)$$

*doubly linked lists*

$$\texttt{dll}(E,L,P,F) \triangleq (E = F \wedge L = P \wedge emp) \vee \big(E \neq F \wedge L \neq P \wedge \exists X.\, E \mapsto \{(\texttt{nxt}, X), (\texttt{prv}, P)\} \ * \ \texttt{dll}(X,L,E,F)\ \big) \quad (5)$$

*binary tree*

$$\texttt{btree}(E) \triangleq (E = \texttt{nil} \wedge emp) \vee \big(E \neq \texttt{nil} \wedge \exists X, Y.\, E \mapsto \{(\texttt{lson}, X), (\texttt{rson}, Y)\} \ * \ \texttt{btree}(X) \ * \ \texttt{btree}(Y)\big) \quad (6)$$

*tree with linked leaves*

$$\texttt{tll}(R,P,E,F) \triangleq (R = E \wedge R \mapsto \{(\texttt{lson}, \texttt{nil}), (\texttt{rson}, \texttt{nil}), (\texttt{parent}, P), (\texttt{nxt}, F)\}) \ \vee \ \big(R \neq E \wedge \exists X, Y, Z.\ R \mapsto \{(\texttt{lson}, X), (\texttt{rson}, Y), (\texttt{parent}, P), (\texttt{nxt}, Z)\} * \texttt{tll}(X,R,E,Z) \ * \ \texttt{tll}(Y,R,Z,F)\big) \quad (7)$$

**Table 1.** Examples of inductive definitions used in the benchmark

where $\Pi_i$ (resp. $\Sigma_i$) is a pure (resp. spatial) formula over parameters $\vec{E}$ and variables in $\vec{X}_i$. Notice that the existential quantification and disjunction are allowed only in inductive definitions. Also, the semantics chosen in the competition is the *precise* semantics.

**Decidability and complexity properties:** The main difficulty that faces automatic reasoning using the symbolic heaps fragment is that the logic, due to its expressiveness, does not have very nice decidability properties [1]. For this reason, most program verification tools use incomplete heuristics to solve the satisfiability and entailment problems. We summarize below the results obtained on the decidability of this fragment. The various benchmarks and competition divisions introduced in SL-COMP stem from these results.

**SLID+:** This fragment considers inductive definitions of the general form given in equation (2). Its satisfiability problem is decidable [4], but the validity of an entailment is not [1]. A sub-fragment of SLID+ that has a decidable entailment problem has been identified in [12]. It includes *bounded tree width* inductive definitions. These definitions are general enough to define (doubly-) linked lists, trees, and structures more general than trees, such as trees whose leaves are chained in a list (i.e., `tll` in Table 1). The restrictions in [12] on the syntax allowed in equation (2) are very technical. They require in substance that all models of such formulas have bounded tree width. In particular, they require that there is exactly one points-to predicate in any $\Sigma_i$ in equation (2). The entailment problem in this fragment is EXPTIME-hard [1].

**SLNL:** This fragment, formally defined in [10], is obtained from SLID+ by mainly restricting the inductive definitions to nested lists data structures, and including skip-lists. This restriction allows to improve the complexity of the entailment problem to NPTIME.

**SLL+:** This fragment is obtained when only the `ls` definition is used. The entailment problem is $\Pi_2^P$-complete [1].

**SLL:** This is a sub-fragment of SLL+ obtained by requiring that: (i) both formulas of the entailment problem do not contain disjunctions and (ii) the right-hand side formula has no existential quantification[2]. This sub-fragment has a PTIME decision procedure for the entailment problem [8].

The competition targeted the fragments SLID+, SLNL, and SLL. Out of the 6 possible divisions (the three fragments above and the two decision problems considered), only five were opened because only one solver was specialized for checking satisfiability in SLNL.

## 3. Competition Organization

The first edition of SL-COMP focused on bringing together a large set of SL solvers and building a common but not trivial benchmark suite.

