Smt.ml: A Multi-Backend Frontend for SMT Solvers in OCaml

João Madeira Pereira, Filipe Marques, Pedro Adão, Hichem Rami Ait El Hara, Léo Andrès, Arthur Carcano, Pierre Chambart, Nuno Santos, José Fragoso Santos

Smt.ml: A Multi-Backend Frontend for SMT Solvers in OCaml

João Madeira Pereira[1], Filipe Marques[1], Pedro Adão[1],
Hichem Rami Ait El Hara[2,3], Léo Andrès[2], Arthur Carcano[2],
Pierre Chambart[2], Nuno Santos[1], and José Fragoso Santos[1]

[1] University of Lisbon, Lisbon, Portugal
[2] OCamlPro, Paris, France
[3] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract. SMT solvers are essential for applications in artificial intelligence, software verification, and optimisation. However, no single solver excels across all formula types, different applications may require the use of different solvers. While the SMT-LIB language enables multi-solver support, it incurs heavy I/O overhead. To address this, we introduce Smt.ml, an SMT solver frontend for OCaml that simplifies integration with various solvers through a consistent interface. Its parametric encoding facilitates the easy addition of new solver backends, while optimisations like formula simplification, result caching, and detailed error feedback enhance performance and usability. Our evaluation demonstrates that Smt.ml maintains correctness and performance comparable to individual solvers, with significant improvements in batched interactions.

Keywords: SMT Solvers · Symbolic Execution · OCaml · SMT-LIB

1 Introduction

Since their emergence in the early 2000s, SMT solvers have become increasingly relevant and are now fundamental to numerous applications in modern life. They are applied in various scientific and industrial domains, ranging from planning problems in artificial intelligence [15] to software verification and test generation in software engineering [6, 16, 27, 33], and even the optimisation of production chains in operations research [10].

While there are now multiple industry-strength SMT solvers to choose from, each has its own strengths and weaknesses, with no single solver excelling at all types of formulas. Recent SMT-COMP [13] results reveal, for instance, that cvc5 [5] achieved the best results in the category for bitvector formulas, while Bitwuzla [40] was the best solver in the category of floating-point arithmetic formulas. This diversity of results indicates that there is no perfect solver for all applications, and even in the context of the same application, it may be beneficial to use different solvers for different formulas.

One strategy to support multiple solvers is to use the SMT-LIB language [8], a solver-agnostic textual format that is supported by multiple solvers. This approach involves serializing the given formula into an SMT-LIB formula and executing the most suitable solver on the generated file. However, this strategy is not effective for applications that require solving a large number of formulas as solver interaction is mediated through the file system, incurring heavy I/O overhead. Therefore, using the SMT-LIB language does not adequately resolve the challenge of supporting multiple solvers in performance-critical applications.

When performance is critical, solvers should be integrated into the client application's code base as external libraries. To this end, each solver provides an API that other applications can use to invoke its supported functionalities. Unfortunately, these APIs are frequently inconsistent, buggy, and challenging to use. Additionally, in strongly-typed functional programming languages such as OCaml [35], Haskell [30], and Scala [42]–which are common choices for applications involving SMT solvers–there is frequently the added challenge of no available API for various solvers.

To address these challenges, we introduce Smt.ml, a new SMT solver frontend for OCaml. Smt.ml simplifies the integration of OCaml programs with multiple SMT solvers. It achieves this by providing a consistent SMT-LIB-compatible language linked to four different solver backends: Bitwuzla [40], Colibri2 [11], cvc5 [5], and Z3 [22]. With Smt.ml, OCaml developers are not required to understand the intricate details of these solvers' APIs to use them; they simply create an Smt.ml formula and select the desired solver backend. As part of the Smt.ml development effort, we created new OCaml APIs for two SMT solvers: Colibri2 [11] and cvc5 [5].

At the core of Smt.ml is a new parametric encoding that streamlines the addition of new solver backends. More concretely, instead of writing an encoding for each backend separately, we defined an interface that all backends must support and used this interface to translate Smt.ml formulas into the logics of the different backends. In addition to avoiding redundant code, this approach greatly simplifies the addition of new backends, which only need to implement our newly defined interface. Beyond supporting the usage of various SMT solvers within OCaml, Smt.ml presents additional benefits over using existing solver APIs directly: **(1)** it includes several simplifications that reduce formula complexity, boosting solver performance; **(2)** it caches satisfiability results to avoid redundant computations; and **(3)** it provides informative error feedback, streamlining the debugging process.

Smt.ml is integrated into OPAM [51], the OCaml package manager, simplifying its use for OCaml programmers. It is actively used in various research projects both in academia [38] and industry [3]. Our evaluation demonstrates that **(1)** the results produced by Smt.ml are consistent with those obtained by directly using the supported solvers, i.e., Smt.ml does not introduce any bugs; and **(2)** Smt.ml introduces no performance degradation with respect to the direct use of the supported solvers and shows significant performance improvements for batched solver interactions.

The remainder of this paper is structured as follows: Section 2 provides an overview of Smt.ml, discusses its benefits, and introduces the supported for-

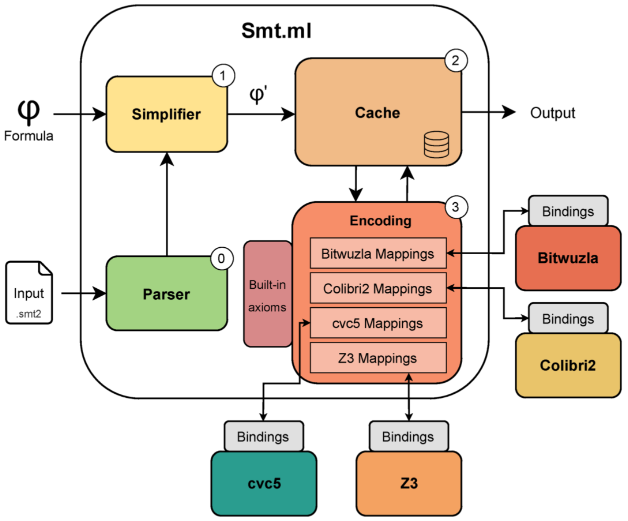

Fig. 1: Overview of Smt.ml's architecture.

mula grammar; Section 3 describes our parametric translation from Smt.ml to the formulas of each backend; Section 4 presents the simplification and caching mechanisms featured in Smt.ml; Section 5 evaluates Smt.ml's performance and correctness, using the SMT-LIB benchmarks [45]; Section 6 presents a case study of Smt.ml in symbolic execution; Section 7 discusses related work; and Finally, Section 8 concludes the paper and discusses future work.

