Electronic Communications of the EASST
Volume 078 (2019)

8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation

-

Doctoral Symposium and Industry Day, 2018

Thoughts about using Constraint Solvers in Action

Malte Mues, Martin Fitzke, and Falk Howar

19 pages

Guest Editors: Falk Howar, Anna-Lena Lamprecht
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Thoughts about using Constraint Solvers in Action

Malte Mues[1], Martin Fitzke[2], and Falk Howar[3]

[1]malte.mues@tu-dortmund.de
[2]martin.fitzke@tu-dortmund.de
[3] falk.howar@tu-dortmund.de
Lehrstuhl für Software Engineering
Technische Universität Dortmund, Germany

Abstract: SMT solvers power many automated security analysis tools today. Nevertheless, a smooth integration of SMT solvers into programs is still a challenge that lead to different approaches for doing it the right way. In this paper, we review the state of the art for interacting with constraint solvers. Based on the different ideas found in literature we deduce requirements for a constraint solving service simplifying the integration challenge. We identify that for some of those ideas, it is required to run large scale experiments for evaluating some of the ideas behind the requirements empirically. We show that the platform is capable of running such an experiment for the case of measuring the impacts of seeds on the solver runtime.

Keywords: SMT Solving, Language Integration

1 Introduction

Cybersecurity is an emerging challenge for the IT industry and society as a whole currently. Protecting society from data breaches as for example the Equifax data leak in 2017 that leaked personal data for 50% of the US citizens is an important challenge for politic and industry today. In contrast, the current talent gap of people working in the sector is expected to reach 3.5 million open position in 2021 according to Cybersecurity Ventures[1]. Therefore, automation of security analysis is an important topic of ongoing research at universities and major companies (c.f. [Coo18, Bey19, God20]).

Most of them have to make at some point a decision as next step in the analysis algorithm used for security analysis. If the security of a program is analyzed, those decision problems are typically encoded as SAT or SMT problem (e.g. CBMC [CKL04], DART [GKS05], JDART [LDG+16], KLEE [CDE+08], PSYCO [GRR12]). Deciding Satisfiability of non-trivial formulae is in most cases reducible in linear time to 3-SAT and therefore NP complete. The single exception are problems reducible in polynomial time to 2-SAT as this problem class is solvable in polynomial time (e.g. with the algorithm presented by Aspvall et al. [APT79]).

Traditionally, there is an agreement that an efficient algorithm should be implemented in polynomial time. Given, that N is the input size of a problem, ideally, the worst-case complexity should be in $O(N)$. Anyhow, many algorithms generating values in the day to day business are not even close to this theoretical worst-case complexity. For example, the previously mentioned

[1] https://cybersecurityventures.com/jobs/

program analysis often uses some kind of constraint solver in the backend. Those solvers apply heuristics to decide the potentially NP complete problems in reasonable time frames and deliver monetary values with theoretically inefficient algorithms. In the remaining of the Introduction, we will briefly describe the usage of constraint solvers, define constraint problems, have a short summary of the history of SMT-LIB, and conclude the section with architectural challenges during the integration of SMT-LIB into algorithm implementations. At the end, we give an outline of the remaining paper.

Constraint Solvers. Today, there is a broad bouquet of constraint solvers powering the development of decision procedures for source code analysis. The most prominent open source constraint solver is Z3 [DB08]. But there are various modern SMT solvers available that are not less efficient (e.g. CVC4 [BCD+11], dReal [GKC13], Princess [Rüm08], MathSAT5 [CGSS13], SMTInterpol [CHN12], Yices 2 [Dut14], Z3Str3 [BGZ17]). All of these constraint solvers have in common that they are capable to decide satisfiability of constraints. A major part of modern constraint solvers uses some kind of Conflict Driven Clause Learning (CDCL), which is a framework that has grown in a series of improvements over time to the DPLL algorithm [DP60, DLL62] for solving satisfiability problems. CDCL uses the backtracking similar to DPLL and combines this with heuristics to determine the next search direction whenever a decision is required (cf. Kroening et al. [KS16, Chapter 2.4]). CDCL is not efficient from a theoretical point of view. Nevertheless, due to the recent work in the area of constraint solving, many practical problem instances are solved in acceptable time windows. The SMT solver community tracks the progress made regarding constraint solving at the SMT competition called SMT-COMP.[2]

Getting back to software analysis tools using SMT solvers, we are able to observe a general architectural pattern: They collect logical constraints describing the current source code under analysis and feed them into a constraint solver at some point in the analysis. Sometimes these constraints are further combined with verification conditions before passing them to a constraint solver. While some tools need to resolve one large constraint problem at the end of the analysis (e.g. CBMC, a bounded model checker for C) targeting an unsatisfiable result, other tools call repeatedly a constraint solver in the background as part of the analysis (e.g. JDART, a concolic execution engine for Java). CBMC will only query the constraint solver more than once if analysis violation leads to satisfiability results. These repeated calls are required to generate models and explanations as counterexamples for the analysis target. Each call excludes a violation of the verification property found before. JDART actually needs the interaction to continue exploring the search space during analysis. The advantage of repeatedly calling a constraint solver during analysis is that the solver might reuse some of the results computed in a previous decision. This methodology is called incremental constraint solving.

Constraint Problems. In this paper, we use the term *constraint problem* to express any logical formula consisting of variables, eventual unary or binary operators valid in a theory associated with the type of the variables and logical combinators (and, or, xor, not, implies, and quantifiers). If we are speaking of a constraint problem encoded for a SAT solver, the problem has been reformulated to a pure Boolean variable problem which is decidable by a SAT solver. Whenever

[2] https://smt-comp.github.io

we refer to a constraint problem passed to the SMT solver, we assume that the SMT solver supports a fitting theory. As the concrete types and operations allowed in a constraint problem also depend on the type support of the used SMT solver, we will not formalize this expression further. For the ideas presented in this paper, it should be sufficient to keep in mind that a constraint problem is always solvable by either a SAT or a SMT solver, depending on the context and required theory. The main difference is that SMT constraints allow a higher-level description than SAT constraints as more expressive theories than the Boolean theory are allowed.

