metaSMT: A Unified Interface to SMT-LIB2

Heinz Riener[1] Mathias Soeken[1,2] Clemens Werther[1] Görschwin Fey[1,3] Rolf Drechsler[1,2]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[3]Institute of Space Systems, German Aerospace Center, 28359 Bremen, Germany
{hriener,msoeken,clemens,fey,drechsle}@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/agra/eng/metasmt.php

***Abstract*—Various problems from artificial intelligence and formal methods are solved utilizing *Satisfiability Modulo Theories* (SMT) solvers. Selecting the best SMT solver for a specific application, however, is a daunting task. In this paper, we present the novel metaSMT TCP server and client architecture which can be used to solve SMT instances expressed in SMT-LIB2 by multiple solver processes in parallel. The metaSMT TCP server provides a unified interface for SMT-LIB2 instances with the capability to either use the API or the file interface of a solver process and thus serves as a highly customizable portfolio solver. We show that the run-time overhead required by the metaSMT TCP server and client architecture is marginal using selected benchmarks from SMT-LIB.**

I. Introduction

Today, various problems from artificial intelligence and formal methods for hardware and software are solved by reducing them to one or more instances of the *Satisfiability* (SAT) problem. The SAT problem is to decide whether a given Boolean formula in *Conjunctive Normal Form* (CNF) is satisfiable. Although SAT is NP-complete [1], [2], effective reasoning engines are available. For most applications, however, formulations in richer fragments of first-order logic with background theories that fix the interpretation of certain predicates and function symbols are more convenient. *Satisfiability Modulo Theories* (SMT) [3] is the field that focuses on deciding satisfiability of standardized first-order logic fragments with respect to some specific background theories. For instance, the first-order logic fragment QF_BV corresponds to closed quantifier-free formulas over the theory of fixed-size bit-vectors. This fragment fixes the interpretation of common bit-vector function symbols such as subtraction of two bit-vectors of length n to two's complement subtraction modulo 2^n.

A large number of different decision procedures for SMT, called *SMT solvers*, has been proposed. Selecting the best SMT solver for a specific application, however, is a daunting task due to two reasons: (1) since deciding satisfiability is NP-complete, efficient SMT solvers strongly rely on heuristic approaches. Thus, each SMT solver has its own strengths and weaknesses and consequently reasoning time and memory consumption varies depending on whether the solver's heuristic is effective for a particular problem instance; (2) different SMT solvers do not use a common input language or API. A standardized textual format, called SMT-LIB [4] and SMT-LIB2 (referring to its second version), for expressing SMT instances has been established. However, not all SMT solvers currently follow this standard.

metaSMT [5] is a flexible framework for integrating multiple solvers into C++ and Python applications using a common interface provided by an *Embedded Domain Specification Language* (EDSL) similar to SMT-LIB2. Thus, metaSMT allows to switch between different SMT solvers by using one common API. metaSMT's EDSL has been effectively used for test stimuli generation of software [6] and hardware [7], fault localization [8], assessing fault tolerance [9], symbolic execution [10], and circuit minimization [11].

In this paper, we introduce two new components of metaSMT which can be used to provide a unified SMT-LIB2 interface to multiple solvers: (1) the SMT-LIB2 parser with the generic evaluator and (2) the metaSMT *Transmission Control Protocol* (TCP) server. These two components together form the novel metaSMT TCP server and client architecture and extend metaSMT by a mechanism to decide an SMT instance using multiple SMT solvers in parallel. The main contribution of the paper is as follows:

1) We present an SMT-LIB2 parser and a generic evaluator which can be used to turn any metaSMT backend (including SAT solvers) into an SMT solver with an SMT-LIB2 compatible input parser.
2) We introduce the novel metaSMT TCP server and client architecture which can be used to decide an SMT instance by multiple solver processes in parallel allowing to easily build customized portfolio solvers.
3) We present experimental results for a selected subset of the SMT library benchmarks[1]. More specifically, we focus on the QF_BV logic benchmarks "bruttomesso/lfsr" of Roberto Bruttomesso [12].

The remainder of the paper is structured as follows. In Section II, we briefly introduce the SMT-LIB2 common language and review the metaSMT layer architecture. In Section III, we describe the two novel components of metaSMT which together form the metaSMT TCP server and client architecture. In Section IV, we present experimental results. Section V concludes the paper.

[1]The Satisfiability Modulo Theories Library, http://www.smt-lib.org/

```
(Commands) c   ::=  set-logic L
                 |  set-option o
                 |  set-info α
                 |  declare-fun f (x : σ)* σ t
                 |  push n
                 |  pop n
                 |  assert t
                 |  check-sat
                 |  get-value t+
                 |  exit
(Scripts) scr  ::=  c*
```

Fig. 1. Abstract syntax for commands

II. Overview

A. SMT-LIB2 Command Language

The SMT-LIB2 standard [4] defines a command language to describe the syntax of commands to and responses from interactive SMT solvers. A tool which uses an SMT-LIB2 compatible SMT solver issues commands in the textual format of the command language. The SMT solver reads the commands from an input channel, processes them, and writes responses in the textual format of the command language to two output channels, a regular output channel and a diagnostic output channel. The input channel usually refers to standard input or a file and the output channels usually refer to standard output and standard error. However, any other input and output channels can be used, too.

