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

João Madeira Pereira[1,2,3], Filipe Marques[1,2], Pedro Adão[1,4], Hichem Rami Ait-El-Hara[5], Léo Andrès[5], Arthur Carcano[5], Pierre Chambart[5], Petar Maksimović[6], Nuno Santos[1,2], and José Fragoso Santos[1,2]

[1] INESC-ID, Lisbon, Portugal
[2] Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
[3] Carnegie Mellon University, Pittsburgh, USA
[4] Instituto de Telecomunicações, Lisbon, Portugal
[5] OCamlPro, Paris, France
[6] Nethermind and Imperial College, London, UK

Abstract. SMT solvers are essential for applications in artificial intelligence, software verification, and optimisation. However, no single solver excels across all formula types, and different applications may require the use of different solvers. While the SMT-LIB language enables multi-solver support, it also 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. Furthermore, Smt.ml is the only SMT frontend that includes a simplification-management engine for streamlining the integration of new formula simplifications and the verification of their correctness. Our evaluation demonstrates that Smt.ml's results are consistent with those of its backend solvers and that its optimisations are highly effective on formulas generated from the symbolic execution of an extensive program-analysis benchmark.

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 [7,16,28,38], and even the optimisation of production chains in operations research [12].

While there are now multiple industry-strength SMT solvers for one to choose from, none of them is perfect for all applications. For example, recent SMT-COMP [14] results reveal considerable differences in solver ranking across various

theories. Hence, even within a single application, there may be benefits to using multiple solvers for tackling different types of formulas. However, switching between solvers often entails costly and error-prone integration work.

A common approach to interfacing with multiple solvers is to use SMT-LIB [9], a solver-agnostic textual format supported by almost all leading SMT solvers. In this approach, the given formula is serialised into an SMT-LIB formula and then the generated file is passed to the most appropriate solver for analysis. This textual interface, however, introduces I/O overhead, which can make it less suitable for performance-critical applications such as symbolic execution, program analysis, or synthesis engines. For efficiency, it is therefore often preferable for solvers to be integrated into the codebase as external libraries, linked directly via their native APIs. Unfortunately, these APIs differ widely in, for example, naming conventions, data representations, and type safety, making them challenging to understand and use in a uniform way.

While shared APIs for multiple solvers exist for some languages, such as Python and C++, they lack mechanisms to verify the correctness of any internal simplifications and do not address performance bottlenecks caused by repeated or redundant solver queries. Moreover, the OCaml ecosystem, which is home to a number of program-analysis and verification tools, has, until now, lacked a unified frontend for integrating multiple SMT solvers efficiently and safely.

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 by providing a consistent SMT-LIB-compatible language connected to five state-of-the-art solver backends: Alt-Ergo [21], Bitwuzla [48], Colibri2 [13], cvc5 [6], and Z3 [24]. With Smt.ml, OCaml developers do not need to understand any intricate detail of the API of these solvers to use them; they only have to create an Smt.ml formula and select the desired solver backend. Importantly, as part of the Smt.ml development effort, we created, for the first time, OCaml APIs for two SMT solvers: Colibri2 [13] and cvc5 [6].

The key novelties of Smt.ml when compared to other SMT frontends are:

1. a *parametric encoding* that relies on a common solver API to translate Smt.ml formulas into the native logic of each backend solver (§4);
2. a *simplification-management system* that allows developers to specify simplification rules using a new declarative domain-specific language (DSL), from which both their OCaml implementations and corresponding Lean [47] proof skeletons are automatically generated (§5.1);
3. a *caching system* for satisfiability results, which normalises SMT queries to maximise cache hits and avoid redundant computation (§5.2).

To the best of our knowledge, Smt.ml is the first SMT frontend to provide all these features simultaneously. The parametric encoding streamlines addition of new solvers to Smt.ml by requiring the developer to implement only a small, uniform set of functions, avoiding code duplication. The simplification management and caching systems are both solver-agnostic and together lead to a substantial increase in SMT performance on realistic program-analysis workloads. In addi-

tion, the former enables systematic verification of simplification correctness and provides a framework for extending the system with new verified simplifications.

We perform a comprehensive evaluation of Smt.ml on approximately 206K formulas from the official SMT-LIB benchmark [55] and 2.3M formulas obtained from symbolically executing the Test-Comp 2023 dataset [11] (§6). The results show that the behaviour of Smt.ml is fully consistent with the behaviour of the supported solvers, that the overhead of Smt.ml is negligible w.r.t. overall solving time (below 1% on average), and that the simplifications and caching of Smt.ml yield up to a 1.6x speed-up on formulas produced by symbolic execution.

We have made Smt.ml fully accessible to the OCaml and research communities. It is actively used in research projects across both academia [44,45] and industry [3], and has been integrated into OPAM [61], the OCaml package manager, simplifying its incorporation into future OCaml projects.

2 Why use Smt.ml?

In this section, we discuss in more detail the three main advantages that Smt.ml introduces for developers working with SMT solvers in OCaml.

Solver Independence. A key advantage of Smt.ml is its ability to interface with multiple SMT solvers through a single solver-independent frontend. In doing so, Smt.ml eliminates the need for developers to tailor their code to a specific solver API and lets them transparently switch between solvers, selecting whichever one is best for a given problem. This decision can even be made at runtime, allowing for the application of customised portfolio strategies [60].

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 [45]. Our evaluation, described in detail in §6, shows that Smt.ml, with its solver-agnostic formula simplifications and caching of satisfiability results, can introduce substantial performance improvements when compared to using any single SMT solver directly.