The SMT-Lib Language. The SMT solver community recognized quite early that they need a common input language to all the different SMT solvers. Ranise and Tinelli supposed the SMT-LIB in close collaboration with the SMT-LIB interest group in 2003 [RT03]. The original mission statement for the SMT-LIB was: "We believe that having a library of benchmarks will greatly facilitate the evaluation and the comparison of these systems, and advance the state of the art in the field [...]" [RT03, p. 1]. From the perspective of an SMT solver's user, the benchmarking focus has been important for pushing the solver's capabilities further providing at the same time a bookkeeping of each solver's capabilities. The more important aspect of SMT-LIB from a user's perspective is that SMT-LIB has been developed towards the primary industry standard interface for interacting with SMT solvers on command lines. SMT-LIB is a textual input format and solvers are often called by interprocess communication. As a consequence, the separation between the analysis and the source code becomes a problem. The analysis is no longer part of the programming language, but instead exists only in value assignments of variables. Therefore, program analysis gets harder. For example, the SMT-Lib syntax of analysis rules is not checked during compile time, as the compiler does not consider the string values as part of the language.

Integration Challenges. At the end of the 20th century, the term 4th generation language (4GL) has been coined for a new generation of languages. DSLs aiming on solving a certain problem, as for example interacting with a database, are considered to be a 4GL. The hope was a productivity gain by reducing the program complexity in exchange for eventually more CPU and memory utilization (cf. the case study by Lehner [Leh90]). Prominent members of the 4GL family are regular expressions for parsing strings and SQL as a database interaction language. There has been drawn the conclusion recently that SMT-LIB has established in a similar way as SMT solver frontend as SQL is established as a database frontend language (cf. [KFB16, HJM19]).

Anyhow, we are convinced that it is not sufficient to establish SMT-LIB as a common interface format. SQL is a fourth-generation language and such is SMT-LIB. Both have in common that they are used in other programs written in 3rd generation languages (or more frequent referred to as high-level languages) as for example JAVA or C++. This means, we keep most of the problems for integrating a fourth-generation language into programs with SMT-LIB.

We have seen this for using regular expression in programming languages as well as for using SQL in programs. Especially for SQL, it has not been sufficient to build an object representation for SQL into a programming language as there remains the question how to serialize a complex object structure into a relational table. This problem is commonly known as the *object-relational impedance mismatch*. Instead, higher abstraction levels for integration have been developed as for example Hibernate[3]. Another branch of research tried to integrate SQL directly into the

[3] http://hibernate.org

Language compiler (c.f S4J [RLO16] or ScalaQL [SZ09]).

SMT-LIB has a similar problem. While it is not required to serialize an object into a database, it is required to find a way for reasoning about the object in a given theory. Once such a reasoning is defined, it is still required to express this as an SMT-LIB string and pass this string to a SMT solver for processing. Once the solver terminates, the result of the solver has to be deserialized into data structures useful to the program logic. While most decision problems encoded in a logic and passed to a solver are expected to be answered in a Boolean *yes* or *no* style, most solver designer allow that the solver refuses solving a task and responds a *do not know* answer. This is a very simple example for what might be called a *inter-theory impedance mismatch* in the decision procedure world similar to the *object-relational impedance mismatch* in the database world. In addition, information provided in the answer model of the solver needs to be translated back into the original programming language to reuse them in later queries against the solver or make decision outside of the solver instance. Especially, if the underlying theory is not easy representable with hardware data types, as for example decision results based on the real theory using real numbers. This introduces additional challenges for aligning the SMT-LIB world with the surrounding programming language not present for other 4GL languages.

Outline. In this paper, we will sketch out our perspective on the challenges for integrating SMT-LIB into industry grade programs. Therefore, we will first point out in Section 2 the state of the art of integrating constraint solvers into programs and current proposed solutions. In continuation, Section 3 describes additional required steps from our point of view. We designed an experiment platform for addressing some of the question raised in Section 3 empirically. This platform is described in Section 4. Section 5 presents some first results. Finally, Section 6 concludes the paper.

2 State of the Art

As SMT solvers power various new research areas and restimulate research branches such as symbolic execution, there exist plenty of publications around algorithms for solving satisfaction problems and optimizing the decision procedures for special cases (cf. [BHM09]). In this paper, we will focus on the software engineering aspects of interacting with SMT solvers in other analysis algorithms and therefore skip literature describing SMT solver internals.

The discussion about SMT-LIB and SQL is not the only parallel in the comparison of the usage of SMT solvers and database engines. Databases are only able to deliver their optimal performance if chosen and configured appropriately for the problem at hand. Each database system has a very different characteristic. This starts from the general design of data storage as a NoSQL, row-store or column-store database and ends with specific fine-tuning configuration options for each database system. Once the right engine is chosen, defining the right database schema layout along with efficient caching indices is still crucial for the resulting overall performance.

SMT solver tackle a different challenge than databases, but many different groups have developed small improvements for speeding up a certain edge case of constraint solving just by improving the way how they interact with constraint solvers. We first present the two main ways of interacting with a 4GL engine in both domains: interprocess communication and language bindings. A subsection on access patterns for SMT solvers concludes this section.

2.1 Interprocess Communication

A popular way to communicate with SMT solvers across processes is communication over pipes. Often, a newly spawned thread executes the solver binary and exchanges the problem and the results using pipes with the main analysis thread. This is comparable to call a database using the database command line interpreter. The interpreter is a small binary capable of receiving strings on the standard input pipe and is therefore controlled by a string-based language. Any result produced by the binary is then printed on the standard output pipe. Due to the string based data transfer between processes, this communication pattern requires good serialization and deserialization support for the data at hand.

Most SMT solvers support a SMT-LIB compliant frontend and are able to consume SMT-LIB constraints directly from the command-line or from a file, as SMT-LIB is the standard for textual constraint representation. Serializing a higher-level object into a SMT-LIB string makes it impossible to reuse already available information in the constraint object for optimization. Therefore, any potential optimization that uses domain specific information in the object structure of the constraint has to be integrated into the serialization method. The SMT-LIB frontend for each solver is comparable to the Open Database Connectivity (ODBC)[4] interface for databases in the way that both approaches use a string based language for controlling the underlying technology.

A downside of this approach is that some new features are not yet part of SMT-LIB. Hence, some solvers, like SMTInterpol [CHN12], require new SMT-LIB commands for special features like calculating interpolants. They added *get-interpolants* as a valid command to the SMT-LIB query language of SMTInterpol. This extends inter process communication using the SMT-LIB to extended features, at the cost of increasing the amount of SMT-LIB dialects that a state-of-the-art SMT parser has to maintain. Otherwise, interprocess communication is not capable of using multiple different solvers to their full potential. For the case of *get-interpolants*, other solvers later adapted the same semantic underlining the need for a shared input language across solvers.

Cok introduced JSMTLIB [Cok11] as an abstraction layer for interacting with SMT solvers in JAVA. JSMTLIB is an example using interprocess communication over pipes to access a solver. The tool consists internally of an abstract syntax tree (AST) as main data structure for representing constraints and a parser that converts between the textual representation in the SMT-LIB format and the JSMTLIB AST. The library does not provide any higher-level views or abstractions to the constraint problems. But it is designed to support different solvers and allows to address subtle divergence between a solver's behavior and the SMT-LIB standard.