2 Architecture

Figure 1 presents a high-level overview of Smt.ml's architecture. While one can use Smt.ml as a standalone tool, Smt.ml was primarily conceived to be used as a library within an OCaml application. Hence, Smt.ml accepts both types of inputs, native Smt.ml's formulas and SMT-LIB [8] textual formulas. In the latter case, the formula is first parsed by the *Parser Module (0)* and converted into an Smt.ml's native formula.

Given an Smt.ml's native formula, Smt.ml performs the following steps:

- *Simplifier Module Step (1):* This module applies a range of transformations to the given formula to reduce its complexity while preserving its original semantics. For instance, it applies the following algebraic identity to simplify bitvector formulas:

$$\texttt{concat}(\texttt{extract}(x,h,m),\ \texttt{extract}(x,m,l)) \rightarrow \texttt{extract}(x,h,l)$$

- *Cache Module Step (2):* This module normalises the given formula, checks if its satisfiability was already computed, and if so, returns the stored result. The normalisation process includes operations like renaming of variables, allowing cache hits to be maximised.
- *Encoding Module Step (3):* This module is responsible for encoding the formula using the native OCaml bindings of the selected solver. It is parametric on a generic solver interface facilitating its extension with support for new solver backends.

2.1 Why use Smt.ml?

In this section, we discuss some of the primary advantages Smt.ml introduces for developers working with SMT solvers in OCaml.

Solver Independence A key advantage of Smt.ml is the ability to interface with multiple SMT solvers through a single solver-independent frontend. Traditionally, developers must tailor their code to the specific syntax and API of each solver. Smt.ml eliminates this constraint, allowing developers to transparently switch between solvers without the need to rewrite their entire program. By abstracting the differences between solver APIs, Smt.ml enables the selection of the most appropriate solver for a given problem. This decision can even be made at runtime, allowing for the application of customised portfolio strategies [49].

Performance Optimisations Interactions with SMT solvers are computationally expensive and can become a bottleneck in client applications. For instance, these interactions are known to be one of the main performance degradation factors in symbolic execution tools. To counter this issue, Smt.ml applies two complementary strategies for minimising solver interactions and optimise performance: formula simplifications and caching of satisfiability results. Formula simplifications reduce the complexity of formulas, often even making them trivially true/false and eliminating the need to query the solver at all. Caching satisfiability results improves performance by avoiding redundant solver queries for formulas that have already been checked. Smt.ml offers these two types of solver-agnostic performance optimisations so that developers do not have the burden of implementing them.

Usability OCaml bindings for SMT solvers often provide few type safety guarantees, with many using the same generic OCaml type to represent SMT expressions denoting different types of values, such as integers, strings, or booleans. For instance, Z3's OCaml bindings use a single type for all general expressions, regardless of their underlying sort. This approach results in ill-typed expressions not being detected at compile time leading to runtime errors that are typically hard to debug. Listing 1 illustrates this issue with a program that uses Z3's bindings. The given program is able to apply an uninterpreted function that

Listing 1 Type violation of function declaration using Z3 OCaml bindings.

```
 1 (* Function Declaration: foo : int -> int *)
 2 let foo = FuncDecl.mk_func_decl_s ctx "foo" [ int_sort ] int_sort in
 3
 4 let str_sort = Seq.mk_string_sort ctx in
 5 let value = Symbol.mk_string ctx "value" in
 6 let str_const = Expr.mk_const ctx value str_sort in
 7
 8 let foo_app = Expr.mk_app ctx foo [ str_const ] in
 9 (* foo(value) + 2 >= 2 *)
10 let formula = Arithmetic.mk_ge ctx
11     (Arithmetic.mk_add ctx [ foo_app ; two ]) two
```

expects an integer argument (line 7) to a string constant (line 13), resulting in a runtime error stating that there is a mismatch between the expected and actual argument types. In contrast, Smt.ml's expressions are well-typed by construction avoiding this type of bug. This promotes code correctness and reliability in the development process.

2.2 Smt.ml Syntax

Figure 2 presents the syntax of Smt.ml. There are two main syntactic categories: expressions, that denote values, and commands, that represent instructions given to the solver.

On the whole, Smt.ml supports the quantifier-free linear integer and real arithmetic (QF_LIA and QF_LRA), bitvectors (QF_BV), floating-point arithmetic (QF_FP), and strings (QF_S) theories. These are the theories most commonly required for software verification and analysis tasks [4, 17], which are now the current main applications of Smt.ml. However, Smt.ml has a modular and extensible architecture, making it easy to add support for new theories.

Expressions Expressions in Smt.ml are built using values and operators. Values, v, include boolean constants (`true` and `false`), integers (`int`), reals (`real`), strings (`str`), numeral constants (`num`), and lists of values. Numerals are machine integers (8, 32 and 64-bit) or IEEE 754 floating-point numbers [31] (32 and 64-bit). Expressions, e, are constructed by combining values with *symbolic variables* of the form x_t, where x denotes the variable and t its type. Expressions are constructed through the application of a variety of operators. Smt.ml includes unary (`unop`(`op`, t, e)), binary (`binop`(`op`, t, e, e)), ternary (`triop`(`op`, t, e, e, e)), and n-ary (`naryop`(`op`, t, `list` e)) operators. Additionally, it includes relational operators (`relop`(`op`, t, e, e)) for comparisons and conversion operators (`cvtop`(`op`, t, e)) for type casting or conversion between value types. In the following, we refer to expressions of type Boolean as *formulas*.