Usability. OCaml bindings for SMT solvers often provide few type safety guarantees, with many using one generic OCaml type to represent SMT expressions denoting different types of values, such as integers or strings. For instance, OCaml bindings for Z3 [24] use a single type for all general expressions regardless of

Listing 1 Type violation of the addition operator using Z3 OCaml bindings.

```
(* String value -> "4" *)
let four = Seq.mk_string ctx "4"
(* Integer value -> 2 *)
let two = Arithmetic.Integer.mk_numeral_i ctx 2
(* "4" + 2 >= 2 *)
let formula = Arithmetic.mk_ge ctx (Arithmetic.mk_add ctx [ four; two ]) two
```

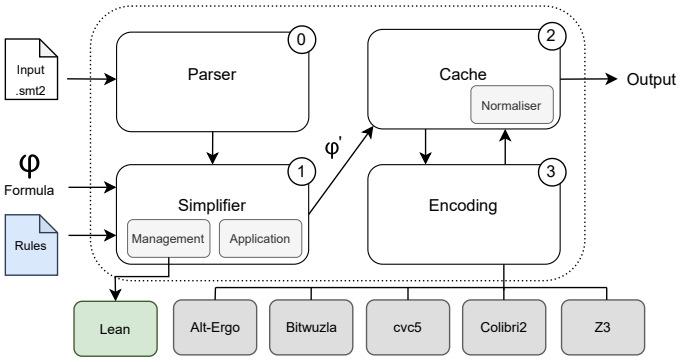

Fig. 1: Overview: Architecture of Smt.ml.

their underlying sort. This leads to ill-typed expressions not being detected at compile-time leading to hard-to-debug runtime errors. Listing 1 illustrates this issue by applying the addition operator, which expects two numeral arguments, to a string constant (line 6). This results in a runtime error indicating that the expected and actual argument types do not match. In contrast, Smt.ml features a typed API, ensuring that expressions are well-typed by construction, avoiding this class of bugs and promoting code correctness and reliability in the development process.

3 Architecture

Figure 1 presents an overview of the architecture of Smt.ml. While Smt.ml was primarily conceived to be used as a library within an OCaml application, it could, in principle, also be used as a standalone tool. Therefore, it accepts inputs either in the form of native Smt.ml formulas or SMT-LIB [9] textual formulas. In the case of the latter, the SMT-LIB formula is first parsed by the *Parser module (Step 0)* and converted into an Smt.ml native formula. Given a 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 bit-vector formulas:

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

  where the `concat` operator concatenates the two given bit-vectors and the `extract` operator returns the slice of the given bit-vector corresponding to the specified bounds, padding with additional zeros if necessary.
- *Cache module (Step 2):* This module normalises the given formula, checks if its satisfiability was already computed, and if so, returns the stored result. As part of the normalisation process, we rename symbolic variables in a standardised way to maximise cache hits.

TYPES
$t ::=$ `Tunit` | `Tbool` | `Tint` | `Treal` | `Tbitv` *int* | `Tfp` 32 | `Tfp` 64 | `Tstr` | `Tregexp` | `Tapp` | `Tlist`

VALUES
$v \in \mathcal{V}_{smt} ::=$ `unit` | `true` | `false` | `int` | `real` | `bitv` n | $f32$ | $f64$ | `str` | `regexp` | `list` v

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

COMMANDS
$c \in \mathcal{C}_{smt} ::=$ `declare`$(x_t) \mid$ `assert` $e \mid$ `check_sat` (`list` $e) \mid$ `get_model`
$\mid$ `get_value` $e \mid$ `pop` *int* $\mid$ `push` *int* $\mid$ `reset` $\mid$ `exit`

Fig. 2: The syntax of SMT.ML.

– *Encoding module (Step 3):* This module encodes SMT.ML expressions using the native OCaml bindings of the selected solver. It is parametric on a Core Solver API, simplifying the addition of new solver backends.

The simplifier module is also responsible for managing simplification rules, both converting them into the executable OCaml code that applies them and generating the Lean proof skeletons that, once completed, establish their correctness.

3.1 Syntax of SMT.ML

The syntax of SMT.ML is presented in Figure 2. There are two main syntactic categories: expressions, which denote values, and commands, which represent instructions given to the solver. SMT.ML currently supports the theories of quantifier-free linear integer and real arithmetic (`QF_LIA` and `QF_LRA`), bit-vectors (`QF_BV`), floating-point arithmetic (`QF_FP`), and strings (`QF_S`). These are the theories most commonly required for software verification and analysis tasks [5,17], which is the main application of SMT.ML. However, SMT.ML has a modular and extensible architecture, making it easy to add support for other theories.

Values. SMT.ML values, $v \in \mathcal{V}_{smt}$, include the unit value, booleans (`true` and `false`), integers, reals, machine integers (`bitv` n, where n denotes an arbitrary bit-width), IEEE 754 floating-point numbers (32 and 64-bit) [34], strings, regular expressions, and lists.

Expressions. Expressions, $e \in \mathcal{E}_{smt}$, consist of: values; typed symbolic variables x_t, where x denotes the variable identifier and t its type; and operators, which can be unary (`unop`, such as logical negation), binary (`binop`, such as addition), ternary (`triop`, such as bit-vector slicing), n-ary (`naryop`, such as list concatenation), relational (`relop`, such as comparisons), conversion-related (`cvtop`, such as casting from an integer to a string), or a special `list` operator that creates lists of expressions. Importantly, expression constructors enforce well-typedness by explicitly carrying the expression type.