In this section, we briefly outline the syntax of SMT-LIB2 commands and the corresponding solver responses. Our description uses the terminology of SMT-LIB2. For a detailed treatment of the syntax and semantics of SMT-LIB2, we refer the reader to the SMT-LIB2 standard [4].

metaSMT in its current implementation is compatible to the SMT-LIB2 standard but not fully SMT-LIB2 compliant, i.e., not all SMT-LIB2 commands of the command language are currently supported. A simplified version of the abstract syntax of the SMT-LIB2 command language that is actually supported by metaSMT is shown in Fig. 1.

An SMT-LIB2 compatible solver responds to every issued command. In general, three possible responses are possible. The SMT solver responds with `success` if the command was successfully processed, `unsupported` if the command is not supported by the SMT solver, and with `error <msg>` if an error occurs during processing the command, where `<msg>` refers to an arbitrary error message in textual format. Moreover, in case of a `check-sat` command the SMT solver either responds with `sat`, `unsat`, or `unknown` corresponding to the satisfiability check where the latter indicates that a given resource limit (memory or time) has been reached.

In the following, we use the term SMT-LIB2 to refer to the subset of the command language that is supported by metaSMT. Thus, an SMT instance is a script consisting of sequences of SMT-LIB2 commands.

B. metaSMT Layer Architecture

The "original" metaSMT architecture [5] consists of three layers: a frontend, a middleend, and a backend layer. The metaSMT architecture is illustrated in Fig. 2.

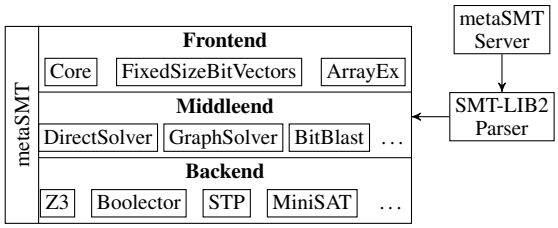

Fig. 2. metaSMT Architecture

The frontend layer provides primitives of the SMT-LIB2 format in form of an EDSL, i.e., an API for a programming language. Currently, the programming languages C++ and Python are supported and primitives for basic Boolean operators (Core theory), bit-vectors with arbitrary size (FixedSizeBitVectors theory), and functional arrays with extensionality (ArrayEx theory) are provided. The frontend layer, however, does not attach any semantics to the primitives.

The middleend layer provides different intermediate representations and offers a set of possible translations and optimizations. The two commonly used middleends are the `DirectSolver` and `GraphSolver`. The `DirectSolver` resolves the meaning of primitives by directly forwarding them to the backend layer with no intermediate representation. Contrarily, the `GraphSolver` constructs a *Directed Acyclic Graph* (DAG) from the primitives as intermediate representation. The DAG can then be manipulated by possible optimizations. Lastly, the DAG is traversed and mapped to the backend layer. The translations and optimizations are combinable by cascading different middleends. For instance, a `BitBlast` middleend is available that transforms QF_BV primitives to Boolean primitives allowing to check an SMT instance with a SAT solver after the middleend processes the SMT instance.

The backend layer provides two interfaces to different SAT and SMT solvers. The first interface is a static mapping of the metaSMT API to the respective API of the SAT or SMT solver. The second interface is a generic *Input-Output* (I/O) streaming interface that opens an input and an output stream to any SMT-LIB2 compatible SMT solver. metaSMT then writes SMT-LIB2 commands to the generic I/O streaming interface. The SMT solver reads the commands from the input stream and writes its responses to the output stream. Both interfaces supported by metaSMT, the statically mapped API interface and the generic I/O streaming interface, allow for checking SMT instances incrementally.

In metaSMT's terminology, the combination of at least one middleend and a backend is called a *context*.

III. Architecture

In this paper, we describe two new components of metaSMT: (1) the SMT-LIB2 parser and generic evaluator and (2) the novel metaSMT TCP server. In terms of the three layer architecture of metaSMT, the SMT-LIB2 parser and generic evaluator serve as a file interface for metaSMT that reads SMT-LIB2 instances from a file and passes them to the middleend layer. As a consequence, metaSMT allows for turning any

configuration of middleend and backend into an SMT solver that reads SMT-LIB2 instances via file interface. This is especially interesting for those SMT solver APIs supported by metaSMT that do not offer an SMT-LIB2 compatible input parser.

The metaSMT TCP server integrates the generic SMT-LIB2 parser and evaluator to provide an SMT-LIB2 compatible interface to multiple solvers. For each SMT-LIB2 instance sent to the server, the server forks multiple SMT solver processes and passes the SMT-LIB2 instance command by command to all processes. A solver process is a customization of a middleend and a backend. As soon as the fastest SMT solver process responds to the metaSMT TCP server with a result, the server passes the result back to the client and kills all remaining SMT solver processes.