### 3.1 Launching procedure and support

A call for participation and benchmarks was sent in May 2014 to several research groups that are actively developing solvers for SL. The researchers that positively replied and some other researchers known for their work on tools for SL were invited to discuss the organization details on a public mailing list sl-comp@googlegroups.com. The support of the officers of the SMT-COMP 2014 organizing committee was solicited to benefit from their experience in the organization of such competitions. A message describing the competition was sent on the SMT-COMP and SMT-LIB discussion lists. Following this message, some members of the SMT-COMP steering committee were added to the SL-COMP discussion list. This inaugural edition of the competition was jointly organized by Mihaela Sighireanu and David Cok: Mihaela Sighireanu enlisted participation, moderated the discussion on competition rules, and organized the collection, translation, and categorization of benchmarks; David Cok advised on the competition organization and rules based on experience with other competitions, and he executed and reported the results of the competition using the StarExec framework.

### 3.2 Participants

Seven teams representing seven different solvers replied to the call for participation. These teams provided: (1) benchmarks, (2) a description of the input format of their solver, (3) a reference to a technical paper describing the principles used by the solver, (4) a web site, and (5) a statically linked Linux binary which could be executed on the StarExec platform. Because the time given for preparing the binary was very short (2 weeks), one solver [11]

2. This is the case for the verification conditions generated for checking safety properties of programs.

retired from the competition. The problem faced by this solver will be discussed further in the concluding remarks. Note that although one person is listed as "submitter" for a solver on the web site, there is generally a team of contributors behind each tool; the identified person is simply the communication contact. The participants in this first edition are described below.

**Asterix** [17]: was submitted by Juan Navarro Perez (UCL, UK) and Andrey Rybalchenko (Microsoft Research Cambridge, UK). The solver deals with the satisfiability and entailment checking in the SLL fragment. For this, it implements a model-based approach. The procedure relies on SMT solving technology (Z3 solver is used) to untangle potential aliasing between program variables. It has at its core a matching function that checks whether a concrete valuation is a model of the input formula and, if so, generalizes it to a larger class of models where the formula is also valid.

**Cyclist-SL** [6, 9]: was submitted by Nikos Gorogiannis (Middlesex University London, UK). The solver deals with the entailment checking for the SLID+ fragment. It is an instantiation of the theorem prover Cyclist for the case of Separation Logic with inductive definitions. The solver builds derivation trees and uses induction to cut infinite paths in these trees that satisfy some soundness condition. For the Separation Logic, Cyclist-SL replaces the rule of weakening used in first-order theorem provers with the frame rule of SL.

**Sleek** [7, 21]: was submitted by Quang Loc Le and Wei Ngan Chin (NUS, Singapore). The solver deals with the satisfiability and entailment checking for SLID+ formulas. It is an (incomplete but) automatic prover, that builds a proof tree for the input problem by using the classical inference rules and the frame rule of SL. It also uses a database of lemmas for the inductive definitions in order to discharge the proof obligations on the spatial formulas. The proof obligations on pure formulas are discharged by external provers like CVC4, Mona, or Z3.

**Slide** [13, 22]: was submitted by Adam Rogalewicz and Tomas Vojnar (VeriFIT, Czech Rep) and Radu Iosif (Verimag, France). The solver deals with the entailment problem in the decidable sub-fragment of SLID+ defined in [12]. The proof method for checking $\varphi \Rightarrow \psi$ relies on converting $\varphi$ and $\psi$ into two tree automata $A_\varphi$ resp. $A_\psi$, and checking the tree language inclusion of the automaton $A_\varphi$ in the automaton $A_\psi$.

**SlSat** [5]: was submitted by Nikos Gorogiannis (Middlesex University London, UK) and Juan Navarro Perez (UCL, UK). The solver deals with the satisfiability problem for the SLID+ fragment with general inductively defined predicates. The decision procedure is based on a fixed point computation of a constraint, called the "base" of an inductive predicate definition. This constraint is a conjunction of equalities and dis-equalities between a set of free variables built also by the fixed point computation from the set of inductive definitions.

**Spen** [10, 25]: was submitted by Mihaela Sighireanu (UPD & CNRS, France). The solver deals with satisfiability and entailment problems for the fragment SLNL. The decision procedures calls the MiniSAT solver on a boolean abstraction of the SL formulas to check their satisfiability and to "normalize" the formulas by inferring its implicit (dis)equalities. The core of the algorithm checking if $\varphi \Rightarrow \psi$ is valid searches a mapping from the atoms of $\psi$ to sub-formulas of $\varphi$. This search uses the membership test in tree automata to recognize in sub-formulas of $\varphi$ some unfolding of the inductive definitions used in $\psi$.