Commands Smt.ml provides a set of commands to manipulate and interact with the SMT solvers. Commands, c, include typical solver instructions such as `assert` e, that asserts a given formula to the solver, and `check_sat` (`list` e),

Numeral Values

$num ::= i8 \mid i32 \mid i64 \mid f32 \mid f64$

Types

$$\begin{aligned} t ::=\; & \texttt{Ty\_int} \mid \texttt{Ty\_str} \mid \texttt{Ty\_bitv}\ int \mid \texttt{Ty\_bool} \mid \texttt{Ty\_fp}\ int \mid \texttt{Ty\_real} \\ & \mid \texttt{Ty\_unit} \mid \texttt{Ty\_list} \mid \texttt{Ty\_app} \end{aligned}$$

Values

$v \in \mathcal{V}_{smt} ::= \texttt{true} \mid \texttt{false} \mid \texttt{unit} \mid \texttt{int} \mid \texttt{real} \mid \texttt{str} \mid \texttt{num} \mid \texttt{list}\ v$

Formulas

$$\begin{aligned} e \in \mathcal{E}_{smt} ::=\; & v \mid x_t \mid \texttt{unop(op},\ t,\ e) \mid \texttt{binop(op},\ t,\ e,\ e) \\ & \mid \texttt{triop(op},\ t,\ e,\ e,\ e) \mid \texttt{relop(op},\ t,\ e,\ e) \mid \texttt{cvtop(op},\ t,\ e) \\ & \mid \texttt{naryop(op},\ t,\ \texttt{list}\ e) \mid \texttt{list}\ e \end{aligned}$$

Commands

$$\begin{aligned} c \in \mathcal{C}_{smt} ::=\; & \texttt{assert}\ e \mid \texttt{check\_sat}\ (\texttt{list}\ e) \mid \texttt{declare}(x_t) \mid \texttt{exit} \\ & \mid \texttt{get\_model} \mid \texttt{get\_value}\ e \mid \texttt{pop}\ int \mid \texttt{push}\ int \mid \texttt{reset} \end{aligned}$$

Fig. 2: Smt.ml's syntax.

that checks the satisfiability of a list of formulas. The command `declare`(x_t) is used to declare a new symbolic variable x of type t. Additional commands include `exit`, used to terminate the interaction with the solver, `get_model`, to retrieve a model when given a list of satisfiable formulas, and `get_value` e, that returns a satisfiable value for the given expression. For state manipulation, Smt.ml provides the `push` *int* and `pop` *int*, commands to introduce or remove a number of assertion levels, respectively. Finally the `reset` command is used to reset the solver's state allowing for a fresh set of assertions and commands to be processed.

3 Encoding

The Encoding module is the core component of Smt.ml. It is responsible for translating Smt.ml's native expressions and commands into the expressions and commands of each solver backend. Instead of having an encoding module for each backend, Smt.ml's encoding module is parametric on a *Core Solver API*, $\mathcal{S}$, that solvers are expected to implement, streamlining the addition of new solver backends and avoiding code duplication. Having established this API, obtaining the encoding of Smt.ml's logic into the logic of a specific solver is straightforward: it suffices to plug the target solver's implementation of the API $\mathcal{S}$ into Smt.ml.

Naturally, we do not expect solver developers to implement our Core Solver API. Instead, we build ourselves the wrappers around those solvers that do implement the expected API and plug the obtained wrapped solvers into Smt.ml. Figure 3 illustrates this process, showing how different SMT solvers, each with its

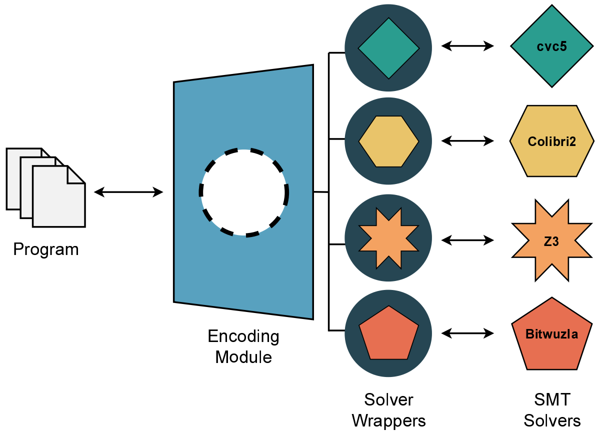

Fig. 3: SMT.ML's parametric Solver module.

own specific API, can be integrated into SMT.ML by first being wrapped within modules that implement the expected API.

Solver-Independent API The Core Solver API $\mathcal{S}$ lists all functions on solver values, expressions, and commands on which the translation of SMT.ML expressions and commands depends. These functions can be divided into the following four main categories:

- *Values:* The functions in this category are responsible for mapping SMT.ML primitive values, $v \in \mathcal{V}_{smt}$, into values from the corresponding target solver.
- *Operators:* These functions map SMT.ML operators to the corresponding solver operators. For example, solver wrappers are expected to implement the addition operator, which, when given two solver expressions, returns a target solver expression representing their sum.
- *Commands:* The funtions in this category map SMT.ML commands, $c \in \mathcal{C}_{smt}$, into commands that manipulate and interact with the target solver.
- *Lifting:* This category contains functions that map solver values back to SMT.ML values, $v \in \mathcal{V}_{smt}$. Such functions are essential when performing model extraction tasks, for instance, as they allow models to be constructed using SMT.ML's native values.