In this section, we describe the generic SMT-LIB2 parser and evaluator (Section III-A) and the metaSMT TCP server and client architecture (Section III-B) in detail.

A. SMT-LIB2 Parser and Generic Evaluator

The SMT-LIB2 parser reads an SMT-LIB2 instance command by command from a file. From each SMT-LIB2 command, an *Abstract Syntax Tree* (AST) is generated and passed to the generic evaluator which is parametrized with a context. The generic evaluator recursively traverses the ASTs and instantiates the metaSMT API for the respective context. The context then processes the data as in the "original" metaSMT architecture, i.e., first the intermediate representation of the first middleend is generated, then passed from one middleend to the next middleend, until the data is finally processed by the backend. All responses from the backend layer are passed back to the evaluator.

B. TCP Server and Client Architecture

1) Architecture: The metaSMT TCP server and client architecture is sketched in Fig. 3. A client reads the SMT-LIB2 instance, establishes a TCP connection to the metaSMT server, and transmits the instance to the server via this connection. The metaSMT TCP server then generates a `Connection` object, buffers the incoming data from the TCP connection, and passes every recognized SMT-LIB2 command from the buffer to the `Connection` object. As soon as results are ready, the metaSMT TCP server reads the results from the `Connection` object and sends them back to the client via the TCP connection.

The internal architecture of a `Connection` object is shown in Fig. 4. Each `Connection` object instantiates an SMT-LIB2 parser and generic evaluator and multiple, different solver processes. Each solver process is a metaSMT context in its own process. Instead from a file, the SMT-LIB2 parser reads the SMT-LIB2 commands recognized by the server and generates for each command an AST. The AST is then passed to the generic evaluator which recursively traverses the AST and calls metaSMT's frontend API for each individual solver process. In case a of `check-sat` command the server blocks and waits until the first solver process responds with a result. This result is passed back to the client and all other solvers are terminated.

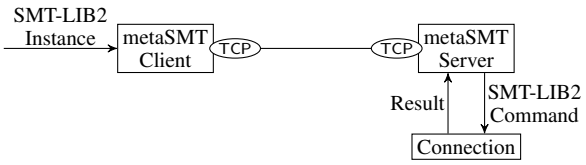

Fig. 3. metaSMT TCP Server and Client Architecture

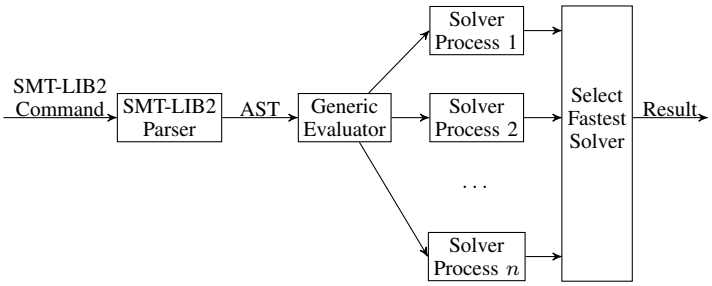

Fig. 4. Internal Architecture of a `Connection` object

2) Protocol: The communication between the server and a client follows a simple communication protocol. An example communication is shown in Fig. 5. The communication consists of two protocol phases. In the first phase the client selects a set of solvers and specifies a maximum time limit (timeout) for deciding satisfiability. In the second phase the client sends the SMT instance to the server. The server parses the SMT instance, forwards it command by command to all solver processes, and returns their answers back to the client.

In the following we describe the two protocol phases in detail: when a client establishes a connection to the server, the server creates a new `Connection` object. The client then sends a sequence of solver names to the server which forks an individual solver for each name process and registers the solver process in the `Connection` object. If the solver process is successfully created and registered, the server acknowledges

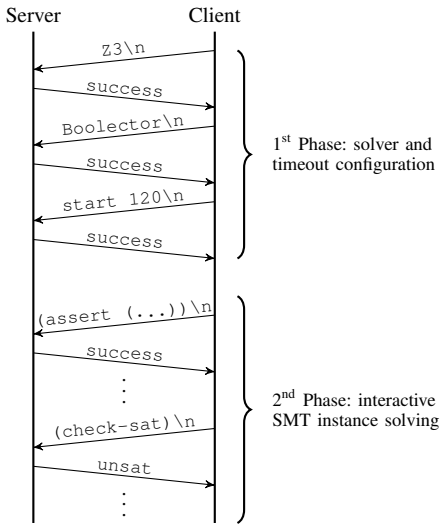

Fig. 5. Example communication between server and client

the solver name with a response `success`. The first phase is ended when the client sends the command `start` t, where t specifies a maximum time limit in seconds for deciding satisfiability, and the server responds with `success`. In the example communication in Fig. 5, the solvers Z3 and Boolector are selected and a timeout of 120 seconds is specified.