A notable fact about this set of participants is the diversity of techniques used by each solver: reduction to SAT or SMT problems (Asterix, SlSat, Spen), resolution based (Cyclist-SL, Sleek), reduction to finding graph homomorphism (Spen), and reduction to tree automata membership test (Spen) or to tree automata inclusion test (Slide). Some of the above solvers deal with other problems than the ones considered for this competition, e.g., the computation of an unsatisfiability core or abduction.

### 3.3 Benchmarks and their input format

The set of benchmarks was built based on contributions from each participant. Some of these problems come from academic software analysis or verification tools based on SL (e.g., SmallFoot [23], Hip [7]). But no problem issued from industrial tools was included in the benchmarks because we didn't receive such input. The benchmarks were collected in the input format submitted by the participants. This set of problems was translated into a common format designed like a logic of the SMT-LIB format [2]. That is, they used the syntax of SMT-LIB, although the SL theory underlying the syntax is not an official SMT-LIB theory or, at this point, even compatible with the logic underlying SMT-LIB. The rationale for this choice of common input format is that SL is combined with or translated into first-order theories that are or will be supported by the SMT-LIB format. Also, we needed a typed input format and some support for processing this format[3].

**Input format:** Each benchmark file is organized as follows:

- Preamble information required by the SMT-LIB format: the theory, the source, the kind (crafted, application, etc) and status of the problem. Here the logic was designated as **QF_S**, even though that is not at this point an official SMT-LIB logic.
- The set of sorts (records), fields, and inductive definitions used in the problem. Notice that the input format is strongly typed, i.e., each program variable has a unique type (reference to some sort) and fields are typed as function symbols from their declaration type to their definition type.
- One resp. two assertions for each satisfiability resp. entailment problem.
- The file ends with a checking satisfiability command[4].

The syntax and semantics of this format were discussed and agreed in the public mailing group. The exact definition of the theory used and the full set of benchmarks is available on-line as a GITHUB archive[5].

**Selection process:** The final version of benchmarks was the result of several iterations of the following actions on the initial set of problems:

3. In particular, we used the parser for SMT-LIB available at www.smtlib.org/utilities.html.
4. Checking the validity of the entailment $A \Rightarrow B$ is encoded by checking the satisfiability of its negation $A \wedge \neg B$.
5. github.com/mihasighi/smtcomp14-sl

- We ensured that the semantics used in the input format of solvers matched the one used in the common format. For example, some solvers rejected problems with non-aliasing constraints ($x \neq y$) between variables of different sorts. The final input format required to deal with such constraints and to interpret them as valid.

- The status (expected result), the source, and the kind of each problem were supplied as SMT-LIB meta-information. SL-COMP benchmarks are only crafted (i.e., demonstrates a particular aspect of a solver or problem domain) or random (instance of a problem template parameterized on some dimension, e.g., a size).

- The benchmark curation process identified some problems that had an incorrect expected result computed by their source solver, which required fixing the problem in the solver and in the benchmark. The status of each problem was checked before the competition using at least two solvers and it did not change during the competition.

**Divisions:** The final benchmarks have the characteristics presented in Table 2. The benchmark set was split into five divisions, using the kind of problems solved and the kind of inductive definitions used (described in Section 2), as follows:

`sll`(⊨)**:** includes satisfiability problems for the SLL fragment. Most of problems in this division were provided by the team of the Asterix solver in [16].

`sll`(⇒)**:** includes entailment problems for the SLL fragment and has the same source. The problems were either randomly generated to prove the capabilities of the solver or they are verification conditions generated by the SmallFoot tool [23].

`UDB`(⊨)**:** includes satisfiability problems for the SLID+ fragment, i.e., formulas using any fixed, user-defined set of inductive definitions satisfying the syntax given in equation (2). Recall that this problem is decidable. The problems were proposed by the teams of SLSat and Sleek solvers.

`UDB`(⇒)**:** includes entailment problems for formulas in the SLID+ fragment. Recall that this problem is undecidable. The problems were proposed by the teams of Cyclist-SL, Sleek, Slide, and Spen.

`FDB`(⇒)**:** includes entailment problems for formulas in the SLNL fragment. Recall that this fragment is decidable. The problems were proposed by the team of Spen.