Parametric Translation Using the interface described above, we implement a generic translation for converting SMT.ML constructs into the corresponding solver-specific constructs. We formalise the main translation for expressions as a function $T_{\mathcal{S}} : \mathcal{E}_{smt} \to \mathcal{S}.Expr$ that receives an SMT.ML expression $e \in \mathcal{E}_{smt}$ and generates a target solver expression $te \in \mathcal{S}.Expr$. Figure 4 presents some of the parametric translation rules of SMT.ML.

$$\begin{array}{l}\textsc{Values}\\ \dfrac{\mathcal{S}.val(v) = v'}{T_{\mathcal{S}}(v) = v'}\end{array} \qquad \begin{array}{l}\textsc{Symbols}\\ \dfrac{\mathcal{S}.symbol(s) = s'}{T_{\mathcal{S}}(s) = s'}\end{array} \qquad \begin{array}{l}\textsc{Unary operators}\\ \dfrac{T_{\mathcal{S}}(e) = te \qquad \mathcal{S}.unop(uop, te) = e'}{T_{\mathcal{S}}(uop\ e) = e'}\end{array}$$

$$\begin{array}{l}\textsc{n-ary operators}\\ \dfrac{T_{\mathcal{S}}(e_i) = e'_i\ |_{i=1}^{n} \qquad \mathcal{S}.naryop(nop, [e'_1, \ldots,\ e'_n]) = e'}{T_{\mathcal{S}}(nop\ [e_1, \ldots,\ e_n]) = e'}\end{array}$$

Fig. 4: Parametric translation rules for Smt.ml.

Values v, and symbols s, are translated into their counterparts in the target solver $\mathcal{S}$, using the functions $\mathcal{S}.val(v)$ and $\mathcal{S}.symbol(s)$, respectively. Unary operators $uop\ e$ are translated into the application of the solver's unary operator to the translation of the operation argument $T_{\mathcal{S}}(e) = te$, which is computed using the Core Solver API function $\mathcal{S}.unop$, which receives as inputs a unary operator uop and a solver expression te and generates the solver expression that denotes the application of uop to te. The translation proceeds similarly for binary, ternary, n-ary, and relational operators.

Currently, Smt.ml supports four backends solvers: Bitwuzla, Colibri2, cvc5, and Z3. As part of the development of Smt.ml, we implemented the Core Solver API wrapper for each of them. Furthermore, for cvc5 we had to implement its entire OCaml bindings from scratch, as cvc5 came with no bindings for OCaml. These bindings bridge the gap between OCaml and the solver's native C++ API, allowing it to be integrated into Smt.ml.

4 Backend-independent Optimisations

Smt.ml's usefulness extends beyond its capability to interact with multiple SMT solvers through a unified syntax. A significant aspect of its design is the inclusion of backend-independent optimisations that enhance performance when checking the satisfiability of logical formulas. In this section, we discuss two key optimisations implemented in Smt.ml: expression simplications and caching.

4.1 Expression Simplifications

In applications that interact with SMT solvers, the solver's performance often becomes the primary bottleneck, with the problem's size and complexity significantly affecting the solver's efficiency [53]. To counter this, Smt.ml features a set of simplifications that are applied to expressions in an attempt to reduce their overall complexity, while maintaining their original meaning.

At the core of our expression-simplification algorithm is a set of simplification rules $r \in \mathcal{R}$ of the form: $f\ ?\ e_1 \rightarrow e_2$, meaning that if the formula f holds the expression e_1 can be rewritten as e_2. For instance, the rule

$$r_1 \equiv h - l = |x|\ ?\ \texttt{extract}(x, h, l) \rightarrow x$$

Algorithm 1 Expression simplification algorithm.

```
procedure Simplify($f$)
    $f' \leftarrow f$
    for all $r \in$ Rules do
        $f' \leftarrow$ apply $r$ to $f'$
    if $f' \neq f$ then
        return Simplify($f'$)
    else
        return $f'$
```

states that the expression $\texttt{extract}(x, h, l)$ can be rewritten as x if $h - l$ coincides with the size of x. In order to apply a simplification rule f ? $e_1 \rightarrow e_2$ to a given expression e under an execution context where the formula f' holds, we proceeds as follows:

1. find a substitution θ such that $\theta(e_1) = e$;
2. check if $f' \implies \theta(f)$;
3. replace e with $\theta(e_2)$.

For instance, applying r_1 to $\texttt{extract}(\texttt{concat}(y, z), |y|{+}2, 0)$ yields the expression $\texttt{concat}(y, z)$ with substitution $\theta = [x \mapsto \texttt{concat}(y, z), l \mapsto 0, h \mapsto |y| + 2]$ under the formula context $f' \equiv |z| = 2$.

Currently, Smt.ml performs constant folding for every theory (215 rules) and comes with 62 additional simplification rules, spanning the theories of bit-vectors (28 rules), floating point arithmetic (3 rules), boolean (2 rules), strings (2 rules), and 27 generic rules which can be applied to multiple theories. The rules must be carefully designed to ensure that the simplifications are sound (i.e., the simplified expression denotes the same values as the original one) and to make sure that the simplification process is not infinite, meaning that rule application does not generate simplification loops, such as $e \xrightarrow{r_1} e' \xrightarrow{r_2} e$, where e is the original expression, e' is the simplified expression, and r_1 and r_2 are two simplification rules.

With a set of sound and terminating simplification rules established, designing our expression-simplification procedure becomes straightforward. Algorithm 1 outlines this procedure: the main loop iterates through the set of simplification rules, applying those whose constraints are satisfied. The loop terminates once no further rules can be applied to the expression at hand, indicating that a fixed point has been reached.

4.2 Caching

Caching and Normalisation Caching of intermediate satisfiability results is a standard technique used in SMT solvers and solver clients to improve performance [43, 46]. However, it is not common for identical formulas to be queried multiple times, even in applications that make an intensive use of SMT solvers. To address this, formula caching systems [52] typically implement normalisation

Listing 2 Hash-consing constructor for boolean disjunction.

```
1 let table = Hashtbl.create 251
2 let mk_or hte1 hte2 =
3   let x = Binary (Or, hte1, hte2) in
4   try Hashtbl.find table x
5   with Not_found -> Hashtbl.add table x x; x
```

strategies with the goal of maximising cache hits. Smt.ml comes with its own formula caching system equipped with a normalisation procedure that performs:

- *Standardisation of associative operators:* a standard order is imposed on expressions that include such operators. For instance, considering the logical *or* operator (commonly denoted by $\vee$), we have that $(x \vee y) \vee z = x \vee (y \vee z)$. In Smt.ml, expressions that include chained associative operators are always rewritten to ensure that the leftmost operations are performed first.
- *Variable renaming:* variables are renamed to ensure structurally identical formulas with different variable names are considered equal.