In the second phase, the client sends the SMT instance line by line to the server. Notice that the client does not need to separate the SMT instance into individual commands but the server collects all lines in a buffer until an SMT-LIB2 command is recognized. The SMT-LIB2 command is then passed to all solver processes registered in the `Connection` object. The solver processes parse and evaluate the SMT-LIB2 command (see Sec. III-A) simultaneously and return their responses to the server which forwards them to the client. In case of the SMT-LIB2 command `check-sat` the server waits for the fastest solver process, sends its response to the client, and terminates all other solver processes. This is a simple decision heuristic assuming that the fastest solver is the best solver. If none of the solver processes returns a result to the `check-sat` command before the timeout is reached, the server terminates all solver processes and returns `unknown` to the client.

IV. Experimental Evaluation

In order to measure the overhead of the metaSMT TCP server and client architecture, we conducted two experiments. In the first experiment, we use the Z3 4.1 executable which provides a SMT-LIB2 compatible file interface and the metaSMT TCP server configured with I/O streaming interface that passes SMT-LIB2 commands as well to the Z3 4.1 executable. Thus, the comparison shows the time overhead required for solving the SMT instance leveraging the metaSMT TCP server and client architecture. In the second experiment, we use the metaSMT server and client architecture as a customized portfolio solver. We selected the three metaSMT API backends Boolector, Z3, and STP as solver processes and compare the run-times of the portfolio solver to the minimum run-times of the three individual metaSMT API backends, i.e., for each SMT instance the fastest of the three metaSMT API backends is selected for comparison. For all experiments, we use the `DirectSolver` at metaSMT's middleend, i.e., no intermediate representation is constructed and no optimization is enabled by metaSMT.

All experiments have been conducted on an AMD Phenom™ II X4 965 processor with four cores and 8GB RAM. We use the following SMT solver and API versions: Boolector v1.5.118, Z3 4.1, and STP 32:1668.

The SMT instances used in the experiments are a selected subset of the SMT library benchmarks. Particularly, we use the 240 SMT instances "bruttomesso/lfsr" which formalize the behavior of *Linear Feedback Shift Registers* [12]. Table I lists details of experimental results for the benchmarks `lfsr_t_bw_n` with $t > 2$ and $n > 32$, i.e., a subset of the 240 SMT instances[2]. The timeout for a response to `check-sat` was set to 120 seconds. In the following, we use scatter plots to present our experimental results for all 240 SMT instances.

[2]Detailed results for all SMT instances are available on http://www.informatik.uni-bremen.de/agra/projects/smtlib.html.

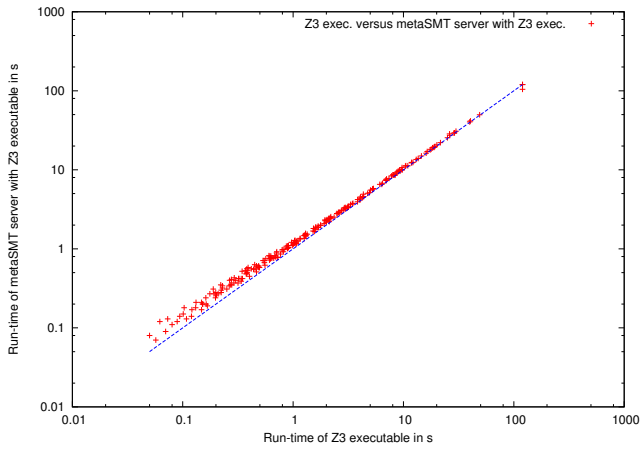

Fig. 6. Run-times between Z3 executable and metaSMT server and client architecture with Z3 executable as backend

A. Z3 Executable versus metaSMT Server

In the first experiment, we compare the run-times of the Z3 executable and the metaSMT TCP server and client architecture with Z3 executable as backend, i.e., metaSMT uses an I/O streaming interface as backend that sends SMT-LIB2 commands to the Z3 executable. Fig. 6 shows a scatter plot of the run-times for all SMT instances in logarithmic scale for both axes. The horizontal axis indicates the run-time required by the Z3 executable and the vertical axis indicates the run-time required by the metaSMT server and client architecture for deciding an SMT instance.

As can be seen, the overhead of using metaSMT's server and client architecture is negligible, in particular when the run-time increases. This gives the impression that only a small constant overhead is required by metaSMT which becomes insignificant as the instances become more complex.

B. Portfolio Solver versus Fastest SMT Solver

In the second experiment, we compare the run-times of a customized portfolio solver leveraging the metaSMT server and client architecture with the API backends Boolector, STP, and Z3 in parallel to the fastest of the three individual metaSMT's backends. Fig. 7 shows a scatter plot of the run-times for all SMT instances in logarithmic scale for both axes. The horizontal axis indicates the minimum of the run-times of the three metaSMT backends for deciding the SMT instance. The vertical axis indicates the run-time of the customized portfolio solver for deciding an SMT instance when all three backends are used as solver processes.