### 3.4 Competition infrastructure

SL-COMP used the StarExec platform [26] under the control of David Cok and required several features provided by this platform. The pre-processing was used for 5 of the 6 solvers in order to translate the problems from the common format to the internal format of each solver. The code of these pre-processors is available on the competition GitHub repository. The competition did not use the scrambling of benchmarks because the names used for inductive definitions defined in the files of some divisions (mainly, `sll`(⊨), `sll`(⇒), and `FDB`(⇒) divisions) are important for the solvers. Each benchmark file included only one problem. The incremental feature was not used and is not supported by most of the

| Total number of problems: 678 | |
|---|---|
| Satisfiability | 25% |
| Entailment | 75% |
| **Origin** | |
| Crafted | 41% |
| Randomly generated | 59% |
| **Problems by division** | |
| `sll`($\models$) | 110 |
| `sll`($\Rightarrow$) | 292 |
| `FDB`($\Rightarrow$) | 43 |
| `UDB`($\models$) | 61 |
| `UDB`($\Rightarrow$) | 172 |

**Table 2.** Final benchmark set

competing solvers. StarExec imposes a time and space limit on each attempt of a solver to solve a given benchmark. For this competition, the CPU time[6.] was limited to 2400 seconds and the memory (RAM) limit was 100GB. Some jobs did have CPU timeout or reached the memory limit. The participants trained their solvers on the platform and provided feedback where the expected result of a problem did not match their result. Several benchmarks and solvers were fixed during this period. One training run was executed before the official run to provide insights about the global results and to do a final check of the benchmarks.

The computation of the scores obtained for each division followed the rules fixed for SMT-COMP'14. The best score is the one with, in order: (a) the least number of incorrect answers, (b) the largest number of correctly solved problems, and (c) the smallest time taken in solving the correctly solved problems. Note that solvers are allowed to respond "unknown" or to time-out on a given benchmark without penalty (other than not being able to count that benchmark as a success). Note also that a single wrong answer, even accompanied by many correct answers, places a competitor behind one that has just one right answer. This choice is intended to emphasize the need for sound, bug-free solvers that can be relied upon by an application-focused user community. Giving the number of correctly solved problems priority over the time taken to solve them is a more debatable choice. To date the emphasis in these competitions has been on raw logical capability of a solver; however, many applications are more concerned with fast solutions of most problems rather than eventual solution of more problems. Thus the scoring metric may be reconsidered for future competitions.

## 4. Results

**New benchmarks:** One of the goals of SL-COMP was to build an interesting set of benchmark problems. Aside from its use in the competition, the current set provides a reference of the kind of problems handled efficiently by the existing solvers and techniques. Moreover, researchers can use the collected benchmarks for the evaluation of competing

6. None of the participating solvers was parallel.

| *Division* | sll(⊨) | sll(⇒) | FDB(⇒) | UDB(⊨) | UDB(⇒) |
|---|---|---|---|---|---|
| *size* | 110 | 292 | 43 | 61 | 172 |
| Asterix | **110/1.06** | **292/2.98** | – | – | – |
| Cyclist-SL | – | 55/11.78 | 19/141.78 | – | **120/145.33** |
| Sleek | 110/4.99 | 292/14.13 | 31/43.65 [*1*] | **61/30.84** | 131/80.60 [*4*] |
| Slide | – | – | – | – | 36/195.61 |
| SlSat | 55/0.55 [*55*] | – | – | 37/2620.52 | – |
| Spen | 110/3.27 | 292/7.58 | **43/0.61** | – | – |

**Table 3.** Overall results $n/t$ $[f]$ where $n$ is the number of solved problems, $t$ is the cumulated CPU time (sec) on solved instances, $f$ is the number of incorrect answers, if any

solvers and for the testing of their own solver quite apart from any competition. In particular, it was important to obtain benchmarks relevant to the actual application area, i.e., software verification.

**Publicly available solvers:** StarExec requires that a public version of a solver be made available on StarExec as a condition of participating in a competition. This allows users of StarExec to rerun a competition if so desired. More importantly, it allows users to upload a new set of benchmarks of interest in their application domain and to try the various solvers against those benchmarks.

**Overall results:** Table 3 and Figure 1 provide an overall view of the competition dashboard for all divisions. This information was presented in a live web-site that was updated every minute throughout the course of the competition. This first competition only took several hours to run (as compared, for example, to SMT-COMP, which consumed a week of computation over 150 nodes of the StarExec cluster).