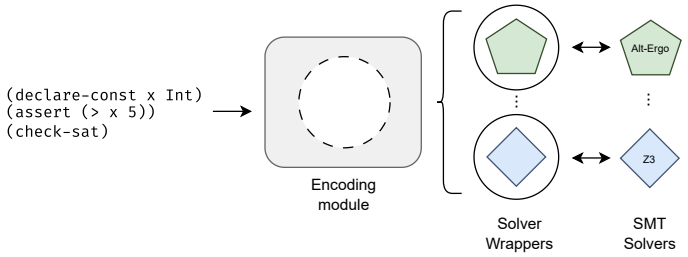

Fig. 3: The parametric Encoding module of Smt.ml.

Commands. Smt.ml provides the developer with a set of commands for interacting with SMT solvers. In particular, one can: declare a symbolic variable x of a given type t; assert that a given Boolean expression (i.e., formula) holds; check satisfiability of the current solver state additionally assuming a given list of formulas; get a model for the last satisfiability check; get a model for a given expression in the context of the last satisfiability check; introduce/remove a number of assertion levels; and reset/terminate the interaction with the solver.

4 Encoding

The Encoding Module is the core component of Smt.ml that is responsible for translating native Smt.ml expressions and commands into the expressions and commands of the solver backends. Instead of having a separate encoding process for each backend, we identify a set of fundamental functionalities required for solver interaction and bring them together to form a parametric *Core Solver API*, which we then use to define a solver-independent encoding. For a solver to be integrated into Smt.ml, all that it needs to do is: support integration with OCaml via native bindings; implement our Core Solver API as a wrapper around these bindings; and plug this implementation into Smt.ml. This parametric approach, which is illustrated in Figure 3, substantially streamlines the addition of new solver backends to Smt.ml and avoids code duplication.

Importantly, extending a given solver with the wrapper code that implements the Core Solver API is significantly easier than implementing a solver-specific encoding from scratch. In particular, as nearly all the functions required by the Core Solver API are typically present in native solver APIs, implementing the wrapper code only amounts to mapping their native names to those expected by the Core Solver API.

The Core Solver API, denoted by $\mathcal{S}$, encompasses all functions on solver values, expressions, and commands on which the encoding of Smt.ml expressions and commands depends. These functions can be divided into the following four main categories:

– *Values:* these functions are responsible for mapping Smt.ml primitive values, $v \in \mathcal{V}_{smt}$, into values from the corresponding target solver;

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

$$\textsc{n-ary operators}\ \frac{T_{\mathcal{S}}(e_i) = e_i'\ |_{i=1}^{n} \qquad \mathcal{S}.naryop(nop, [e_1', \ldots,\ e_n']) = e'}{T_{\mathcal{S}}(nop\ [e_1, \ldots,\ e_n]) = e'}$$

Fig. 4: Parametric translation rules for SMT.ML (excerpt).

- *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:* these functions map SMT.ML commands, $c \in \mathcal{C}_{smt}$, into commands that manipulate and interact with the target solver; and
- *Lifting:* these functions map solver values back to SMT.ML values, and are essential when performing model extraction, as they allow models to be constructed using native SMT.ML values.

Parametric Translation. Using the API described above, we implement a generic translation from SMT.ML constructs to the corresponding solver-specific constructs. Expression translation is formalised 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 $te \in \mathcal{S}.Expr$. An excerpt of the translation rules is shown in Figure 4.

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 by applying the solver's unary operator to the translated argument using the Core Solver API function $\mathcal{S}.unop(uop, te)$, which produces the solver expression denoting uop applied to te. The translation proceeds similarly for all other kinds of operators.

SMT.ML supports five backend solvers: Alt-Ergo, Bitwuzla, Colibri2, cvc5, and Z3. For each of these solvers we have implemented the Core Solver API. Furthermore, for Colibri2 we needed to implement a user-facing API to enable solver interaction, and for cvc5 we had to implement OCaml bindings from scratch [41], as none were originally available. In this way, we allow for cvc5 to be natively integrated not only into SMT.ML but also into any other OCaml-based tool.

5 Backend-independent Optimisations

The usefulness of SMT.ML 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 satisfiability. Here, we discuss the two most important such optimisations: expression simplifications (§5.1) and caching (§5.2).

5.1 Expression Simplifications

In applications that interact with SMT solvers, the size and complexity of the problem at hand can significantly affect solver efficiency, making performance the primary bottleneck [2,64]. To address this, Smt.ml tries to reduce expression complexity by applying a set of semantics-preserving simplifications before these expressions are passed to a solver. Currently, Smt.ml comes with 42 simplification rules, spanning the theories of bit-vectors (11 rules), strings (3 rules), booleans (2 rules), and 26 generic rules that are applicable to multiple theories.

Importantly, instead of hardcoding these simplifications directly into the OCaml codebase, we design a simple domain-specific language (DSL) that enables their declarative specification. This allows users of Smt.ml to easily examine, add, or remove simplifications by need. From these specifications, we automatically generate the corresponding OCaml implementations and, if applicable, Lean [47] proof skeletons, allowing users to formally prove simplification correctness by hand, thereby reducing the trusted computing base of Smt.ml. While there is prior work on declarative languages for specifying SMT simplifications [50] and on methods for their validation [37], Smt.ml is the first SMT frontend to adopt this approach.