Caching Expressions via Hash-consing In addition to minimizing the number of queries, another way to enhance the performance of solver clients is to reduce the number of formulas and expressions created at runtime. In fact, solver clients often generate a large number of formulas, frequently with repeated elements. As the number of queries grows, memory consumption increases, significantly impacting client's performance; a prime example of this is symbolic executors [18]. The standard technique to reduce the memory impact of solver systems is the use of *hash-consing* [26], a technique that ensures that no two physical copies of the same expression are ever created by storing expressions in a hash table. Smt.ml includes a hash-consing module that prevents the duplication of identical expressions. To this end, whenever an Smt.ml expression constructor is called, it checks whether the expression already exists, and, if it does, returns the previously stored expression.

Listing 2 illustrates this process for the `Or` constructor. We define the `mk_or` hash-consing constructor, which builds a boolean disjunction between two hash-consed expressions. In line 3, we construct the binary expression, and in line 4, we attempt to retrieve a previously constructed expression from the hash-consing table. If the expression is not found (line 5), we add it to the table and return the value constructed in line 3. Smt.ml only allows creating expressions through these smart constructors, ensuring that every expression is correctly hash-consed.

The more attentive reader might notice the extra heap allocation in line 3 when there already exists an hash-consed expression. Specifically, we allocate memory to construct an expression, only to later use an existing one retrieved from the hash-consing table. However, because OCaml initially allocates values in the minor heap using a bump allocator [39], this allocation incurs no cost. Additionally, OCaml can quickly collect the temporarily allocated values during minor garbage collection.

5 Evaluation

We evaluate Smt.ml with respect to three evaluation questions:

- **EQ1:** Does Smt.ml exhibit consistent behaviour with the supported solvers?
- **EQ2:** Does Smt.ml exhibit consistent performance with the supported solvers?
- **EQ3:** Does Smt.ml improve the performance of the supported solvers for batched interactions?

5.1 Experimental Setup

To answer our research questions, we leverage the SMT-LIB benchmarks dataset [45], more concretely, we focus on the benchmarks that cover the following theories

- Quantifier-Free Linear Integer Arithmetic (`QF_LIA`);
- Quantifier-Free Floating-Point Arithmetic (`QF_FP`);
- Quantifier-Free Bitvector Arithmetic (`QF_BV`);
- Quantifier-Free String Theory (`QF_S`);
- Quantifier-Free String Theory and Linear Integer Arithmetic (`QF_SLIA`)

We chose these benchmarks as they cover the main theories supported by Smt.ml solver backends, and because they are widely used in the SMT community.

All experiments were performed on a server with a 12-core Intel Xeon E5–2620 CPU and 32GB of RAM running Ubuntu 24.04.1 LTS. For compiling Smt.ml, we used the OCaml 5.2.0 compiler. For the SMT solvers, we employed Bitwuzla version 0.5.0, the development version of Colibri2 pinned to commit `1feba887`, cvc5 version 1.2.0, and Z3 version 4.13.0.

The benchmarking code, reproducibility scripts, and diagram generation scripts are all available in Smt.ml's GitHub repository.[4]

5.2 EQ1: Correctness

To assess the correctness of Smt.ml, we conducted a comparative evaluation using the selected SMT-LIB benchmarks. Specifically, we compared the results obtained when running Smt.ml on these benchmarks with the results produced by executing each supported solver directly on the same datasets. Note that, each benchmark in the SMT-LIB benchmark dataset is annotated with its expected outcome, allowing us to cross-validate the results obtained from both Smt.ml and the native solvers.

[4] https://github.com/formalsec/smtml

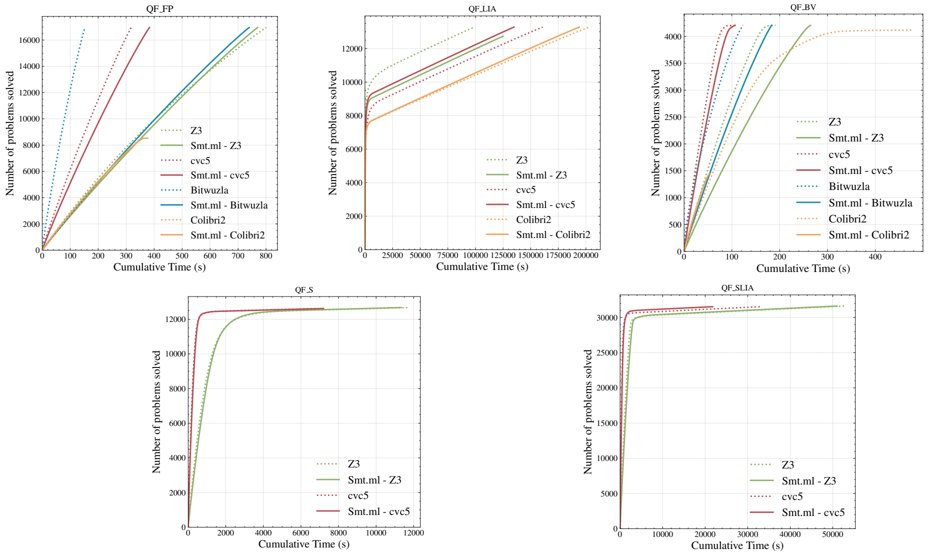

Fig. 5: Accumulated single-query results by theory.

Results. Upon comparison, we observed that all results produced by Smt.ml were consistent with those produced by the solvers directly and aligned with the expected results for each benchmark. This demonstrates that Smt.ml consistently reproduces the behaviour of the underlying solvers without introducing any correctness bugs.