Therefore, JSMTLIB allows the formulation of constraints by creating elements of the AST and combining these parts into the final query. This query is then passed to the solver and the answer parsed using the SMT-LIB parser suitable for each solver. The advantage of this approach is that any SMT-LIB compliant solver might be plugged in to a program at the cost of a slim adapter class for the solver. This adapter creates a process running the solver and takes care of the communication with the solver. Once this adapter is available, the solver can be selected in the configuration as solver used for the JSMTLIB instance. Anyhow, it is still up to the user to select the right solver for the problem at hand. The official SMT-LIB website[5] names six other tools for parsing SMT-LIB (supporting ANTLR, c99, Haskell, OCaml and SWI-Prolog).

[4] e.g. http://www.unixodbc.org, on of the state-of-the-art projects evolved form the SQL CLI Standard [VP95]
[5] http://smtlib.cs.uiowa.edu/utilities.shtml

2.2 Language Bindings

Using a language binding API of an SMT solver is the other main way to interact with a solver. The advantage is that any decision data structure is directly mapped to data structures in the solver API. Therefore, the serialization and deserialization step is no problem, avoiding also any trouble with SMT-LIB dialects. The downside is that the comfort of an universal exchange format in textual form like SMT-LIB is lost. Using the complete available API and not only the exposed subset of functionality that is exported through an SMT-LIB frontend might nevertheless be desirable. Therefore, different projects established a native connection to solvers using the corresponding language bindings.

As most of the solvers are written in C++, a truly native connection might also be written in C++. SMT Kit[6] provides a C++ library for interacting natively with Z3, CVC4 and MathSAT5.

In the Java world, jConstraints [HJM19] and JavaSMT [KFB16] are both developed as an intermediate Java API towards constraint solvers. We compare them briefly. At the moment (May 2020), JavaSMT supports MathSAT5, SMTInterpol, Z3, CVC4, Boolector and Princess according to its website[7]. jConstraints supports a slightly different subset of solvers. Apart from Z3 and SMTInterpol, jConstraints supports dReal, Coral and an implementation of the concolic walk algorithm [DA14]. Further, representing SMT constraints in an expression tree structures of Java objects is a shared idea between both libraries. This expression tree might be modified using visitors allowing a powerful way to modify expressions in a controlled way. The visitor also allow to implement domain specific optimizations on the constraint structures. Nevertheless, jConstraints and JavaSMT have been developed with subtle different focuses. As described in detail in the JavaSMT paper [KFB16], Karpenkov et al. aimed for a library allowing easy interaction with a constraint solver at optimal speed. To archive this, JavaSMT ships with its own JNI-library for accessing the Z3 binaries optimizing memory strategies [KFB16, Section 4]. In addition, the configuration amount required to exchange a SMT solver should be minimized.

jConstraints main focus was an easy and efficient way to store and manipulate constraints and has been designed for speed in the second place. As it was designed initially along with jDart [LDG+16], the main focus was the constraint management during jDart's analysis. To achieve the usability during JAVA analysis, an evaluation model has been designed as part of jConstraints that models native type's semantic in JAVA. This evaluation model evaluates that a given constraint solver solution aligns with the JAVA type semantic. This evaluation model is the core feature of jConstraints opening the field for decision procedures speeding up the decision using a problem relaxation before solving (cf. FEAL [HJM19]).

pySMT [GM15] is a robust Python frontend to various SMT solvers combing the ideas behind JSMTLIB, jConstraints, and JavaSMT in a single library. It creates some kind of DSL for integrating SMT into Python and allows a lot of constraint simplification and modeling support in Python. The library supports interaction with Z3, MathSAT5, CVC4, Yices, Cudd, PicoSAT and Boolector using the Python language bindings. In addition, there is a general-purpose communication support for an arbitrary SMT-LIB compliant SMT solver comparable to the solver communication of JSMTLIB. To do so, pySMT also provides a parser for SMT-LIB as one component.

[6] http://ahorn.github.io/smt-kit/
[7] https://github.com/sosy-lab/java-smt

Additionally, pySMT allows portfolio solving. The idea of portfolio solving is running more than one solver in parallel on the same problem. Based on the observation that different solvers have different weaknesses, but also strengths, the hope is that one solver returns faster than others and therefore speeds up the execution. The very same approach has been discussed as an idea for the solver layer of the symbolic execution solver in KLEE [PC13] which is a symbolic execution engine for LLVM. They used the metaSMT [RHF+17] framework for the realization of a portfolio solver. metaSMT connects various different solvers to C++ applications using the native API of each solver. It provides a DSL for generating constraints in a solver independent way and translate them for each connected solver to the native API.

Up to this point, the described solutions are comparable to SQL query builder libraries as for example JOOQ[8] or querydsl[9]. Next, we will shortly examine further ideas for solver interaction.

2.3 Access Patterns

Apart from the above described features pySMT also provides caching of constraint solutions. Therefore, constraints are normalized and then stored with the corresponding solution in a store. Klee [CDE+08] has also built a constraint cache internally as a frontend to metaSMT and demonstrated that caching queries instead of passing all of them directly to the SMT solver speeds up analysis algorithms. For the JAVA world, the Green [VGD12] solver introduced the idea of constraint caching in a JAVA library. It consists of a constraint representation that can be canonicalized. Further, Green defines a way for looking up previous result maintained for these canonicalized form in a cache. The framework has been used successfully in SPF [PR10] as constraint cache and solver access layer.

Last but not least, Clarke et al. [CKL04] introduced CBMC as a bounded model checker for C. It is a prominent tool using constraint satisfaction as part of their reasoning. During the symbolic execution of a program, CBMC collects constraints in its own internal representation. Once the symbolic execution engine is done, the constraints are converted to fit a backend constraint solver and this solver is invoked to solve the decision problem instance resulting from the analysis. At the moment, mainly SAT solvers are supported. The SMT integration is currently under a rewrite. Anyhow, they have a complete implementation of Tseitin-transformation, allowing the conversion of their SAT encoding into the conjunctive normal form (CNF) before passing it to the solver. It has not been demonstrated, whether this speeds up query time in general or not.