Simplification management. Smt.ml simplification rules, $r \in \mathcal{R}$, are of the form $e_1 \Rightarrow e_2$ **when** e_f, meaning that if the boolean expression e_f holds, then the expression e_1 can be rewritten to e_2. For instance, consider the simplification rule:

$$\begin{aligned} &\texttt{Extract}(bv, h, l) \Rightarrow bv \\ &\quad \textbf{when} \;\; h < \texttt{size}(bv) \;\texttt{\&\&}\; l \geq 0 \;\texttt{\&\&}\; \texttt{size}(bv) = h - l + 1 \end{aligned} \tag{1}$$

where `Extract` is an Smt.ml operator that returns the slice of a given bit-vector x from bit l to bit h inclusive, padding with zeros if the slice falls outside x. This rule says that $\texttt{Extract}(x, h, l)$ can be rewritten to x when l is non-negative, h is within the bounds of x and $h - l + 1$ equals the size of x. From this rule, Smt.ml automatically generates the OCaml code that implements it (Fig. 5, left), as well as the corresponding Lean proof skeleton (Fig. 5, right).

```
let simplify_triop ty op hte1 hte2 hte3 =
  match op, hte1, hte2, hte3 with
  | ... (* other simplification cases *)
  | Extract, bv, Val (Int h), Val (Int l)
      when l >= 0
       and h < Ty.size (ty bv)
       and h - l + 1 = Ty.size (ty bv)
      -> bv
```

```
lemma simplification_triop_000004
  {w : Nat}
  {h l : Int}
  (n : BitVec w)
  (h0 : l >= 0)
  (h1 : h < w)
  (h2 : (w : Int) = h - l + 1) :
BitVec.extractLsb h l n = n := by ...
```

Fig. 5: Example: generated OCaml code (left) and Lean proof skeleton (right).

Simplification application. At runtime, Smt.ml takes the formulas that are to be checked for satisfiability and, for each such formula, keeps looping over the simplification rules, applying those whose constraints are satisfied until no further rules can be applied; this process is illustrated in Algorithm 1. Importantly, Smt.ml keeps track of simplified expressions and, when a new expression is simplified,

Algorithm 1 Expression simplification algorithm.

```
1: procedure SIMPLIFY(e)
2:     e' ← e
3:     for all r ∈ Rules do
4:         e' ← apply r to e'
5:     if e' ≠ e then
6:         return SIMPLIFY(e')
7:     else
8:         return e'
```

it checks if it has occurred before ensuring that the collective application rules does not result in infinite loops, such as $e \xrightarrow{r_1} e' \xrightarrow{r_2} e$, where e is the original expression, e' the simplified expression, and r_1 and r_2 two simplification rules.

To apply a simplification rule $e_1 \Longrightarrow e_2$ **when** e_f to a given expression e in execution context e'_f, SMT.ML proceeds as follows:

1. find a substitution θ such that $\theta(e_1) = e$;
2. if successful, check that $e'_f \models \theta(e_f)$;
3. if successful, replace e with $\theta(e_2)$.

For instance, in execution context $\pi \equiv |z| = 2$, applying the simplification from Equation 1 to the expression $\texttt{extract}(\texttt{concat}(y, z), |y|+2, 0)$ yields the expression $\texttt{concat}(y, z)$ with substitution $\theta = [x \mapsto \texttt{concat}(y, z), h \mapsto |y| + 2, l \mapsto 0]$.

Lean correctness proofs. We generate Lean proof skeletons for 33 out of the 42 simplifications and provide the corresponding proofs. As all of our simplifications describe basic properties of the underlying datatypes, their proofs are very simple and highly automated, relying on the comprehensive mathematical library of Lean. The remaining 9 simplifications are purely *definitional*, in that they describe the behaviour of SMT.ML operators on concrete inputs in terms of the corresponding OCaml operators. One such simplification, for example, is:

$$\texttt{Length}(l) \Rightarrow \texttt{List.length}\ l\ \textbf{when}\ \texttt{concrete}(l) \tag{2}$$

which declares that the SMT.ML `Length` operator coincides with the OCaml `List.length` operator. Note that this simplification uses the $\texttt{concrete}(x)$ predicate in the **when** clause, which holds if and only if x is concrete.

5.2 Caching

Caching of intermediate satisfiability results is a standard technique used in SMT solvers and solver clients to improve performance [52,56]. 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 [2,62] typically implement normalisation strategies [33] 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 disjunction operator, $\vee$, 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.

In addition to minimising the number of queries, one can also enhance the performance of solver clients by reducing the number of expressions created at runtime. In fact, solver clients often generate a large number of expressions, frequently with repeated elements. As the number of queries grows, memory consumption increases, impacting client performance; a prime example of this are symbolic execution engines [18]. The standard approach to reducing the memory impact of such systems is *hash-consing* [27], 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 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 of 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, we add it to the table and return the value constructed in line 3. One might notice that in line 3 we allocate memory to construct an expression, only to later use an existing one retrieved from the hash-consing table. However, as OCaml initially allocates values in the minor heap using a bump allocator [46], this allocation incurs no cost. Additionally, OCaml will collect the temporarily allocated values during minor garbage collection.