Takeaway 1: Smt.ml does not introduce any inconsistent behaviour when compared to its backend solvers.

5.3 EQ2: Single-Query Performance

To investigate whether the Smt.ml negatively impacts the performance of the backend solvers, we execute Smt.ml on the selected dataset a single query at a time for each solver, creating a new solver instance per query, and comparing the runtimes obtained for each query with the runtimes obtained by running the corresponding solvers directly on the given query.

Results. Figure 5 shows the Cumulative Distribution Function (CDF) plots for single-query results with each sub-figure showing the results for a specific theory. From the results, we conclude the following:

- In general, the performance of Smt.ml is closely aligned with the performance of the backend solvers when used independently.
- For the Bitwuzla solver, a slight performance degradation is observed when comparing the results of Smt.ml-Bitwuzla to Bitwuzla alone. This is pri-

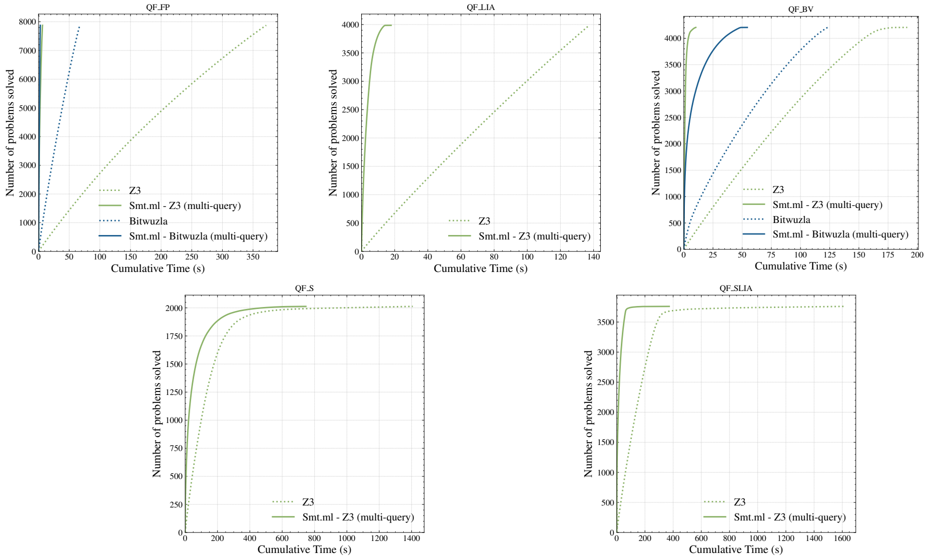

Fig. 6: Accumulated multi-query results by theory.

marily due to the overhead associated with bookkeeping created constants in logical formulas – something Bitwuzla's API does not perform.

- In certain cases, such as with the `QF_SLIA` theory, Smt.ml outperforms the backend solvers, solving more problem instances in less time. This performance gain highlights the effectiveness of the formula simplifications Smt.ml applies before querying the solvers.

Takeaway 2: Smt.ml does not degrade solver performance in single-query interactions, and even improves performance for some theories.

5.4 EQ3: Multi-Query Performance

Smt.ml's main use case lies in applications that require frequent interactions with SMT solvers, such as symbolic execution engines. To evaluate Smt.ml's performance in such scenarios, we consider a multi-query setting where interactions are batched and executed using a single solver instance. We compare the runtimes of Smt.ml in batch mode against the accumulated runtimes obtained by running directly each solver one query at the time.

For these experiments, we did not consider the cvc5 or Colibri2 solvers, as their wrappers do not yet support Smt.ml's multi-query mode. In the case of cvc5, we were unable to run the multi-query mode due to a garbage collection issue, where solver objects are not properly reference counted and are, therefore, prematurely collected. This issue is currently being addressed by the cvc5 development team. For Colibri2, a bug was detected during evaluation that crashes the solver after various push and pop operations, preventing it from being run

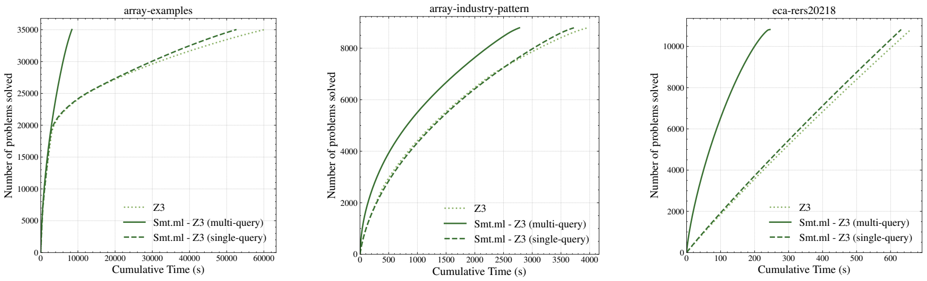

Fig. 7: SMT.ML performance on the selected categories of formulas from the Test-Comp 2023 benchmark suite.

in multi-query mode. We are currently investigating this issue in collaboration with the Colibri2 development team.

Results. Figure 6 presents the CDF plots for the multi-query results with each sub-figure showing the results for a specific theory. From the results, we conclude that SMT.ML's multi-query mode substantially outperforms the base solvers when used in single-query mode for all theories. This is expected given that, when executing base solvers independently, a new solver instance needs to be created for each query, incurring in significant overhead due to the solver's initialisation and setup. In SMT.ML's multi-query mode, the solver instance is reused during the batched interaction, avoiding this overhead and leading to substantial performance improvements. Furthermore, SMT.ML's caching mechanisms contribute to this performance boost by reusing previously computed satisfiability results.

Takeaway 3: SMT.ML significantly improves performance in multi-query solver interactions.