The overhead for the portfolio solver is marginal, particularly for instances that require a long run-time. Hence, when using the portfolio solver one requires almost the same run-time as the fastest solver without knowing the fastest solver in

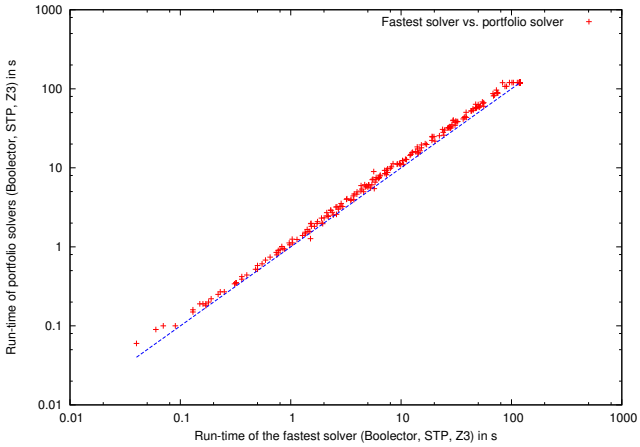

Fig. 7. Run-times between a customized portfolio solver with API backends Boolector, STP, and Z3 and the fastest of the three metaSMT backends

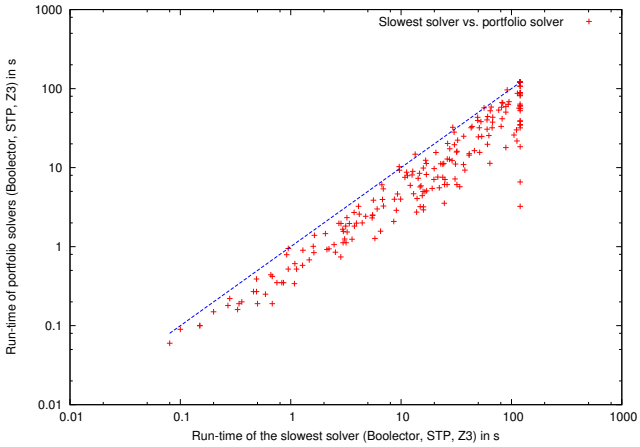

Fig. 8. Run-times between a customized portfolio solver with API backends Boolector, STP, and Z3 and the slowest of the three metaSMT backends

advance. This effect is emphasized by the scatter plot shown in Fig. 8 which compares the portfolio solver with the slowest solver. The run-times required by different solvers can differ significantly.

C. Threats to Validity

metaSMT attempts to provide an SMT-LIB2 compliant API. Thus, a metaSMT API backend uses a fixed API mapping which is not necessarily optimal with respect to the developer's intention. Thus, long run-times do not necessarily indicate bad performance of an SMT solver but unoptimized API usage.

We use a single benchmark set ("bruttomesso/lfsr") for all experiments. The benchmark category for all SMT instances is "crafted". According to the SMT-COMP 2013 rules and procedures [13], these benchmark neither stem from an industrial use case nor were randomly created but were particularly designed to stress the SMT solver.

A broader evaluation with a larger benchmark set and especially incremental SMT instances is planed for future work.

V. Conclusion

In this paper, two novel components of metaSMT have been presented: (1) the SMT-LIB2 parser with the generic evaluator and (2) the metaSMT TCP server. Together these two components enable the novel metaSMT TCP server and client architecture which allows for deciding SMT instances simultaneously with multiple solvers. Thus, metaSMT can be used to easily create customized portfolio solvers. Experiments for a selected set of SMT instances from SMT-LIB have been presented showing that the run-time overhead consumed by the architecture is marginal.

metaSMT is available as open source software. The reader is referred to http://www.informatik.uni-bremen.de/agra/eng/metasmt.php for further information.

Acknowledgment

This work was supported by the *German Research Foundation* (DFG) within the Emmy Noether programme (DFG, grant no. FE 797/6-1) and the Reinhart Koselleck project (DFG, grant no. DR 287/23-1). We would like to thank all contributors and users of metaSMT.

References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[2] L. A. Levin, "Universal search problems," in *Problemy Peredaci Informacii 9*, 1973, pp. 115–116, translated in problems of Information Transmission 9, 265–266.

[3] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. IOS Press, 2009, pp. 825–885.

[4] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," 2012.

[5] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler, "metaSMT: Focus on your application not on solver integration," in *International Workshop on Design and Implementation of Formal Tools and Systems*, 2011, pp. 22–29.

[6] H. Riener, R. Bloem, and G. Fey, "Test case generation from mutants using model checking techniques," in *International IEEE Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 388–397.

[7] H. M. Le and R. Drechsler, "CRAVE 2.0: The next generation constrained random stimuli generator for SystemC," in *DVCon Europe*, 2014.