The results corroborate the theoretical complexity analysis provided in Section 2: the difficulty of divisions is increasing from sll(⊨) to UDB(⇒) (from left to right in Table 3) and this for any technique used. The UDB(⇒) division is the most challenging because none of the solvers competing on it could solve all the problems. However, each problem of this division can be solved by at least one solver. This fact reveals the complementarity of the techniques used (resolution techniques for Sleek and Cyclist-SL, reduction to tree automata inclusion for Slide). The results obtained on the easier divisions (sll(⊨) and sll(⇒)) may interrogate the difficulty of problems included in these divisions. However, the size of formulas in these problems is bigger than the ones included in, e.g., the UDB(⇒) division. The lack of difficulty comes from the inductive definition used, i.e., ls, which allows to obtain NPTIME decision procedures.

In general, the solvers based on resolution techniques (Sleek and Cyclist-SL) solved the bigger number of problems. For example, the Sleek solver competed in all divisions and solved 92% of benchmarks. This is due to its big database of lemmas which helps to cut proof trees. The techniques based on tree automata (Slide and Spen) were more efficient on easier divisions but solved only 65% of problems. In the FDB(⇒) division, the tree automata technique proved its completeness for this class of problems, while the deduction based techniques failed to find adequate lemmas for building the proof trees. Since the