6 Case Study

To demonstrate the practical utility of SMT.ML, we describe its application in the development of Owi [3], a novel symbolic execution tool for WebAssembly (Wasm). Owi was originally conceived as a concrete interpreter for WebAssembly, which we later parameterized on a generic value interface, allowing it to handle both concrete and symbolic values. By using SMT.ML's expressions as Owi's values, we obtained a new symbolic interpreter for Wasm tightly connected to the formal semantics of the language. The details of this parameterization can be found in [3].

To evaluate the impact of SMT.ML within the Owi framework, we executed Owi on three distinct categories of formulas from the Test-Comp 2023 benchmark suite [9], a well recognized standard for evaluating symbolic execution tools. Throughout the execution process, all generated formulas were serialized for further analysis. We selected the categories `array-examples`, array-industry-pattern, and eca-rers2018, as they offer a diverse representation of bitvector (QF_BV) formulas commonly encountered in symbolic execution contexts. Subsequently, we used Smt.ml with the Z3 backend to solve the serialized formulas and compared its performance against executing the Z3 solver directly on each formula.

Results. Figure 7 illustrates the CDF plots for the three selected categories from the Test-Comp 2023 benchmark suite. The results indicate that using Smt.ml's multi-query mode with Z3 as the backend significantly improves performance compared to both running Z3 directly on each formula and using Smt.ml in single-query mode.

Discussion. Currently, several other ongoing research projects use Smt.ml, including a new symbolic executor for JavaScript [20] and a new tool for exploit generation for Node.js packages, which was built on top of a state-of-the-art static analyser for vulnerability detection in JavaScript [25]. Numerous applications, particularly in the fields of systems construction and analysis, could benefit from adopting Smt.ml. It is open-source and available in Smt.ml's GitHub repository.[5] Additionally, Smt.ml is also available as an OPAM package,[6] greatly simplifying its adoption and integration by the OCaml community.

7 Related Work

SMT Solvers SMT solvers have seen significant advancements since their emergence in the early 2000s [5, 7, 21–23], such as support for new theories, like strings [36] and quantified arithmetic [1], and new optimisation techniques, like new caching mechanisms [52] and portfolio strategies [44, 48]. As a result of this, they are now used across the entire computer science community, with a wide variety of applications ranging from software verification and test generation [28, 32, 34] to combinatorial optimisation and classical operations research [10].

Currently, many SMT solvers are actively maintained, with 20 submitted to the 2024 edition of SMT-COMP [13]. Importantly, there is no one-size-fits-all solver; some solvers excel in certain theories, while others excel in others. Our selection of SMT solvers for integration within Smt.ml was driven by our specific needs. Initially, we supported the Z3 [22] and Colibri2 [11] solvers, as they were already being used in our ongoing projects. Subsequently, we added support for cvc5 [5] and Bitwuzla [40] due to their excellent performance in the theories of bitvectors and floating-point arithmetic, which are frequently required in symbolic execution contexts. In the future, we plan to incorporate support for additional solvers.

[5] https://github.com/formalsec/smtml

[6] https://opam.ocaml.org/packages/smtml/

Frontends for SMT Solvers Frontends play an essential role in making SMT solvers more accessible to the wider computer science audience. These interfaces often come with user-friendly input languages that are both more expressive and closer to real-world problem domains than the logics of existing solvers, streamlining user interaction. Frontends also facilitate integration with high-level programming languages, allowing SMT-solving capabilities to be seamlessly embedded into applications and formal verification processes.

PySMT [29] and Smt-Switch [37] are examples of frontends for solvers in Python and C++, respectively. They equip users with a high-level API for interacting with various SMT solvers, abstracting the low-level solver-specific details. Both frontends support five SMT solvers:

- PySMT: Z3, cvc5, Yices2 [23], Bitwuzla, and MathSAT [14];
- Smt-Switch: Z3, cvc5, Yices2, MathSAT, and Boolector [41].

PySMT further implements a solver-agnostic optimization layer that simplifies the given formulas by applying a set of simplification rules similar to ours. These are the two tools closest in spirit to Smt.ml; however, unlike them, Smt.ml has the additional advantage of having an integrated caching system, which is essential for containing memory consumption in our target applications. Furthermore, Smt.ml is the first such frontend for the OCaml programming language.

Solver-aided languages, such as Rosette [50] and Why3 [27], offer a flexible approach to writing programs that interact with SMT solvers to reason about logical formulas in multiple first-order theories. Rosette extends the Racket programming language [24] with a symbolic compiler that translates solver-aided programs into logical constraints, enabling seamless interaction with SMT solvers for tasks like program synthesis [2, 12] and test generation [47]. Similarly, Why3, an OCaml-based platform, supports formal program verification by translating high-level program specifications into verification conditions that various SMT solvers can process.

8 Conclusions

We presented Smt.ml, a novel OCaml frontend for multiple SMT solvers. In contrast to existing frontends for SMT solvers in other languages, Smt.ml incorporates a solver-agnostic caching system and a set of formula simplifications that boost its performance, even in single-query interactions. We further demonstrate that, when used in multi-query mode, Smt.ml substantially outperforms the direct use of native SMT solvers.

In the future, we plan to extend this work in various ways:

- We will support new solver backends. Currently, there is already an ongoing effort to integrate the Alt-Ergo solver [19] into our infrastructure.
- We will improve the solver wrappers for cvc5 and Colibri2, enabling them to support batched interactions.
- We will offer users the possibility to create solver portfolios parameterized on solver selection strategies.

