PySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms

Marco Gario[1,2] and Andrea Micheli[1,2]

[1] University of Trento
[2] Fondazione Bruno Kessler
{gario, amicheli}@fbk.eu

Abstract

Satisfiability Modulo Theory is increasingly being used as workhorse in multiple and disparate domains. Several efficient solvers exist and cross-compatibility is provided by a wide adoption of the SMT-Lib interface. Nevertheless, there are still some obstacles that slow down the development of algorithms based on the SMT technology. First, there is no common programmatic API among different solvers: a user needs to commit to a specific solver API early in the development, or implement a text-based interaction via the SMT-Lib interface, thus losing the ability of exploiting features provided by solvers that are not part of the SMT-Lib. Second, there is no high-level SMT API for fast prototyping: a user often needs to re-implement common routines such as substitution and type-checking. In this paper, we introduce PySMT, an open-source python library that provides a solver agnostic interface to define, manipulate and solve SMT formulae. PySMT leverages the native APIs of solvers (if available) or their SMT-LIB interface. The library allows the seamless combination of different solvers and fast prototyping of complex SMT-based algorithms.

1 Introduction

Nowadays, SMT solvers are on the spotlight for their success in several application domains. They are used as workhorses in several formal verification tools [34, 15, 26, 12] as well as in program analysis [24, 19], theorem proving [29], automatic test generation [31, 33, 11] and in many other settings. Apart for the intrinsic efficiency and effectiveness of the solvers, one key aspect of this success story is due to the standard input language that almost all the solvers support. The SMT-Lib initiative [3, 4] defines a standard input language designed to be universal and easy to parse. As shown by the annual SMT-Competition [2], the existence of a common input format provides a simple way to run different solvers on the same problem, comparing the strength and weaknesses of each one.

The SMT-Lib format works well when a problem can be written beforehand and then passed to the solver. However, in many practical situations, a top-level algorithm and an SMT solver interact, in such a way that the results of the SMT-queries drive the caller algorithm. In this situation, two approaches are common: directly call the solvers API or send textual commands in SMT-Lib format to a solver using a POSIX pipe. The first approach yields better performances and allows the user to access features of the solver that are not covered by the SMT-Lib specification. However, this approach constitutes a lock-in to the chosen solver. Different solvers have different primitives to create and manipulate expressions, and slightly different ways to call the solving routines and gather the results. If direct calls to the APIs are present, it becomes difficult to swap the underlying solver with a different one. However, the possibility to experiment with different solvers is often desirable: another solver might yield better performances or provide some useful feature.

If interoperability is required, one usually exploits the SMT-Lib by executing a solver in an external process, feeding the input and parsing the output. This approach limits the possible actions to the ones defined by the SMT-Lib initiative. Moreover, the calling application must deal with a set of well known book-keeping activities and features, such as definition and memoization of expressions (to avoid representing the expressions as trees), SMT-Lib (de-)serialization, a minimal level of type-checking, human readable serialization (to simplify debugging) and several other boiler-plate code that performs formula manipulation.

PySMT tries to bridge the gap between the power and specificity of a solver API and the generality of the SMT-Lib. PySMT takes care of all the book-keeping and boilerplate code needed in order to use SMT solvers: it provides a solver agnostic way of creating and manipulating SMT formulae and a seamless integration with the underlying API of solvers. The goal of PySMT is to make fast-prototyping of SMT-based algorithms as simple as possible, while maintaining compatibility across a wide range of solvers. PySMT allows the user to quickly experiment new SMT-based algorithms and evaluate the performance of different solvers on the same algorithm. Moreover, it provides access to solver capabilities that are not currently covered by the SMT-Lib standard, such as (theory) quantifier elimination and interpolation.

We initially developed PySMT to solve problems dealing with mixed integer and rational linear arithmetic ($\mathcal{UFLIRA}$), but the library also supports bit-vectors and pure Boolean formulae, with the extension to arrays being underway. PySMT transparently supports a wide range of solvers through their native API; moreover, it allows the use of any solver supporting the SMT-Lib interface.