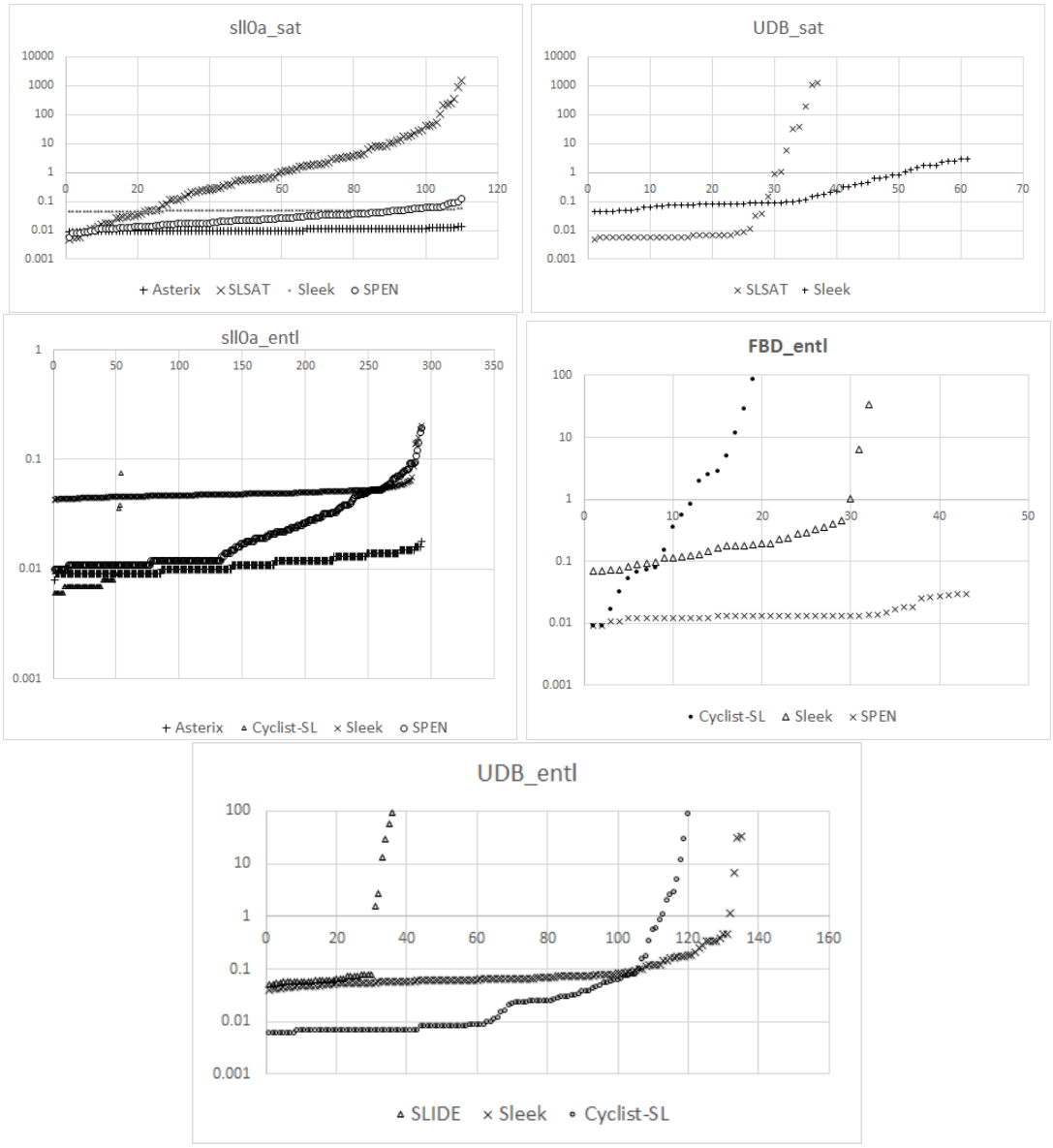


**Figure 1.** Charts for the execution times in each division (vertical axis is log of time in sec.)

techniques using tree automata are the youngest, these results encourage to improve them by a careful engineering of solvers. For the polynomial divisions `sll`($\models$) and `sll`($\Rightarrow$), the constraint programming techniques implemented in Asterix is impressive. But the results obtained with the other techniques are also good: a problem is solved in less than tens of mili-seconds. An interesting insight of the `sll`($\Rightarrow$) division is given by comparing the results of the two solvers based on the resolution techniques, i.e., Cyclist-SL and Sleek. A particular choice for the resolution technique may improve 5 times the execution time.

## 5. Conclusions and Perspectives

SL-COMP fulfilled its goals: an interesting suite of SL problems is publicly available in a common format and the maturity of solvers submitted for this competition has been proven.

**Findings:** The following additional goals were also reached:

- We achieved a first step in establishing a common format for the SL. This work revealed the features required by the existing solvers, e.g., the strong typing of formulas, the kind of inductive definitions handled, etc. Also, the use of the current format shows that it needs some readjustments to decrease its verbosity and to obtain a better integration in the semantics of the SMT-LIB format. The new features of the SMT-LIB format, e.g., abstract data types and recursive function definitions, could facilitate the revision of the common format.

- Some of the competing solvers have been improved (extensions, bugs fixed) during the preparation of this competition. They also gained experience with the preparation of a distribution for the StarExec platform. Note that the preparation of an version of a solver for StarExec was non-trivial. To avoid system-dependencies and to treat every participant in a competition equally, StarExec requires a statically-linked instance of a solver runnable in StarExec's Linux environment. Preparing such a version of a solver was time-consuming and resulted in one intended participant's withdrawal. This problem exists for first-time competitors in other StarExec-hosted competitions as well.

- A community interested in such tools has been identified and informed about the status of the existing solvers. This community could benefit from improving the tools built on the top of decision procedures for SL.

- The SMT-LIB community discovered the status of the solvers for SL and became interested in this theory.

- A interesting problem has been submitted to the StarExec platform by one solver, SeLoger [11], initially submitted to SL-COMP. This solver needs to have a batch execution in order to avoid the loading of the interpreter executing its binary (the solver needs a Windows platform) at each job. This feature is different from the incremental mode provided by the StarExec platform. It requires to measure only the score obtained on a bunch of problems, without measuring the details on each problem.

**Future work:** First of all, we are trying to reach a consensus for a good cadence of this competition. Yearly competitions could be very exciting for the first years, but may focus on engineering improvements rather than fundamental work. A good cadence may be, for example, alternating a competition year with a year of benchmark evaluation and improvement.

In the intervening time to the next edition of the competition, several tasks are intended to be accomplished.

1. The relationship of the SL theory to SMT-LIB must be understood. This may require alterations to the basic SL theory or designing an embedding of SL as an SMT-LIB theory. The discussion group sl2smtlib@googlegroups.com has been opened for working

out this relationship. The new features of the SMT-LIB format, e.g., recursive function definitions, are very important for the SL theory. The use of the SMT-LIB format for SL-COMP is convenient because some SL solvers already deal with a combination of SL with theories currently available in SMT-LIB e.g., integer arithmetics theory [16], array theory [3], set theory [18], etc.