[8] H. Riener and G. Fey, "Model-based diagnosis versus error explanation," in *International Conference on Formal Methods and Models for Codesign*, 2012, pp. 43–52.

[9] H. Riener, S. Frehse, and G. Fey, "Improving fault tolerance utilizing hardware-software-co-synthesis," in *Design, Automation and Test in Europe*, 2013, pp. 939–942.

[10] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *International Conference on Computer Aided Verification*, 2013, pp. 53–68.

[11] N. Abdessaied, M. Soeken, R. Wille, and R. Drechsler, "Exact template matching using boolean satisfiability," in *IEEE International Symposium on Multiple-Valued Logic*, 2013, pp. 328–333.

[12] R. Bruttomesso and N. Sharygina, "A scalable decision procedure for fixed-width bit-vectors," in *International Conference on Computer-Aided Design*, 2009, pp. 13–20.

[13] R. Bruttomesso, D. R. Cok, and A. Griggio, "Satisfiability modulo theories competition (SMT-LIB) 2013: Rules and procedures," 2012, this version revised 2012-6-2.

TABLE I

RESULTS FOR A SELECTED SUBSET OF THE BRUTTOMESSO/LFSR BENCHMARKS `lfsr_t_bw_n`

Benchmark	Boolector		STP		Z3 API		Portfolio		SMT2		Z3 (Exec.)	
	decision	time	decision	time	decision	time	decision	solver	time	decision	time	decision	time
lfsr_004_111_096	unsat	27.25	unsat	28.41	unsat	51.12	unsat	Boolector	31.97	unsat	2.29	unsat	2.010
lfsr_008_015_112		timeout		timeout		timeout			timeout		timeout		timeout
lfsr_004_127_096	unsat	36.96	unsat	26.46	unsat	66.80	unsat	STP	31.95	unsat	2.43	unsat	2.122
lfsr_008_063_112		timeout		timeout		timeout		timeout		unsat	8.62	unsat	8.007
lfsr_008_127_064		timeout	unsat	44.79	unsat	97.53	unsat	STP	55.71	unsat	9.07	unsat	8.611
lfsr_004_095_064	unsat	8.17	unsat	9.60	unsat	9.62	unsat	Boolector	10.29	unsat	1.25	unsat	1.071
lfsr_004_111_128	unsat	56.43	unsat	56.05	unsat	82.21	unsat	Boolector	66.69	unsat	3.24	unsat	2.978
lfsr_008_159_064		timeout	unsat	87.36		timeout	unsat	STP	107.25	unsat	11.25	unsat	10.954
lfsr_008_159_096		timeout		timeout		timeout			timeout	unsat	17.89	unsat	17.464
lfsr_004_143_112		timeout		timeout		timeout			timeout	unsat	8.62	unsat	8.158
lfsr_008_031_064		timeout	unsat	104.02	unsat	30.79	unsat	Z3	34.41	unsat	3.72	unsat	3.442
lfsr_004_015_112		timeout		timeout		timeout			timeout	unsat	27.36	unsat	26.231
lfsr_008_159_128		timeout		timeout		timeout			timeout	unsat	22.07	unsat	21.640
lfsr_004_015_064		timeout		timeout	unsat	56.16	unsat	Z3	59.90	unsat	3.37	unsat	3.170
lfsr_008_047_080		timeout	unsat	82.28	unsat	43.53	unsat	Z3	52.58	unsat	4.91	unsat	4.330
lfsr_008_143_112		timeout		timeout		timeout			timeout	unsat	41.52	unsat	40.436
lfsr_004_095_096	unsat	28.26	unsat	35.58	unsat	43.94	unsat	Boolector	33.17	unsat	1.87	unsat	1.609
lfsr_004_047_128	unsat	79.16	unsat	116.99	unsat	47.24	unsat	Z3	56.62	unsat	2.90	unsat	2.575
lfsr_004_159_128	unsat	70.73	unsat	68.81		timeout	unsat	STP	81.31	unsat	4.53	unsat	4.084
lfsr_008_127_112		timeout		timeout		timeout			timeout	unsat	18.21	unsat	17.807
lfsr_008_047_064	unsat	80.61	unsat	44.46	unsat	27.61	unsat	Z3	33.23	unsat	3.48	unsat	3.162
lfsr_004_047_112	unsat	49.93	unsat	67.40	unsat	30.91	unsat	Z3	37.68	unsat	2.32	unsat	1.942
lfsr_008_079_064	unsat	67.64	unsat	46.80	unsat	39.07	unsat	STP	43.44	unsat	4.40	unsat	4.048
lfsr_004_095_080	unsat	17.52	unsat	16.54	unsat	26.42	unsat	STP	20.18	unsat	1.47	unsat	1.255