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 with Not_found -> Hashtbl.add table x x; x
```

6 Evaluation

We evaluate Smt.ml with respect to the five following questions:

- *EQ1:* Does Smt.ml exhibit behaviour consistent with the supported solvers?
- *EQ2:* How much does Smt.ml's overhead impact overall performance?
- *EQ3:* How do Smt.ml's optimisations impact its overall performance?
- *EQ4:* Are Smt.ml's simplifications transparent and trustworthy?
- *EQ5:* How does Smt.ml compare to other SMT frontends?

All experiments were performed on a server with a 12-core Intel Xeon E5-2620 v4 CPU and 32GB of RAM, running Ubuntu 24.04.1 LTS. We compiled Smt.ml using the OCaml 5.3.0 compiler. For the SMT solvers, we used Alt-Ergo v2.6.2, Bitwuzla v0.8.0, Colibri2 development version (commit `1feba887`), cvc5 v1.3.0, and Z3 v4.13.0. Each benchmark was run with a timeout of 60s and 10GB memory limit. The benchmarking code, reproducibility scripts, and diagram generation scripts are all available at [29].

6.1 Datasets

To assess the correctness (§6.2) and performance overhead (§6.3) of Smt.ml, we used a subset of the official SMT-LIB benchmark [55]. In particular, we used approximately 206K SMT formulas that comprise the (quantifier-free) theories of: linear integer arithmetic (`QF_LIA`), floating-point arithmetic (`QF_FP`), bit-vector arithmetic (`QF_BV`), string theory (`QF_S`), and string theory with linear integer arithmetic (`QF_SLIA`). We chose these theories as they are the ones both most commonly used in practice [14] and well-supported by the Smt.ml backends.

To assess how the optimisations of Smt.ml impact performance (§6.4), we used a dataset of approximately 2.3M SMT formulas generated through symbolic execution of the following sub-categories from the `c/ReachSafety` category of the official Test-Comp 2023 benchmark suite [11]: Arrays, Bit-vectors, Heap, ProductLines, and Sequentialized. These categories provide a diverse set of formulas commonly encountered during symbolic execution of binary code. For formula generation, we used the Owi symbolic execution engine [3] as it does not implement its own formula caching system and includes only a small set of built-in optimisations, thereby allowing for a more trustworthy assessment of the true impact of Smt.ml.

Note that the formulas found in the SMT-LIB benchmarks by design share few commonalities and lack redundancies by design. This is the exact opposite of our intended use case for Smt.ml, which is integration with tools that interact multiple times with a solver during execution and generate similar formulas in doing so, such as symbolic execution tools. For this reason, we do not evaluate the optimisations of Smt.ml against the SMT-LIB benchmarks.

6.2 EQ1: Behaviour consistency

To assess the correctness of Smt.ml, we compared the results obtained when running Smt.ml and when running each supported solver directly on the targeted SMT-LIB benchmarks. This cross-validation is facilitated by the fact that most SMT-LIB benchmarks are annotated with their expected outcome.

Results. All results produced by Smt.ml matched those directly produced by the solvers and were aligned with the provided expected outcomes.

Takeaway 1: The behaviour of Smt.ml is fully consistent with the behaviour of its backend solvers.

Table 1: Average times on SMT-LIB benchmarks: Smt.ml vs. backend solvers.

Theory	Smt.ml		Backend solver check-sat	
	avg. (ms)	(%)	avg. (ms)	(%)
QF_BV	33.90	0.97	3460.69	99.03
QF_FP	0.36	0.01	9794.61	99.99
QF_LIA	20.97	3.13	648.97	96.87
QF_S	0.93	0.38	238.27	99.62
QF_SLIA	0.56	0.37	155.02	99.63
Total	11.09	0.50	2215.12	99.50

6.3 EQ2: Performance overhead

To quantify the overhead introduced by Smt.ml, we measured the time spent in Smt.ml versus the time spent in the backend solvers on the SMT-LIB benchmarks. Smt.ml's tasks include parsing SMT-LIB formulas, applying simplifications, and encoding formulas for each of the backend solver APIs. For each theory, we used the times obtained when using the solver that performed best on average for that theory. Table 1 reports the average time spent in Smt.ml and in the solvers for each theory, along with the corresponding percentages.

Results. The obtained results show that the overhead of Smt.ml is minimal when compared to solver runtime, staying below 1% for all theories except `QF_LIA` and rising to at most 3.13%, for `QF_LIA`. This larger percentage is due to the corresponding formulas being larger, and thus their parsing times.

Takeaway 2: Smt.ml introduces only negligible overhead on top of the time spent in its respective backend solvers.

6.4 EQ3: Performance impact on symbolic execution benchmarks

We evaluated the impact of the Smt.ml backend-independent optimisations through an ablation study using a dataset of SMT formulas generated by the Owi symbolic execution engine [3] while executing the Test-Comp 2023 benchmark suite [11]. We considered the following three configurations:

- *Raw*: no optimisations, formulas are sent directly to the backend solver;
- *Simplifications*: formulas are simplified before being sent to the solver;
- *Simplifications and caching*: simplified formulas are also cached to avoid redundant computation.

Figure 6 summarises the cumulative runtime for each configuration across the selected benchmark categories. We consider only cvc5 and Z3 as they support all of the theories required for the dataset at hand and are also the most performant.

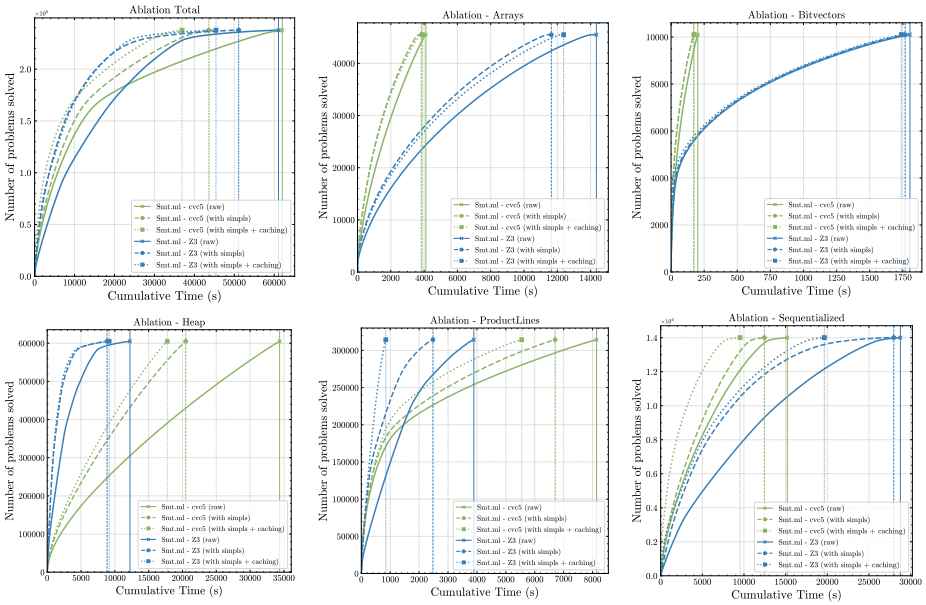

Fig. 6: Ablation study results by Test-Comp 2023 benchmark category.