2. With the experience of the current competition, the set of benchmarks has to be improved in the following directions:

   - More benchmarks must be added in the existing divisions, especially the ones with smallest number of benchmarks. Moreover, we have to increase the number of benchmarks coming from the software verification tools developed in academia or industry. The easy divisions, i.e., `sll`($\models$) to `FDB`($\Rightarrow$), should include more difficult problems to push at their limits the existing solvers.
   - Some new divisions will be introduced to take into account the needs of the software verification tools. For example, the combination of SL with integer arithmetics is used in tools that reason about the heap consumption.
   - The scoring system will be improved by assigning a difficulty to each benchmark problem. This should lead to a better evaluation of each solver. A classic way to assign a difficulty level is to take into account the size of the formulas and of the inductive definitions used in the problem. However, we think that for divisions like `UDB`($\models$) and `UDB`($\Rightarrow$), an additional factor of difficulty is the number of fields of the data structure and the "uniformity" of its shape. For example, it seems to be easier to solve problems using `btree` definition than the `tll` definition.

3. The use of solvers in software verification tools requires improving solver capabilities such as computing a witness for satisfiability problems, providing unsatisfiability cores (i.e., diagnosis) or proofs, and supporting abduction.

4. An interesting task is to find a way to measure individual solver improvement (in the face of a changing benchmark set), because this should encourage teams to work on both the engineering and the theoretical foundations of their tools.

5. Finally, we should ease the task of entrants in this competition by providing, well in advance, clear rules, guidelines for solver preparation, pre-processors for their input language, etc. Many of these aspects were worked out in this first competition as participants were making their preparations.

## Acknowledgments

The authors thank the contributors to this competition: Nikos Gorogiannis, Juan Navarro Perez, Adam Rogalewicz, Ondrej Lengal, Tomas Vojnar, Wei Ngan Chin, Le Quang Loc, Radu Iosif, Andrey Rybalchenko, and Constantin Enea.

We thank the participants to the discussion list, especially Josh Berdine, John Brotherston, Christoph Hasse, and Thomas Wies, for their interesting comments on the input theory and format used in the competition. We also thank reviewers of the paper for their interesting suggestions for improving its presentation.

## 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*, **8412** of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2014.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[3] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, **7148** of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2012.

[4] 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.

[5] James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *CSL-LICS*, page 25. ACM, 2014.

[6] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In *APLAS*, **7705** of *LNCS*, pages 350–367. Springer, 2012.

[7] 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.

[8] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*, **6901** of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2011.

[9] CYCLIST. https://github.com/ngorogiannis/cyclist.

[10] Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Compositional entailment checking for a fragment of separation logic. In *APLAS*, Lecture Notes in Computer Science. Springer, 2014. To appear.

[11] Christoph Haase, Samin Ishtiaq, Joël Ouaknine, and Matthew J. Parkinson. SeLoger: A tool for graph-based reasoning in separation logic. In *CAV*, **8044**, pages 790–795, 2013.

[12] Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In *CADE*, **7898** of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2013.

[13] Radu Iosif, Adam Rogalewicz, and Tomás Vojnar. Deciding entailments in inductive separation logic with tree automata. In *ATVA*, LNCS. Springer, 2014.

[14] Peter O'Hearn. Separation logic. www0.cs.ucl.ac.uk/staff/p.ohearn/SeparationLogic/Separation_Logic/SL_Home.html.

[15] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, **2142** of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[16] Juan A. Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011.

[17] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *APLAS*, **8301** of *Lecture Notes in Computer Science*, pages 90–106. Springer, 2013.

[18] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *CAV*, **8044** of *Lecture Notes in Computer Science*, pages 773–789. Springer, 2013.

[19] 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.

[20] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[21] SLEEK. http://loris-7.ddns.comp.nus.edu.sg/~project/s2/beta/.

[22] SLIDE. http://www.fit.vutbr.cz/research/groups/verifit/tools/slide/.

[23] SmallFoot. www0.cs.ucl.ac.uk/staff/p.ohearn/smallfoot/.

[24] SMT-COMP. smtcomp.sourceforge.org.

[25] SPEN. https://www.github.com/mihasighi/spen.

[26] StarExec. www.starexec.org.