This paper provides a brief introduction to PySMT and its main features. In Section 2 we present more in detail the architecture of the library, and discuss application case studies. Section 3 is dedicated to present some usage examples. Finally, in Section 4 we discuss the related work while Section 5 presents future directions for the library development.

2 PySMT

PySMT is a library for solver-agnostic formulae manipulation, meaning that you can build an expression, modify it, perform basic operations such as simplification, substitution, visualization, and file-based export and import. All of this without the need of having any SMT solver installed. This is achieved as a pure-python implementation of common formula management features. This abstract level makes it possible to quickly prototype variants of the same algorithm, and then evaluate it on different solvers. In this way, it is possible to focus on the algorithm, and decide which solver to use as a second step, without the need of having a deep knowledge of the chosen solver API.

At the moment of writing, PySMT supports the representation and the satisfiability checking of quantified logics modulo the $\mathcal{UFLIRA}$ (and all its fragments) and $\mathcal{BV}$ theories. Several solvers are currently integrated through their native APIs: MathSAT [16], Z3 [21], CVC4 [1] and Yices [22]. We also integrated solvers to perform pure Boolean reasoning, i.e., the CUDD BDD package [32] and the PicoSAT [5] SAT-solver. Moreover, any SMT-Lib compliant solver can be integrated through the use of a generic SMT-Lib wrapper. We fully support incrementality and backtracking as well as solving under assumptions. For these logics we provide UNSAT-core and Craig interpolants extraction when using the MathSAT or Z3 solvers. Finally, quantifier elimination (i.e. computation of an equivalent quantifier-free formula given a quantified formula) for the $\mathcal{LIA}$ and the $\mathcal{LRA}$ theories is provided through ad-hoc capabilities of MathSAT and Z3. We highlight that this feature is not part of the SMT-Lib specification: while it is possible to create algorithms based on the SMT-Lib capabilities that compute

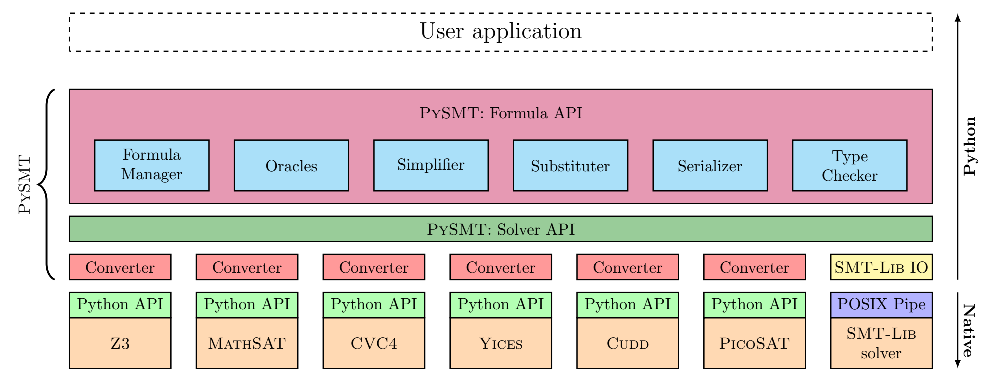

Figure 1: Overview of the PySMT architecture. Solvers (including SMT-Lib one) are abstracted through a common API and a solver-specific converter.

the quantifier elimination in some very special cases, in general this problem requires ad-hoc algorithms.

The library is developed using the Python programming language. This choice has been driven by two key factors. First, Python is a good language for fast-prototyping of ideas. Second, Python has recently been gaining a lot of attention in the scientific community, especially when dealing with data analysis. In fact, Python provides a huge eco-system of widely used libraries: PySMT has been designed to naturally fit in the Python eco-system to be integrated with other libraries. With this work, we hope to lower the entry-barrier for people interested in SMT-based reasoning.