lfsr_004_111_080	unsat	24.08	unsat	30.26	unsat	26.07	unsat	Boolector	28.10	unsat	1.86	unsat	1.695
lfsr_004_063_080	unsat	32.53	unsat	23.33	unsat	19.00	unsat	Z3	22.24	unsat	1.21	unsat	0.968
lfsr_008_079_096		timeout	unsat	105.44	unsat	95.95			timeout	unsat	8.61	unsat	8.089
lfsr_004_095_112	unsat	37.30	unsat	49.37	unsat	46.88	unsat	Boolector	43.34	unsat	2.27	unsat	1.950
lfsr_004_079_128	unsat	48.19	unsat	48.92	unsat	56.44	unsat	Boolector	57.02	unsat	2.43	unsat	2.111
lfsr_004_143_064	unsat	46.29	unsat	37.96	unsat	60.89	unsat	STP	44.38	unsat	3.40	unsat	3.159
lfsr_008_063_080		timeout	unsat	52.08	unsat	56.62	unsat	STP	60.86	unsat	6.55	unsat	6.100
lfsr_004_047_064	unsat	7.80	unsat	14.70	unsat	7.14	unsat	Z3	8.46	unsat	0.75	unsat	0.566
lfsr_004_031_096	unsat	36.67	unsat	82.31	unsat	36.54	unsat	Z3	41.55	unsat	1.81	unsat	1.537
lfsr_004_063_064	unsat	7.59	unsat	7.54	unsat	11.23	unsat	STP	8.73	unsat	0.89	unsat	0.707
lfsr_008_063_096		timeout	unsat	105.56	unsat	83.01	unsat	Z3	119.57	unsat	7.80	unsat	7.392
lfsr_008_079_112		timeout		timeout		timeout			timeout	unsat	9.84	unsat	9.255
lfsr_008_143_064		timeout		timeout		timeout			timeout	unsat	25.46	unsat	25.058
lfsr_008_015_064		timeout		timeout		timeout			timeout	unsat	12.23	unsat	11.890
lfsr_008_015_128		timeout		timeout		timeout			timeout		timeout		timeout
lfsr_008_159_112		timeout		timeout		timeout			timeout	unsat	19.55	unsat	19.068
lfsr_008_047_096		timeout		timeout	unsat	74.22	unsat	Z3	89.27	unsat	5.69	unsat	5.247
lfsr_004_015_096		timeout		timeout		timeout			timeout	unsat	19.16	unsat	18.881
lfsr_008_095_096		timeout	unsat	101.05		timeout			timeout	unsat	8.35	unsat	7.802
lfsr_004_111_064	unsat	12.25	unsat	13.41	unsat	13.42	unsat	Boolector	14.78	unsat	1.47	unsat	1.271
lfsr_004_159_064	unsat	23.64	unsat	14.10	unsat	27.63	unsat	STP	17.34	unsat	2.09	unsat	1.915
lfsr_004_127_080	unsat	31.64	unsat	28.20	unsat	43.08	unsat	STP	31.97	unsat	1.98	unsat	1.775
lfsr_008_063_128		timeout		timeout		timeout			timeout	unsat	9.63	unsat	8.956
lfsr_004_079_112	unsat	32.00	unsat	37.34	unsat	52.90	unsat	Boolector	38.15	unsat	2.00	unsat	1.776
lfsr_004_079_096	unsat	26.90	unsat	28.30	unsat	29.39	unsat	Boolector	32.40	unsat	1.50	unsat	1.282
lfsr_008_127_080		timeout	unsat	75.35	unsat	117.64	unsat	STP	88.05	unsat	13.94	unsat	13.597
lfsr_008_143_128		timeout		timeout		timeout			timeout	unsat	49.77	unsat	49.019
lfsr_004_143_128		timeout		timeout		timeout			timeout	unsat	9.85	unsat	9.196
lfsr_004_015_080		timeout		timeout		timeout			timeout	unsat	9.94	unsat	9.713
lfsr_008_063_064	unsat	111.07	unsat	24.78	unsat	40.25	unsat	STP	30.03	unsat	3.22	unsat	2.973
lfsr_008_031_080		timeout		timeout		timeout			timeout	unsat	5.80	unsat	5.331
lfsr_004_111_112	unsat	51.18	unsat	52.69	unsat	65.53	unsat	Boolector	58.38	unsat	2.48	unsat	2.176
lfsr_004_031_112	unsat	90.37		timeout		timeout	unsat	Boolector	107.55	unsat	3.65	unsat	3.316
lfsr_008_095_112		timeout		timeout			timeout		timeout	unsat	12.41	unsat	11.827
lfsr_008_015_080		timeout		timeout			timeout		timeout	unsat	40.34	unsat	39.883
lfsr_008_111_128		timeout		timeout			timeout		timeout	unsat	17.07	unsat	16.404
lfsr_004_063_128	unsat	119.66	unsat	47.37	unsat	74.35	unsat	STP	63.70	unsat	3.31	unsat	2.909
lfsr_008_015_096		timeout		timeout			timeout		timeout	unsat	104.48		timeout