Results. The results indicate that, on the whole, the backend-independent optimisations significantly improve performance. In particular, the total results for the considered dataset (Figure 6, top left), show that the two optimisations together yield a 1.59× speedup with Z3 and 1.54× with cvc5 relative to the Raw configuration baseline. When examining categories in isolation, we can see, for example, that for `c/ReachSafety-Arrays` and `c/ReachSafety-Heap` simplifications alone account for most of the performance improvement, whereas for `c/ReachSafety-ProductLines` and `c/ReachSafety-Sequentialized` it is the caching that dominantly speeds up the execution. In particular, for the latter category, caching exhibits 1.42× and 1.31× speedups over the Simplifications only configuration for Z3 and cvc5, respectively. These differences can be attributed to the nature of the formulas in each category: categories with many repeated sub-expressions benefit more from caching, whereas categories with complex but less repetitive formulas benefit primarily from simplifications.

Takeaway 3: The backend-independent optimisations significantly improve the performance of SMT.ML on symbolic execution benchmarks.

6.5 EQ4: Transparency and trustworthiness of simplifications

By developing a DSL for writing simplifications and a mechanism for automatic generation of corresponding OCaml code and Lean proof skeletons, we have provided users of SMT.ML with an easy-to-use simplification management system. When it comes to simplification correctness, on the one hand, the definitional

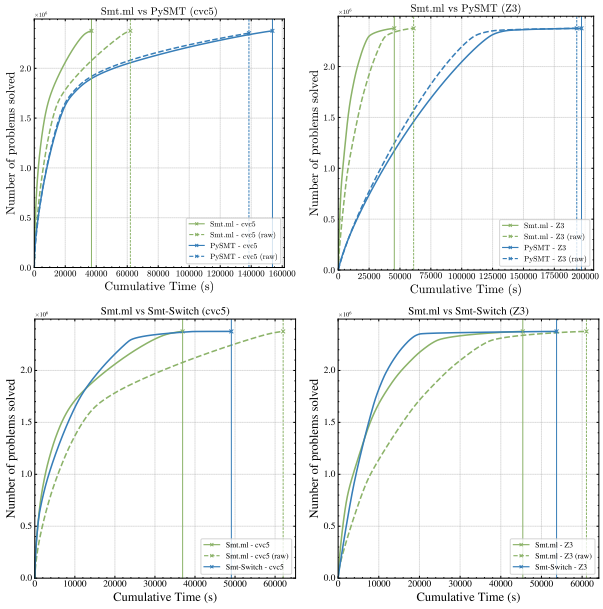

Fig. 7: Comparison: SMT.ML vs. PySMT (top); SMT.ML vs. Smt-Switch (bottom).

simplifications only connect SMT.ML operators to basic OCaml data-structure operators in a way that if one trusts the latter, one should also trust the former. On the other hand, the correctness of all 33 non-definitional simplifications has been fully proven in Lean. At a higher level, we opted to prove simplification correctness in a proof assistant rather than using SMT as some simplifications require inductive reasoning. For instance, to prove that reversing a list twice yields the original list, $\texttt{Rev}(\texttt{Rev}(l)) \Rightarrow l$, one must induct on the size of the list.

Takeaway 4: Users can easily examine and manage SMT.ML simplifications. All of these simplifications are trustworthy: most are formally verified in Lean, and the remaining ones rely on the correctness of fundamental OCaml operators.

6.6 EQ5: Comparison with other SMT frontends

We identified six relevant open-source SMT frontends: JavaSMT [4], PySMT [31], rsmt2 [19], SBV [26], Smt-Switch [43], and what4 [35]. We excluded JavaSMT, rsmt2, SBV, and what4 as they do not accept SMT-LIB input, making it impossible to measure their performance using standard benchmarks.

We compare SMT.ML against the remaining frontends: PySMT [31] and Smt-Switch [43]. For Smt-Switch we found an issue when trying to process multiple formulas with the same solver instance, as it would throw an error when given two independent formulas in which there is a shared variable name. To perform the comparison, we therefore create a new solver instance and SMT-LIB reader for each formula. We evaluated both frontends on the symbolic execution

dataset described in §6.4, using cvc5 and Z3 as backend solvers, as they are the only common solvers supported by both Smt.ml, PySMT, and Smt-Switch. We evaluated both Smt.ml and PySMT under two settings: one with all optimisations enabled and one without any optimisations. Smt-Switch was evaluated only in its standard configuration, as it does not support optimisations.

Results. Figure 7 shows the comparisons between Smt.ml and PySMT (top diagrams), and between Smt.ml and Smt-Switch (bottom diagrams). The results show that, using the configuration with the most optimisations enabled, Smt.ml substantially outperforms PySMT, with speedups of 4.34× and 4.25× with Z3 and cvc5, respectively. On the other hand, disabling optimisations on both frontends narrows the gap between them, with speedups of 3.16× and 2.26× with Z3 and cvc5, respectively. This can be partly attributed to the implementation languages: Smt.ml is implemented in OCaml, a compiled language, whereas PySMT is written in Python, an interpreted language that incurs additional overhead. We note that the raw configuration for PySMT outperforms its optimised configuration possibly indicating that the simplifications implemented by PySMT are not effective on the Test-Comp formulas. The results also show that, with optimisations enabled, Smt.ml outperforms Smt-Switch, with speedups of 1.18× for Z3 and 1.33× for cvc5. However, under the raw configuration, Smt.ml is slower than Smt-Switch, with slowdowns of 0.88× and 0.79× for Z3 and cvc5, respectively. This can be attributed to the fact that Smt-Switch is implemented in C++, which is a more performant language than OCaml.

Takeaway 5: With optimisations enabled, Smt.ml outperforms both PySMT and Smt-Switch on symbolic execution benchmarks.

7 Related Work

SMT Solvers. SMT solvers have seen significant advancements since their emergence in the early 2000s [6,8,10,24,25], such as support for new theories, like strings [40] and quantified arithmetic [1], and new optimisation techniques, like new caching mechanisms [62] and portfolio strategies [54,59]. As a result, they are now used across the entire computer science community, with a wide variety of applications ranging from software verification and test generation [30,36,39] to combinatorial optimisation and classical operations research [12].

Currently, many SMT solvers are actively maintained, with 20 submitted to the 2024 edition of SMT-COMP [14]. Importantly, there is no one-size-fits-all solver: some excel in handling certain theories, others excel in handling others. Our solver selection for Smt.ml integration is guided by our specific needs. Initially, we supported Z3 [24] and Colibri2 [13], as they were already being used in our ongoing projects. Subsequently, we added support for cvc5 [6] and Bitwuzla [48], due to their excellent performance with bit-vectors and floating-point arithmetic, which are frequently required in symbolic execution. Most recently, we have also interfaced with Alt-Ergo [21], because it is the only SMT

solver implemented entirely in OCaml. This makes it particularly attractive for our ecosystem, as it enables applications depending on Alt-Ergo to be compiled with `js_of_ocaml` [63], allowing native execution directly within the browser.