To the best of our knowledge pySMT is the only library at the moment that is able to select the right solver based on a set of available solvers. This is the first step in the direction of an automated solving service that chooses automatically a solving strategy for a given constraint problem. Therefore, it combines and selects solvers form the available SMT solver bouquet using heuristics. This is comparable in the idea to an autonomous database manager that is able to work with different data bases deciding on its own which is optimal used for which purpose. We are convinced that the managed service for constraint solving might overcome a couple of problems managed services for databases are not able to solve due to consistency and persistence requirements. In the following section, we will derive some requirements for a full featured constraint solving service and motivate them from a software engineering point of view.

[8] http://www.jooq.org
[9] http://www.querydsl.com

3 Requirements for a Constraint Solving Service

Summing up the above described ideas, we already find evidence in the literature that investing into an easy to use constraint solving frontend is worth it. In addition, major software companies start adopting constraint solver-based tools for maintenance task in their infrastructure (e.g. [Coo18]). Z3 originates from Microsoft Research and is actively developed there until today. Nevertheless, we have seen that especially pySMT is way more feature rich than the Java and C++ libraries. We think that pySMT is the first step toward offering a constraint solving service instead of just a DSL for formulating constraint problems. From the above examples, we identify following points as an integral part of a constraint solving service to make it competitive with the state-of-the-art solver abstraction libraries:

- Support for close to all available SMT solvers to catch up with all the different performance improvements and avoiding eventual solver lock-in with an application.
- Deserialization and serialization support for SMT-LIB in the most current version.
- A constraint problem DSL that integrates well to the target programming language of the constraint solver service and allows the definition of constraint problems including a type semantic conversion for native types.
- Powerful standard manipulations of queries like establishing CNF, replacing variables, etc.
- Solver strategies and portfolio solver infrastructure allowing to push the complete constraint solver bouquet to its limit using all available hardware resources.
- A caching framework for queries to reduce the number of solver invocations. This framework should further interact with eventual incremental solving stacks allowing to reuse a partial solution.

The envisioned architecture is sketched out in Fig. 1. A user should use the provided constraint DSL to formulate a constraint problem. This constraint problem is passed to a service manager. The service manager needs to support as much constraint solvers as possible, in order to select between them and to choose one that is capable to solve the problem at hand. The choice is made based on empirical evaluated auto configuration rules.

CVC4 [BCD+11] already explains in the accompanying paper that there is more than one way to compile the binary allowing to use or suppress certain theorem proving behaviour. This is one example for a complicated solver configuration option. Most likely, a single user will not invest the required time to finetune the solver interaction layer to interact with different compiled binary versions. It is also possible to configure and fine-tune some of Z3 parameters. Running *z3 -p* lists more than 400 configurable parameters influencing Z3 and the integrated theorem engines. Concentrating the solver configuration knowledge in a central place that is used across different projects allows the community to share the solver tuning knowledge. This implies that progress in solver speed is faster delivered across projects.

Depending of the constraint problem size, it might further be desirable to distribute the solver across different machines. Therefore, a network interface between the solver and the manager

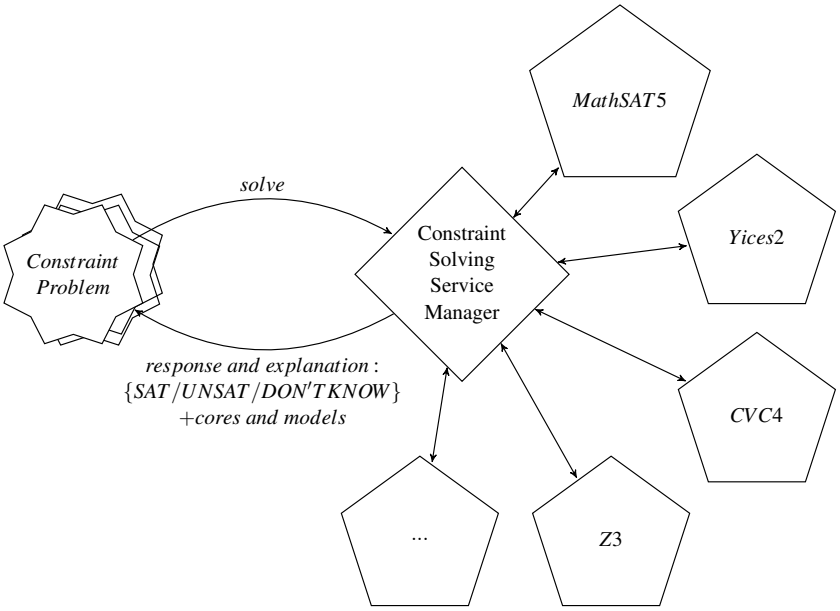

Figure 1: Vision for a constraint solving service architecture. The user should only interact with a single service component demonstrated in the center. The constraint problems are passed to the central service manager and somewhen in the future, the manager answers with a response and additional resources if they are requested. All interaction with solvers is done by the manager.

node will be required. We want to investigate this kind of workload distribution in the future to refine our solver platform idea. If the solver is expected to spend a few hours on the problem, it might be worth to pay the overhead for problem distribution and result collection. This might also allow analysis algorithms using the first result available for continuing computation while the slower solvers are running in the background. Once all solvers terminate, the results can be compared later identifying potential mismatches. In case of a mismatch, the analysis might be spurious and the result might be persisted by the solver manager for further research. This way, new real-world examples of hard to solve problems for state-of-the-art solver might be collected.

Apart from the pure decision result, other artefacts have to be collected and provided to the user as well. These artefacts depend on the formulated request, but might consist of interpolants, satisfaction models, minimal unsatisfiable cores, or simplified formulas. The DSL used as input language must be flexible enough for also encoding these results in an easy to use way.

Independent of the result encoding, the developed DSL for formulating constraint problems should contain DSL primitives that already encode the default semantic for native datatypes in different high-level languages. For languages like JAVA, where the semantic of a native type is well defined, a semantic mapping should be defined in the DSL definition. This mapping might later be used to create variables in the SMT problem that allow to reason about the original source code variable. Especially the design of source code analysis algorithms would benefit from such a mapping. For languages like C, where the semantic mapping allows some flexibility, a default should be defined. It is possible to select a standard architectural mapping of int, long, char, etc.

```
1  static void foo(
2      int a, int b){
3    int c = 8
4    if (a < 6 || b > 4){
5      c = a + 3;
6    }
7    assert c > 9;
8  }
```

```
(and (= c_0 8)                          1
     (or (< a_0 6) (> b_0 4))           2
     (= c_1 (+ a_0 3))                  3
     (> c_1 9))                         4
```

Figure 2: A small example and the CNF encoding of the happy path in the *foo* method.

and make this easily configurable to other architectures if needed. This allows to start with a reduced configuration option for the initial architecture selection in the DSL layer and fine tune the exact semantic as needed.