References

1. Ábrahám, E., Kremer, G.: SMT solving for arithmetic theories: Theory and tool support. In: 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 1–8. IEEE (2017)
2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. IEEE (2013)
3. Andrès, L., Marques, F., Carcano, A., Chambart, P., Fragoso Femenin dos Santos, J., Filliâtre, J.C.: Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. The Art, Science, and Engineering of Programming **9**(2) (Oct 2024), https://hal.science/hal-04627413
4. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)
5. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
6. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4. pp. 364–387. Springer (2006)
7. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. pp. 171–177. Springer (2011)
8. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). vol. 13, p. 14 (2010)
9. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. Fundamental Approaches to Software Engineering LNCS 13991 p. 309 (2023)
10. Bjørner, N., Levatich, M., Lopes, N.P., Rybalchenko, A., Vuppalapati, C.: Supercharging Plant Configurations Using Z3. In: Proc. of the 18th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR) (Jul 2021). https://doi.org/10.1007/978-3-030-78230-6_1
11. Bobot, F., Marre, B., Bury, G., Graham-Lengrand, S., Ait El Hara, H.R.: Colibri2. webpage: https://colibri.frama-c.com, source code: https://git.frama-c.com/pub/colibrics
12. Bodík, R., Jobstmann, B.: Algorithmic program synthesis: introduction. International journal on software tools for technology transfer **15**, 397–411 (2013)
13. Bromberger, M., Bobot, F., , et al.: The International Satisfiability Modulo Theories Competition (SMT-COMP)(2024). https://smt-comp.github.io/
14. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver: Tool paper. In: Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20. pp. 299–303. Springer (2008)
15. Bryce, D., Gao, S., Musliner, D., Goldman, R.: SMT-based nonlinear PDDL+ planning. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 29 (2015)

16. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
17. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM **56**(2), 82–90 (2013)
18. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on Binary Code. In: 2012 IEEE Symposium on Security and Privacy. pp. 380–394 (2012). https://doi.org/10.1109/SP.2012.31
19. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018)
20. Costa, M.: Memory Models for Symbolic Execution of JavaScript Applications. Master's thesis, Instituto Superior Técnico, University of Lisboa, Portugal (2023)
21. Damm, W., Hermanns, H.: Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, vol. 4590. Springer (2007)
22. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
23. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
24. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: The racket manifesto. In: 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
25. Ferreira, M., Monteiro, M., Brito, T., Coimbra, M.E., Santos, N., Jia, L., Santos, J.F.: Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs. Proceedings of the ACM on Programming Languages **8**(PLDI), 417–441 (2024)
26. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 Workshop on ML. p. 12–19. ML '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1159876.1159880, https://doi.org/10.1145/1159876.1159880
27. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22. pp. 125–128. Springer (2013)
28. Fragoso Santos, J., Maksimović, P., Ayoun, S.É., Gardner, P.: Gillian, part i: a multi-language platform for symbolic execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 927–942 (2020)
29. Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT workshop. vol. 2015 (2015)
30. Hutton, G.: Programming in haskell. Cambridge University Press (2016)
31. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019)
32. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA formal methods symposium. pp. 41–55. Springer (2011)

33. Lee, J., Kim, D., Hur, C.K., Lopes, N.P.: An SMT encoding of LLVM's memory model for bounded translation validation. In: Proc. of the 33rd International Conference on Computer-Aided Verification (CAV) (Jul 2021). https://doi.org/10.1007/978-3-030-81688-9_35
34. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. arXiv preprint arXiv:1404.6602 (2014)
35. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Sivaramakrishnan, K., Vouillon, J.é.: The OCaml system release 5.1: Documentation and user's manual. Ph.D. thesis, Inria (2023)
36. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Formal Methods in System Design **48**, 206–234 (2016)
37. Mann, M., Wilson, A., Zohar, Y., Stuntz, L., Irfan, A., Brown, K., Donovick, C., Guman, A., Tinelli, C., Barrett, C.: SMT-switch: a solver-agnostic C++ API for SMT solving. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 377–386. Springer (2021)
38. Marques, F., Fragoso Santos, J., Santos, N., Ad ão, P.: Concolic Execution for WebAssembly. In: 36th European Conference on Object-Oriented Programming (ECOOP 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2022)
39. Minsky, Y., Madhavapeddy, A., Hickey, J.: Real World OCaml: Functional programming for the masses. " O'Reilly Media, Inc." (2013)
40. Niemetz, A., Preiner, M.: Bitwuzla. In: International Conference on Computer Aided Verification. pp. 3–17. Springer (2023)
41. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, btormc and boolector 3.0. In: International Conference on Computer Aided Verification. pp. 587–595. Springer (2018)
42. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language (2004)
43. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25. pp. 53–68. Springer (2013)
44. Pimpalkhare, N., Mora, F., Polgreen, E., Seshia, S.A.: MedleySolver: online SMT algorithm selection. In: Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24. pp. 453–470. Springer (2021)
45. Preiner, M., Schurr, H.J., Barrett, C., Fontaine, P., Niemetz, A., Tinelli, C.: SMT-LIB release 2024 (non-incremental benchmarks) (Apr 2024). https://doi.org/10.5281/zenodo.11061097, https://doi.org/10.5281/zenodo.11061097
46. Rakadjiev, E., Shimosawa, T., Mine, H., Oshima, S.: Parallel SMT solving and concurrent symbolic execution. In: 2015 IEEE Trustcom/BigDataSE/ISPA. vol. 3, pp. 17–26. IEEE (2015)
47. Santos, J.F., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic execution for JavaScript. In: Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming. pp. 1–14 (2018)
48. Scott, J., Niemetz, A., Preiner, M., Nejati, S., Ganesh, V.: Algorithm selection for SMT: MachSMT: machine learning driven algorithm selection for SMT solvers. International Journal on Software Tools for Technology Transfer **25**(2), 219–239 (2023)

49. Slivkins, A., et al.: Introduction to multi-armed bandits. Foundations and Trends® in Machine Learning **12**(1-2), 1–286 (2019)
50. Torlak, E., Bodik, R.: Growing solver-aided languages with Rosette. In: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software. pp. 135–152 (2013)
51. Tuong, F., Le Fessant, F., Gazagnaire, T.: OPAM: an OCaml packa manager. In: ACM SIGPLAN OCaml Users and Developers Workshop (2012)
52. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11 (2012)
53. Wilson, A., Noetzli, A., Reynolds, A., Cook, B., Tinelli, C., Barrett, C.W.: Partitioning Strategies for Distributed SMT Solving. In: FMCAD. pp. 199–208 (2023)