As shown in Figure 1, solving an SMT formula in PySMT requires passing through three software layers. The first layer is the Formula API that deals with formula creation and manipulation. Here we find common services, such as type-checking, serialization, simplification and substitution. All these services are implemented on top of the `Walker` class, that provide an extensible implementation of the visitor pattern to navigate and operate on a formula, by taking care of iterative navigation and memoization. Once a formula is created, we can pass it to the *solver* layer, to perform satisfiability checking, model construction, etc. All the logic for interacting with the solver API (or pipe in the SMT-Lib case) is handled by the library. Thanks to the SMT-Lib initiative, we are able to provide a common set of features across solvers. In order to interact with the solver, however, we need to convert PySMT expressions into the internal representation of the target solver. This is where the *converter* layer comes into play. Since this layer is implemented separately from the solver interface, we are also able to use it to access features that are exposed by the solver API but that are not wrapped by PySMT. For example, the all-SMT computation function in MathSAT API, can be called directly through the Python API, using the converter to create the internal MathSAT representation of an expression, and reconstructing the PySMT version of the result.

Apart from pure SMT solving, the library offers a number of additional services that simplify the development of SMT-based algorithms, for example: printing in multiple formats, type-checking, substitution, infix notation, constants simplifications, free-variables computation, expression size evaluation with multiple metrics, and an interactive shell environment. Particularly interesting is the ability of PySMT to detect the logic used in a formula; this is used to automatically select a solver that supports the given logic during solving. In fact, the library does not require any solver to be installed to work. However, if multiple solvers are installed, the library knows which ones can be used to solve certain logics.

All these features have been implemented in a thread-safe way, and all expressions are serializable in a binary format that can be used to cache intermediate results (this is done using Python's built-in library called `pickle`). This makes it possible to use the `multiprocessing` Python module for out-of-the-box parallel processing.

PySMT is publicly available under the terms of the open-source APACHE 2 license at `https://github.com/pysmt/pysmt` and also through the standard Python Package Index (PyPI)[1]. PySMT works with both Python 2 and 3, although currently only the MathSAT and PicoSAT APIs work on Python 3. Finally, in the distribution we provide several examples of usage, some introductory documentation, and hundreds of tests that can be used to get a better understanding of the inner workings of the library.

2.1 Case-studies

PySMT has already been used in two different research areas: scheduling and safety assessment.

PySMT formulae representation and manipulation capabilities are exploited in [18] to produce the encoding of a particular type of scheduling problem called *Strong Controllability of Temporal Networks*. Thanks to PySMT, we could easily implement an SMT translation for the problem at hand despite its inherent complexity and the different flavors of the encoding. Moreover, we exploited the quantifier elimination capabilities of PySMT to produce efficient quantifier-free encodings. A similar application has been presented for another scheduling problem in [17]: PySMT is employed to provide the SMT capabilities needed for the different algorithms analyzed. In this work we exploited several features of PySMT to manipulate the formulae in the presented algorithms and we also implemented an optimization procedure on top of PySMT, combining SMT-solving with the Python `sympy` library for symbolic calculus.

Another domain in which we used PySMT is safety assessment by means of *Timed Failure Propagation Graphs* (TFPG) [8]. TFPGs are a formalism used to describe the propagation of faults in a system. The challenge we faced was to provide an encoding that would allow the user to validate the design of a TFPG. Although at the beginning of our research we considered only solving of SMT formulae, we then realized that one of the tasks our users wanted to perform, required quantifier elimination (see *Refinement* in [8]), that was however not supported by our solver. However, thanks to PySMT we were able to perform this reasoning task using Z3, by only changing one line of code. Moreover, we managed to leverage the huge ecosystem of Python libraries in order to perform random graphs generation (using the package `networkx`) to perform stress testing of our encoding.

Both examples presented here had a focus on finding a suitable way of solving the problem, thus importance was placed on the ability to play and explore the encodings rather than on optimal performances. Once we were satisfied with the algorithm designed, we realized that the overhead provided by PySMT was negligible, since most of the time was spent inside the SMT solver. For example, in the scheduling case study 90% of the execution time was due to the solver search.