Transforming normal SAT problems with Boolean variables into CNF is a well understood problem and for example CBMC supports this internally. So far, we haven't found an implementation for a standard conversion to CNF in one of the constraint libraries. During the development of jStateExplorer which is used in PSYCO [HGMN18], we figured out that it is desirable to get this kind of transformations as an additional feature. Some parts of constraint might be pruned, because they are created about variables in the original system that are no longer of interest after passing a certain point in the analysis. Consider for example the constraint in the right side of Figure 2. The constraint encodes the happy path for the method *foo* passing the assertion at the end of the method. The given representation makes it easy to reason about the subclauses leading to the contradiction between $(< a_0\ 6)$ and $(> (+ a\_0\ 3)\ 9)$. As a consequence, none of the successful execution does apply the value bound on a_0 in the if condition. Hence, the left part of the or-clause can be removed. It is a small example, for an optimization that is easily conducted in the CNF format. We are convinced that there are more small optimizations that might reduce constraints and also variables in constraints over time.

From our point of view, it is worth to discuss the right language for developing such a constraint solving service. As many solvers are written in C++ and some other solvers like SMTInterpol are pure JAVA libraries it is unclear what is a good main language. JavaSMT has already shown that it might be possible to came up with efficient JAVA bindings using the JAVA Native Interface (JNI) but that it also might be worth to question the prebuild language bindings. As we are planning to continue the development of jConstraints as a solving service for JDart, we will stick with JAVA for now. Nevertheless, it would be interesting to see a feature equivalent C++ version for benchmarking purpose in the future.

We mentioned in various points that it is required to make decisions for the right solver evaluation based on empirical results. Such a decision requires the capability to run a large amount of experiments in an efficient way, comparable to batch-job processing. In the remaining of this paper, we will report on our approach for developing an experiment platform and try to answer the question, whether seeding Z3 with different seeds might be a successful strategy for a portfolio solver or not. This question is one example for a potential optimization in a solving service,

that need some empirical input. A design goal of Z3 is stable seeding and robust implementations to minimize effects of seeding. Anyhow, internally Z3 uses heuristics that are influenced by randomness and therefore also effected by seeding. Turning this design knowledge into runtime benefits by utilizing hardware more exhaustively requires to investigate the solver behavior empirically in upfront. Next, we describe the design of a batch execution platform allowing such experiments for generating data about a solver.

4 A Platform for Running Experiments in the Cloud

The general problem, we are trying to solve is running the same binary with different configurations passed on the command line. Following the current trend for packaging and running applications, a Docker image containing the experiment has been built. The Docker image can be run and the configuration required passed as command line parameter to the Docker image. Further the docker daemon can be configured to enforce resource limits using Dockers cgroup support. Cgroups are a Linux kernel technique for limiting the resources of a process and its children. This turns the execution of the experiment into running Docker containers with resource limits and a set of command line parameters. In order to achieve some throughput, this should be managed across different machines. To the best of our knowledge, the only tool available offering this service of batch processing used to be AWS Batch [10] by end of 2018. As this is a closed source service behind a pay wall, it is not available for our experiments. During 2019 kubernetes introduced kubernetes tasks. A feature delivering the same benefit. But is has been released after we completed the experiments.

We are aware, that using a Docker image might not be the best available possibility for reproducibility of experiments in the future. Using the BenchExec framework provided by Wendler et al. [WB19] might have resulted in a better reproducible measurement if we are only interested on the performance of a single parameter. Hence, for the particular supposed experiment in this paper, a BenchExec setting might have been better. Beyer et al. [BLW19] describe the ideas behind the BenchExec approach in more detail. The main feature they point out, is pinning processes to dedicated cores avoiding effects in measurement due to cache misses. Additionally, changing CPU frequencies due to workloads is disabled on their CPUs by default for reducing the effects on the measurement. We will call the concept of changing CPU frequencies Boost Clock as every hardware manufacturer has a particular name for this technology. In addition, Beyer et al. vote for not using hyperthreading (in the sense of simultaneous multithreading) as this might slow down the execution. Instead of disabling hyperthreading on the BIOS level and therefore changing the actual CPU behavior, the operation system is used to ensure scheduling of work is only on one of two vCPUs (or threads) bounded to a physical core. By design, hyperthreading is only expected to have a positive impact on I/O intensive workloads. As BenchExec is designed to measure CPU-bound tasks, it would be more adequate to run it on CPUs with disabled hyperthreading to make sure there are no side effects of the supposingly idle vCPU on out-of-order execution.

While disabling Boost Clock makes an experiment more reliable, our focus is the overall wall clock time for the end user, and Beyer et al. demonstrate clearly a significant performance

[10] https://aws.amazon.com/batch/

drop for disabling Boost Clock. Hence, if a verification method is allowed to be running for example on 4 vCPUs, which might be simulated by running a Docker container with the *–cpus* flag, we want to see effects of using these CPU resources as efficiently as possible for lowering the overall wall clock time. That's why we started to design the solving service as most state-of-the-art verification methods are backed by a solver and getting CPU time is still comparable cheap on modern multicore systems. Docker uses standard Linux cgroups in the background for limiting resource access. Pinning tasks fix to always the same CPU core might not allow to use the complete thermal reserve while boosting the CPU. Therefore, we are not supporting any approach for core pinning at the moment in our execution platform. Nevertheless, Docker has a *–cpuset-cpus* flag allowing a restriction of the vCPUs of the host system used for running the container. In the same way, Docker allows to setup cgroups in the background for limiting the memory usage. Resource constraints are therefore enforced while running a Docker image in the same way BenchExec enforces them on a Linux machine.

In the future, running experiments for a solving component following the design in Figure 1 is likely to communicate over a network with different parts of the architecture, unless everything is fitted in the same physical CPU. Therefore, the planed future direction of the empiric measuring task is not fitting the target of the BenchExec software right now. We observe more effects introduced by I/O bound interaction as accessing a network interface. From our point of view, running different solvers in different docker container and communicate between them within a virtual defined network as part of an integration test, seems easily doable with modern cluster orchestration engines for an end user. We are aware, that the results obtained are likely influenced by measurement inaccuracies due to effects in the cluster. Nevertheless, we think this is a well-suited setup to detect general trends by repeating experiments and figure out whether there are single problem instances, which should be investigated in more detail in future work on a high precision measurement system for benefiting the solver development. In the meantime, we can use the improvements gained by the solving service running it on docker containers independently of eventual single solver improvements.