lfsr_008_047_112		timeout		timeout	unsat	119.04			timeout	unsat	7.67	unsat	7.024
lfsr_004_063_112	unsat	83.30	unsat	55.24	unsat	62.26	unsat	STP	65.52	unsat	2.35	unsat	2.032
lfsr_008_031_128		timeout		timeout			timeout		timeout	unsat	28.50	unsat	26.159
lfsr_008_143_080		timeout		timeout			timeout		timeout	unsat	29.38	unsat	29.027
lfsr_004_079_064	unsat	8.23	unsat	9.66	unsat	7.10	unsat	Z3	9.28	unsat	1.10	unsat	0.908
lfsr_008_095_080	unsat	113.10	unsat	68.06	unsat	94.82	unsat	STP	87.10	unsat	6.76	unsat	6.458
lfsr_004_127_064	unsat	19.59	unsat	7.54	unsat	19.91	unsat	STP	9.72	unsat	1.53	unsat	1.340
lfsr_004_079_080	unsat	10.29	unsat	15.29	unsat	16.68	unsat	Boolector	12.37	unsat	1.28	unsat	1.043
lfsr_004_063_096	unsat	41.29	unsat	12.06	unsat	29.13	unsat	STP	14.38	unsat	1.81	unsat	1.529
lfsr_004_143_096		timeout		timeout			timeout		timeout	unsat	8.53	unsat	8.365
lfsr_008_031_096		timeout		timeout			timeout		timeout	unsat	9.49	unsat	8.970
lfsr_004_047_096	unsat	37.35	unsat	62.44	unsat	22.23	unsat	Z3	25.48	unsat	1.57	unsat	1.296
lfsr_004_031_064	unsat	10.47	unsat	36.76	unsat	9.93	unsat	Z3	10.93	unsat	0.81	unsat	0.629
lfsr_004_047_080	unsat	12.24	unsat	26.19	unsat	10.95	unsat	Z3	12.85	unsat	0.97	unsat	0.794
lfsr_008_143_096		timeout		timeout			timeout		timeout	unsat	30.80	unsat	29.900
lfsr_008_079_080	unsat	91.55	unsat	77.05	unsat	73.10	unsat	Z3	96.32	unsat	5.75	unsat	5.284
lfsr_004_143_080	unsat	83.12	unsat	69.91		timeout	unsat	STP	83.09	unsat	7.33	unsat	7.004
lfsr_008_127_128		timeout		timeout			timeout		timeout	unsat	20.67	unsat	20.060
lfsr_008_159_080		timeout		timeout		timeout			timeout	unsat	14.98	unsat	14.538
lfsr_004_159_080	unsat	31.69	unsat	19.10	unsat	36.94	unsat	STP	24.94	unsat	2.78	unsat	2.540
lfsr_008_095_064	unsat	88.91	unsat	38.86	unsat	56.61	unsat	STP	50.44	unsat	5.74	unsat	5.298
lfsr_008_111_080		timeout	unsat	117.12	unsat	116.53			timeout	unsat	9.38	unsat	8.729
lfsr_004_015_128		timeout		timeout		timeout			timeout	unsat	29.11	unsat	28.645
lfsr_004_127_112	unsat	65.43	unsat	44.96	unsat	76.60	unsat	STP	53.42	unsat	2.80	unsat	2.487
lfsr_004_159_096	unsat	32.56	unsat	35.88		timeout	unsat	Boolector	38.58	unsat	3.34	unsat	3.109
lfsr_008_111_096		timeout	unsat	118.93		timeout			timeout	unsat	11.28	unsat	10.518
lfsr_004_127_128	unsat	74.21	unsat	93.00		timeout	unsat	Boolector	88.09	unsat	3.34	unsat	2.963
lfsr_008_111_064	unsat	93.33	unsat	49.95	unsat	58.21	unsat	STP	63.25	unsat	7.37	unsat	6.963
lfsr_008_031_112		timeout		timeout		timeout			timeout	unsat	19.79	unsat	19.288
lfsr_008_095_128		timeout		timeout		timeout			timeout	unsat	13.52	unsat	13.044
lfsr_004_031_080	unsat	31.82	unsat	51.51	unsat	14.33	unsat	Z3	15.56	unsat	1.27	unsat	1.031
lfsr_004_095_128	unsat	48.26	unsat	62.53	unsat	81.41	unsat	Boolector	58.20	unsat	2.55	unsat	2.197
lfsr_004_159_112	unsat	55.59	unsat	49.88	unsat	88.95	unsat	STP	59.17	unsat	3.77	unsat	3.486
lfsr_008_047_128		timeout		timeout		timeout			timeout	unsat	10.10	unsat	9.660
lfsr_008_127_096		timeout		timeout		timeout			timeout	unsat	16.22	unsat	16.061
lfsr_008_111_112		timeout		timeout		timeout			timeout	unsat	11.13	unsat	10.405
lfsr_004_031_128		timeout		timeout	unsat	113.01			timeout	unsat	4.46	unsat	4.263
lfsr_008_079_128		timeout		timeout		timeout			timeout	unsat	10.67	unsat	9.711