Frontends for SMT Solvers. Frontends [4,19,26,31,35,43] play an essential role in making SMT solvers more accessible to the broader 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.

The two solver frontends closest to Smt.ml in spirit are PySMT [31] and Smt-Switch [43]. They are written in Python and C++, respectively, and equip users with a high-level API for interacting with various SMT solvers, abstracting low-level solver-specific details. PySMT supports five solvers, while Smt-Switch supports six. In particular, PySMT supports Z3, cvc5, Yices2 [25], Boolector [49], and MathSAT5 [20], whereas Smt-Switch supports Z3, cvc5, Yices2, MathSAT5, Bitwuzla, and Boolector. PySMT further implements a solver-agnostic optimisation layer that uses a set of simplification rules not unlike ours before passing the formulas to the solvers. However, Smt.ml is the only SMT frontend that includes a simplification-management engine for streamlining the integration of new simplifications and the verification of their correctness. Furthermore, unlike PySMT and Smt-Switch, Smt.ml also includes an integrated caching system, which is essential for containing memory consumption in our target applications and further improves performance. Finally, as the first SMT-solver frontend for OCaml, Smt.ml makes it easier for program analysis and verification tools implemented in OCaml (e.g., Frama-C [22], Binsec [23], Gillian [30]) to interface with multiple SMT solvers and take advantage of its optimisations.

Caching and Simplifications for SMT Solvers. While most SMT solvers apply some built-in simplifications as part of a preprocessing step before the core satisfiability check [6,24], the practical scope of these simplifications is often limited. For this reason, prior work has explored augmenting SMT workflows with additional simplification capabilities by either integrating them into the solver codebase [51] or applying them as a preprocessing stage before invoking the solver [30,32,42]. Empirical evidence shows that such simplifications are important for tool performance: for example, the verification of the real-world AWS code done in [42] would not have been tractable without simplifications. Prior work has also explored the use of domain-specific languages for specifying simplifications [50] and proving them using theorem provers [37]. Our contribution differs from these systems in that we propose a declarative DSL for specifying simplifications, from which we generate *both* their implementations and proof skeletons. This design streamlines the management and verification of simplifications, keeping them out of the trusted computing base.

The primary SMT application that benefits from formula caching is symbolic execution [5,17]. Symbolic execution of a program produces path formulas that

record the branch conditions encountered during execution; these formulas often contain redundancies across different executions and even within the same one. For this reason, many high-profile symbolic execution tools implement formula caching [16,53,58], often reimplementing similar code. To address this, the Green system was proposed to factor formula caching out of individual symbolic execution tools so that different tools can share a common cache [62]. We follow the same philosophy in Smt.ml, shifting the responsibility for caching from SMT clients to Smt.ml. Unlike Green, however, we combine formula caching with hash-consing to reduce memory footprint.

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 introduces a simplification-management engine that allows users to specify formula simplifications and prove their correctness. Our evaluation shows that Smt.ml's maintains consistent behaviour with its backend SMT solvers and introduces no significant overhead. We further demonstrate that, for formulas generated from the symbolic execution of programs, Smt.ml's optimisations are highly effective, leading to significant performance improvements.

In the future, we plan to explore a number of pathways for advancing the work on Smt.ml. Firstly, we will continue to add support for new solver backends and also offer developers the option to create solver portfolios parameterised on solver selection strategies. Moreover, we plan to improve our simplification module by adding further, more complex, simplifications, based on those currently present in state-of-the-art symbolic executors [30,57]. Finally, we will also investigate the option of integrating large language models with Smt.ml, with the aim of automating the proofs of simplification correctness.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) via grants 2024.18500.PRT and PRT/BD/154029/2022, as well as the projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025), and WebCAP (ref. 2024.07393.IACDC, DOI: https://doi.org/10.54499/2024.07393.IACDC), and by IAPMEI under grant ref. C6632206063-00466847 (SmartRetail).

Data Availability Statement

The artifact associated with this paper, including the implementation of the tool and scripts for reproducing the experimental results, is available at https:

//doi.org/10.5281/zenodo.17489264. The benchmarks used in the evaluation are available at https://doi.org/10.5281/zenodo.16740866 (SMT-LIB benchmark suite) and https://doi.org/10.5281/zenodo.17474996 (dataset of formulas generated from the symbolic execution of subset of the Test-Comp 2023 benchmark suite). For reuse and further development, the current version of the Smt.ml's source code is maintained in a public GitHub repository at [29].

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. Amrollahi, D., Preiner, M., Niemetz, A., Reynolds, A., Charikar, M., Tinelli, C., Barrett, C.: Towards smt solver stability via input normalization (2025), https://arxiv.org/abs/2410.22419
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. Baier, D., Beyer, D., Friedberger, K.: Javasmt 3: Interacting with smt solvers in java. In: International Conference on Computer Aided Verification. pp. 195–208. Springer (2021)