Given a cloud platform consisting of different machines for running the experiments, a platform running the Docker containers utilizing the cloud hardware efficiently is required. As part of the reported work, a Docker based experiment execution platform has been designed. As said, a single experiment consists of a Docker image along with a set of configuration parameters passed on the command line. Bundling the image in a description together with the command line parameter turns all the experiments into executable and isolated tasks. Isolation is needed to prevent software dependencies from disturbing one another. To structure the experiment further, these tasks can be grouped into single jobs for the experiment execution platform. The bundling and configuration is done using Docker Compose files. These Docker Compose files can easily be run in a Docker Swarm as executor. With Docker Swarm we can run multiple containers each representing one sub experiment on one or more machines. In case images need to collaborate for the execution of the experiment, the Docker Compose file might be used to describe a stack of docker container collaborating together and how to wire a network between these containers. In the presented experiments one Docker Compose file describes all runs executed for a single benchmark file as a set. Anyhow, Docker Compose files can be used in theory to describe arbitrary tasks as long as they are expressible as a Docker Compose stack. Unfortunately, Compose files alone are not sufficient to describe the required experiments. They do not cover all needed

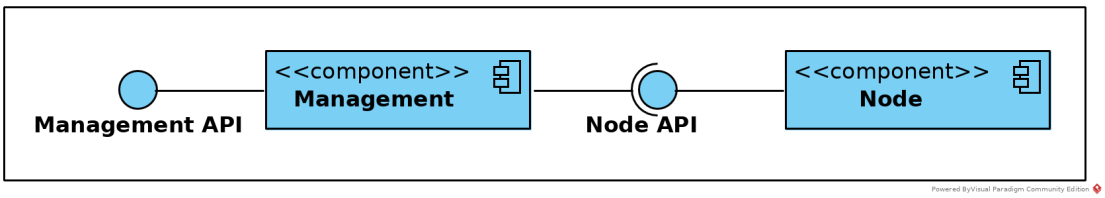

Figure 3: Platform architecture

requirements as a maximum runtime and credentials for private Docker Repositories are not covered yet, but are required for pulling images from a private repository. Therefore, the Job file was designed that encloses the Docker Compose file and contains additional information.

Each container in a stack can be configured individually and those configuration options also include hardware requirements that are later enforced on each host machine by the Docker layer below the Docker Swarm. Docker takes care of setting up cgroups on each host to meet these hardware limits. The experiment execution platform designed takes care of bundling available machines into a Docker Swarm cluster. During this task, the hardware requirements in the Docker Compose files are respected to efficiently use the available hardware resources. Once a job definition is received by the platform, a Docker Swarm of a suitable size is created and the compose file deployed to the Docker Swarm. A Docker Swarm consists of Managers and Workers. Managers receive and process the Compose files and distribute the tasks on their Workers autonomously. A Worker receives a task, pulls the needed container image, and runs it in a container. The platform designed uses REST calls for job submissions and result retrieval once the job is completed. Between the Docker Swarm cluster and the REST endpoint, a component has been added that parses Job files and takes care of credential handling and time limit enforcement.

Next, this additional component is described in more detail. It consists of two components, the management component and the node component as shown in Figure 3.

The Management component controls and organizes the platform. User interaction is done through the Management API and users can send jobs and manage the registered worker machine representing a node in the platform. When a job is received the platform adds it to a queue and schedules it for execution. The queue is worked off sequentially. For every job the Management calculates which resources are needed in sum and tries to build a fitting sub cluster of machines from the available nodes. If the Management node finds a suitable subset of machines in the cluster, it triggers the creation of a Docker Swarm on this sub cluster. Once the new Swarm is online, it sends the job to the swarm manager and marks all nodes of that cluster as unavailable for other jobs. The Swarm runs the job to completion. Once the job is completed, the results are pulled from the cluster node and made available for the users through the central Management API. This is required to extract the results from the docker container so that the container can be destroyed. The Swarm terminates leading to the release of all resources in the sub cluster. The Management node marks them as free again. From now on, other jobs might claim these Nodes for job execution. Whenever sufficient resources are available for a job in the queue, the Management repeats all those steps for computing the results leading to one swarm per job.

The node component is responsible for running the jobs and therefore able to utilize the Docker

Id	Problem	Same Seed		Multi Seed		Difference (Same - Multi)	
		SD	Mean	SD	Mean	Mean	SD
1	/45vars/v45_problem_2__008.smt2.slack.smt2	11.49	7149.0	5.08	7140.0	9.0	6.41
2	/size50/c50_problem__006.smt2.slack.smt2	9.22	253.0	7.50	238.5	14.5	1.72
3	/size100/c100_problem__004.smt2.slack.smt2	13.30	238.0	7.64	231.5	6.5	5.65
4	/size100/c100_problem__006.smt2.slack.smt2	9.78	184.5	2.55	181.0	3.5	7.23
5	/size60/c60_problem__006.smt2.slack.smt2	16.15	311.0	2.93	299.0	12.0	13.22
6	/size60/c60_problem__005.smt2.slack.smt2	13.54	7281.0	4.54	7274.5	6.5	9.01
7	/size80/c80_problem__005.smt2.slack.smt2	6.06	7349.0	3.20	7344.0	5.0	2.87
8	/size20/c20_problem__003.smt2.slack.smt2	5.95	183.0	4.16	173.0	10.0	1.79
9	/size20/c20_problem__004.smt2.slack.smt2	15.55	186.5	7.45	178.0	8.5	8.10
10	/size20/c20_problem__006.smt2.slack.smt2	10.46	258.5	5.69	250.0	8.5	4.78
11	/size10/c10_problem__006.smt2.slack.smt2	9.64	150.5	2.17	143.0	7.5	7.47
12	/size10/c10_problem__001.smt2.slack.smt2	5.32	53.5	4.29	52.5	1.0	1.03
13	/size10/c10_problem__004.smt2.slack.smt2	9.52	7146.0	1213.12	6435.0	711.0	-1203.60
14	/size10/c10_problem__002.smt2.slack.smt2	8.98	109.5	6.93	109.0	0.5	2.05
15	/size10/c10_problem__005.smt2.slack.smt2	9.37	193.0	12.45	182.0	11.0	-3.09
16	/size90/c90_problem__004.smt2.slack.smt2	7.24	206.0	6.89	198.0	8.0	0.35
17	/size90/c90_problem__005.smt2.slack.smt2	6.91	269.0	8.24	261.0	8.0	-1.33
18	/size90/c90_problem__002.smt2.slack.smt2	13.06	564.0	7.02	564.5	-0.5	6.04
19	/size30/c30_problem__003.smt2.slack.smt2	18.76	513.0	9.34	491.0	22.0	9.42
20	/size30/c30_problem__002.smt2.slack.smt2	8.90	253.0	6.63	246.0	7.0	2.28
21	/size30/c30_problem__006.smt2.slack.smt2	11.66	173.5	5.25	165.0	8.5	6.41
22	/45vars/v45_problem__001.smt2.slack.smt2	12.26	131.0	3.11	112.5	18.5	9.15

Table 1: Experiment-Report: Problems are taken from the QF_LIA[8] category of the SMT-LIB benchmarks. We are reporting the standard deviation (SD) and mean for each experiment set. In addition, the difference between the same seed and the multi seed experiment is reported.

client on the machine. On the sub clusters are, as already mentioned, Docker Swarms created that can be dynamically build and terminated by the management. The Node API of the Swarm manager receives the job and deploys it on the other nodes in the Swarm. Every task is distributed inside the Swarm and executed in a container. After a task is done the manager node pulls the result from the worker node. After all workers finished their task, the results are bundled on the swarm management node and wait to be transferred to the central platform storage on the platform management node.

In conclusion, the platform is able to run many jobs in parallel and automated. The number of parallel jobs depends on the available hardware and requested hardware requirements for a job. As a result, it is possible to automatically run a large set of experiments on a cloud infrastructure and keep the user interaction as minimal as possible, so that the focus can be kept on results and their evaluation, which we will describe next.

5 Experiments and Evaluation

jSMTLIB has been integrated into jConstraints as part of the reported work. This allows to evaluate solver timings empircally on SMT-Lib benchmarks running with jCon-

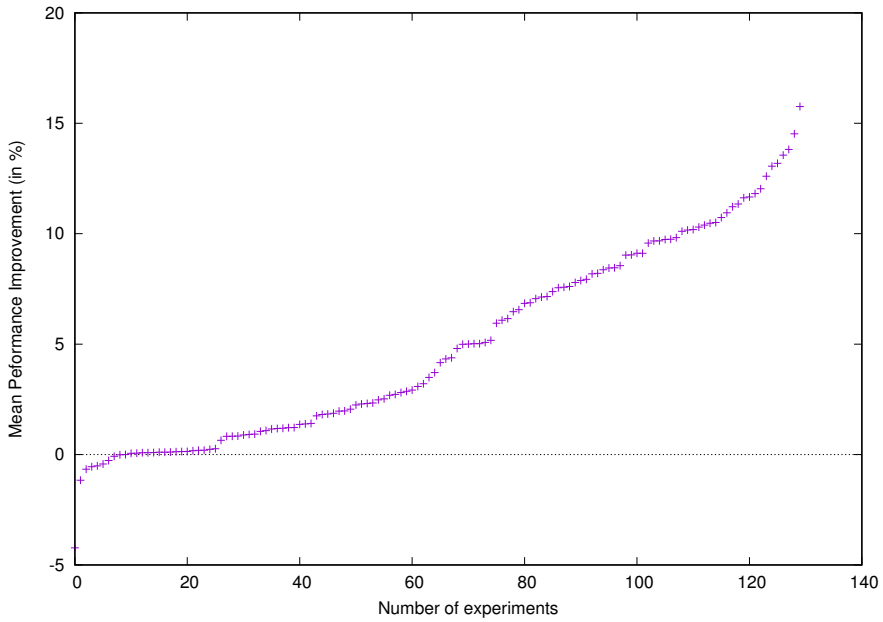

Figure 4: The plot demonstrates the performance gain between the mean over 10 Z3 invocations with different seed values compared to running the solver 10 times with the same seed. The performance improvement is reported in percent. It is visible that sometime the single seed is substantial better than the mixed seed set, if the experiment ended up on the left side, and there are other seed values improving the performance, if the experiment ended up on the right side.

straints. For the experiments reported in Table 1, we used SMT problems located in the `QF_LIA/20180326Bromberger/more_slacked/CAV_2009_benchmarks/smt`, `QF_LIA/20180326Bromberger/more_slacked/CAV_2009_benchmarks/coef-size/smt/` and `QF_LIA/20180326Bromberger/more_slacked/cut_lemmas/` folders of the `QF_LIA`[11] benchmark. We packaged them along with the jConstraints-Jar[12] in a runner that accepts a seed value and a path to an SMT problem instance. This runner is deployed as docker image. For this paper, we also included the jConstraints-z3 plugin to the docker image.

In a first step, we have run each of the SMT problems ten times with the same seed[13], and ten times with different seeds[14] using the Z3 solver only. The seeds are selected randomly by a human being without any planed pattern. For each of the runs, 8 cores and 16 GB of RAM has been provided. We used a cluster of 10 virtual machines hosting in total 400 cores and 700GB of RAM to crunch through the tasks. These virtual machines are running on a 19 node cluster with 1216 Cores (AMD Opteron 6272) and 4864 GB RAM in total providing OpenNebula for VM management and KVM with a Linux kernel as hypervisor.

[11] https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LIA
[12] The source code is publicly available: https://github.com/mmuesly/jConstraints-Runner
[13] [1234]
[14] [124, 1325, 14253, 164, 145308, 13257, 1325156126412, 142357, 732, 1234]

Table 1 reports the range between the highest and lowest runtime in addition to the mean runtime across each set of 10 runs. It represents a randomly selected subset of the 131 benchmark problems we have run our experiment with. In the right column, we report in addition the difference between the mean runtime using one seed and the mean runtime using multiple seeds.

A positive number means that the mean time using multiple seeds has been faster than the mean time using a single solver. But in addition to the difference in the mean times, also the span between the fastest and the slowest solver run is important. In case, we really observe an improvement based on a seeded heuristic, we expect this difference to grow too. The only case, where we observe the expected behavior is the problem with id `13`. Z3 further aims on stability in the heuristics. This means that the goal of the Z3 developers is met, if seeding the solver has no substantial effect on the heuristic. That we have found at least one problem where seeding has an obvious influence on the solving performance motivates us to explore further this branch of experiments running the same setup with benchmarks from other theory categories in the future. We do not know yet, why Z3 reacts in the way it does for different seeds on this single example. Based on the initial evaluation presented in the table, seeding a single solver multiple times and use it in a portfolio does not seem to be a good strategy for reducing computation time.

Nevertheless, if we just look at the percental gain in solving time reported in Figure 4 across all 131 experiments taken from the `QF_LIA` benchmark set, the result looks more promising. We can clearly identify examples, where computing the mean across 10 runs with different seeds beats the mean runtime of examples running with a single seed by more than 10 %.