5. 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)
6. 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)
7. 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)
8. 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)
9. 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)
10. Barrett, C., Tinelli, C.: CVC3. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 4590, pp. 298–302. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_34, https://doi.org/10.1007/978-3-540-73368-3_34
11. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. Fundamental Approaches to Software Engineering LNCS 13991 p. 309 (2023)
12. 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

13. 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
14. Bromberger, M., Bobot, F., , et al.: The International Satisfiability Modulo Theories Competition (SMT-COMP)(2024). https://smt-comp.github.io/
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. Champion, A.: rsmt2. webpage: https://docs.rs/rsmt2/latest/rsmt2/index.html, source code: https://github.com/kino-mc/rsmt2
20. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7, https://doi.org/10.1007/978-3-642-36742-7_7
21. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018)
22. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. In: International conference on software engineering and formal methods. pp. 233–247. Springer (2012)
23. Daniel, L.A., Bardin, S., Rezk, T.: Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1021–1038. IEEE (2020)
24. 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)
25. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
26. Erkok, L.: SBV. webpage: https://hackage.haskell.org/package/sbv, source code: https://github.com/LeventErkok/sbv
27. 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
28. 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)
29. formalsec: smtml: An smt solver frontend for ocaml. https://github.com/formalsec/smtml

30. 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)
31. Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT workshop. vol. 2015 (2015)
32. Grannan, Z., Vazou, N., Darulova, E., Summers, A.J.: Rest: Integrating term rewriting with program verification (extended version). arXiv preprint arXiv:2202.05872 (2022)
33. Guilloud, S., Bucev, M., Milovančević, D., Kunčak, V.: Formula normalizations in verification. In: International Conference on Computer Aided Verification. pp. 398–422. Springer (2023)
34. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019)
35. Inc., G.: What4. source code: https://github.com/GaloisInc/what4
36. 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)
37. Lachnitt, H., Fleury, M., Aniva, L., Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: IsaRare: Automatic verification of SMT rewrites in Isabelle/HOL. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 14570, pp. 311–330. Springer (Apr 2024). https://doi.org/10.1007/978-3-031-57246-3_17, http://theory.stanford.edu/~barrett/pubs/LFA+24.pdf
38. 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
39. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. arXiv preprint arXiv:1404.6602 (2014)
40. 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)
41. Madeira Pereira, J., Marques, F.: cvc5 OCaml bindings. source code: https://github.com/formalsec/ocaml-cvc5
42. Maksimovic, P., Ayoun, S., Santos, J.F., Gardner, P.: Gillian, part II: real-world verification for javascript and C. In: Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 827–850. Springer (2021)
43. 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)
44. Marques, F., Ferreira, M., Nascimento, A., Coimbra, M.E., Santos, N., Jia, L., Fragoso Santos, J.: Automated exploit generation for node. js packages. Proceedings of the ACM on Programming Languages **9**(PLDI), 1341–1366 (2025)
45. 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)
46. Minsky, Y., Madhavapeddy, A., Hickey, J.: Real World OCaml: Functional programming for the masses. " O'Reilly Media, Inc." (2013)

47. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25. pp. 378–388. Springer International Publishing, Cham (2015)
48. Niemetz, A., Preiner, M.: Bitwuzla. In: International Conference on Computer Aided Verification. pp. 3–17. Springer (2023)
49. 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)
50. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: FMCAD. pp. 65–74 (2022)
51. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C., Tinelli, C.: Syntax-guided rewrite rule enumeration for smt solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 279–297. Springer (2019)
52. 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)
53. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. pp. 179–180 (2010)
54. 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)
55. Preiner, M., Schurr, H.J., Barrett, C., Fontaine, P., Niemetz, A., Tinelli, C.: SMT-LIB release 2025 (non-incremental benchmarks) (Aug 2025). https://doi.org/10.5281/zenodo.16740866, https://doi.org/10.5281/zenodo.16740866
56. 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)
57. Roşu, G., Şerbănuţă, T.F.: An overview of the k semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010)
58. Ryan, K., Sturton, C.: Sylq-sv: Scaling symbolic execution of hardware designs with query caching. In: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. pp. 195–211 (2025)
59. 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)
60. Slivkins, A., et al.: Introduction to multi-armed bandits. Foundations and Trends® in Machine Learning **12**(1-2), 1–286 (2019)
61. Tuong, F., Le Fessant, F., Gazagnaire, T.: OPAM: an OCaml packa manager. In: ACM SIGPLAN OCaml Users and Developers Workshop (2012)
62. 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)

63. Vouillon, J., Balat, V.: From bytecode to javascript: the js_of_ocaml compiler. Software: Practice and Experience **44**(8), 951–972 (2014)
64. 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)