Hence, extending the platform and experiment setup towards more solver and a broader theory support will be worth the effort for identifying slower and faster seeds for a problem. These seeds might be useful for evaluating the internal solver heuristics in the future. In the paper [HJM19], we observed more effects of seeding on the runtime of floating-point theory solver. Therefore, we will conduct more experiments in the future to investigate the reasons in detail. Independent of potential improvements for a single solver, the overall results are a promising indication that a solving service using a mixture of different constraint solvers and the same solver with different seeds might lead to substantial performance improvements for tools being slowed down by constraint solvers at the moment.

6 Conclusion

In this paper, we reported on general ideas for improving the integration of SMT solvers into programs and deduced requirements for a solving service. We proposed to collect more data about SMT solver profiling than the current SMT-Challenge does. We presented a platform for running large scale empirical experiments allowing to collect this additional data. In a preliminary evaluation of the platform, we collected some data for empirical testing Z3 behavior after seeding. The data indicates, it might be worth running the same solver with different seeds to decrease solving times.

Those preliminary results motivate us to continue running experiments for profiling performance characteristics of the heuristics used within SMT sovlers. We hope to be able to use those profiles for fine tuning SMT solver configurations. In the future, we aim for a full solving service exploiting the profiling data and optimization potential for speeding up SMT based analysis.

Bibliography

[APT79] B. Aspvall, M. F. Plass, R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters* 8(3):121–123, 1979.

[BCD+11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*. Pp. 171–177. 2011.

[Bey19] D. Beyer. Automatic verification of C and Java programs: SV-COMP 2019. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 133–155. 2019.

[BGZ17] M. Berzish, V. Ganesh, Y. Zheng. Z3str3: a string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. Pp. 55–59. 2017.

[BHM09] A. Biere, M. Heule, H. van Maaren. *Handbook of satisfiability*. Volume 185. IOS press, 2009.

[BLW19] D. Beyer, S. Löwe, P. Wendler. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer* 21(1):1–29, 2019.

[CDE+08] C. Cadar, D. Dunbar, D. R. Engler et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. Volume 8, pp. 209–224. 2008.

[CGSS13] A. Cimatti, A. Griggio, B. Schaafsma, R. Sebastiani. The MathSAT5 SMT Solver. In Piterman and Smolka (eds.), *Proceedings of TACAS*. LNCS 7795. Springer, 2013.

[CHN12] J. Christ, J. Hoenicke, A. Nutz. SMTInterpol: An interpolating SMT solver. In *International SPIN Workshop on Model Checking of Software*. Pp. 248–254. 2012.

[CKL04] E. Clarke, D. Kroening, F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 168–176. 2004.

[Cok11] D. R. Cok. jSMTLIB: Tutorial, validation and adapter tools for SMT-LIBv2. In *NASA Formal Methods Symposium*. Pp. 480–486. 2011.

[Coo18] B. Cook. Formal Reasoning About the Security of Amazon Web Services. In *International Conference on Computer Aided Verification*. Pp. 38–47. 2018.

[DA14] P. Dinges, G. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Pp. 425–436. 2014.

[DB08] L. De Moura, N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 337–340. 2008.

[DLL62] M. Davis, G. Logemann, D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM* 5(7):394–397, July 1962. doi:10.1145/368273.368557

[DP60] M. Davis, H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM* 7(3):201–215, July 1960. doi:10.1145/321033.321034

[Dut14] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*. Pp. 737–744. 2014.

[GKC13] S. Gao, S. Kong, E. M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *International conference on automated deduction*. Pp. 208–214. 2013.

[GKS05] P. Godefroid, N. Klarlund, K. Sen. DART: directed automated random testing. In *ACM Sigplan Notices*. Volume 40(6), pp. 213–223. 2005.

[GM15] M. Gario, A. Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*. 2015.

[God20] P. Godefroid. Fuzzing: hack, art, and science. *Communications of the ACM* 63(2):70–76, 2020.

[GRR12] D. Giannakopoulou, Z. Rakamarić, V. Raman. Symbolic Learning of Component Interfaces. In *Proceedings of the 19th International Conference on Static Analysis*. SAS'12, pp. 248–264. Springer-Verlag, Berlin, Heidelberg, 2012.

[HGMN18] F. Howar, D. Giannakopoulou, M. Mues, J. A. Navas. Generating component interfaces by integrating static and symbolic analysis, learning, and runtime monitoring. In *International Symposium on Leveraging Applications of Formal Methods*. Pp. 120–136. 2018.

[HJM19] F. Howar, F. Jabbour, M. Mues. JConstraints: a library for working with logic expressions in Java. In *Models, Mindsets, Meta: The What, the How, and the Why Not?* Pp. 310–325. Springer, 2019.

[KFB16] E. G. Karpenkov, K. Friedberger, D. Beyer. JavaSMT: A unified interface for SMT solvers in Java. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Pp. 139–148. 2016.

[KS16] D. Kroening, O. Strichman. *Decision procedures*. Springer, 2016.

[LDG+16] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, V. Raman. JDart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 442–459. 2016.

[Leh90] F. Lehner. Cost Comparison for the Development and Maintenance of Application Systems in 3rd and 4th Generation Languages. *Information & Management* 18(3):131–141, 1990.

[PC13] H. Palikareva, C. Cadar. Multi-solver support in symbolic execution. In *International Conference on Computer Aided Verification*. Pp. 53–68. 2013.

[PR10] C. S. Păsăreanu, N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. Pp. 179–180. 2010.

[RHF+17] H. Riener, F. Haedicke, S. Frehse, M. Soeken, D. Große, R. Drechsler, G. Fey. metaSMT: focus on your application and not on solver integration. *International Journal on Software Tools for Technology Transfer* 19(5):605–621, 2017.

[RLO16] K. Richly, M. Lorenz, S. Oergel. S4J-Integrating SQL into Java at Compiler-Level. In *International Conference on Information and Software Technologies*. Pp. 300–315. 2016.

[RT03] S. Ranise, C. Tinelli. The SMT-LIB format: An initial proposal. In *Proceedings of the 1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'03), Miami, Florida*. Pp. 94–111. 2003.

[Rüm08] P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Pp. 274–289. 2008.

[SZ09] D. Spiewak, T. Zhao. ScalaQL: language-integrated database queries for scala. In *International Conference on Software Language Engineering*. Pp. 154–163. 2009.

[VGD12] W. Visser, J. Geldenhuys, M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. P. 58. 2012.

[VP95] M. Venkatrao, M. Pizzo. SQL/CLI—a new binding style for SQL. *ACM SIGMOD Record* 24(4):72–77, 1995.

[WB19] P. Wendler, D. Beyer. sosy-lab/benchexec: Release 2.0. July 2019.
doi:10.5281/zenodo.3341460
https://doi.org/10.5281/zenodo.3341460