This is the best situation in which to use PySMT. In other situations, where the solving is lighter and the controlling algorithm is more complex, there might be a case for pushing the implementation closer to the solver, and maybe in a compiled language. We strongly believe, however, that this should be done as a second step, when the design choices have been fixed, conceptual bugs addressed and profiling on a representative benchmark performed. Thus, we envisage a situation in which PySMT is used for prototyping of algorithms and only afterwards

[1]On Linux, PySMT can be installed with the command: `pip install pysmt`.

(if needed) these algorithms are re-implemented in a solver-specific way or using a low-level library.

3 Examples of Usage

```
1  from pysmt.shortcuts import *
2  from pysmt.typing import INT
3
4  # Infix-Notation is disabled by default
5  get_env().enable_infix_notation = True
6
7  hello = [Symbol(s, INT) for s in "hello"]
8  world = [Symbol(s, INT) for s in "world"]
9  letters = set(hello+world)
10 domains = And([((1 <= l) & (10 >= l)) for l in letters])
11
12 sum_hello = Plus(hello) # n-ary operators can take lists
13 sum_world = Plus(world) # as arguments
14 problem = sum_hello.Equals(sum_world) & sum_hello.Equals(25)
15 formula = And(domains, problem)
16
17 print("Human-readable version of the formula: " + str(formula))
18
19 model = get_model(formula)
20 if model:  print(model)
21 else:      print("No solution found")
```

Figure 2: Hello World Puzzle Example

In this section, we provide an introduction to PySMT main features by means of a few examples. These and additional examples are available in the project repository.

Hello World Puzzle We start with a simple puzzle (Figure 2): match a digit to each letter of the words *Hello* and *World* so that the sum of the letters in each word is equal to 25.

$$H + E + L + L + O = W + O + R + L + D = 25$$

After importing PySMT (L1-2) we enable infix notation of the operators (L5). Infix notation is disabled by default, due to the ambiguity of the precedence of some operators used to represent theory relations. Nevertheless, to make the example shorter we enable it here. From this line you can see that by default PySMT features a global environment that can be configured. Lines 7-8 create an SMT integer variable for each letter in the words *hello* and *world*. Two variables having the same name yield identical objects, hence only one variable named "o" is created. Line 9 creates a set of all the created SMT variables: {`d,e,h,l,o,r,w`} all of integer type. Line 10 defines the domain of each variable. Notice the infix operators: the logical theory operator *less-than-or-equal-to* creates a Boolean expression on which the conjunction operator is called. Line 12-15 provide the main constraints of our problem. In PySMT all n-ary operators can take an iterable (such as lists or generators) as input, but also an arbitrary number of arguments is supported. The formula is then printed in a human readable format (L18), for inspection. Line 20-24 provide the actual solving and handling of the result. The package `pysmt.shortcuts` defines several functions similar to `get_model`. These functions provide one-liners for common operations such as getting a model from a formula. Under the hood, `get_model` performs several operations. First, it detects the theories used in `formula` and selects the SMT-Lib logic needed to solve this formula. It then checks if any of the solvers currently installed in the system supports the given logic. If so, a solver is selected according to a preference list. The formula is then solved with the selected solver and, if satisfiable, a model is extracted. The solver is then disposed and a solver agnostic model object is returned. If the formula is not satisfiable `None` is returned. We highlight that, alternatively, it is possible to specify the desired solver name as an additional parameter to `get_model`. This allows the evaluation of the performances for the problem at hand on different engines.

```python
def formula_at_time(vars, f, t):
    return f.substitute({v : var_at_time(v, t) for v in vars}) \
            .substitute({var_next(v) : var_at_time(v, t+1) for v in vars})

def inc_bmc(vars, init, trans, prop, k, logic=QF_BOOL):
    with Solver(logic=logic) as s:
        s.add_assertion(formula_at_time(vars, init, 0))
        for t in xrange(k):
            not_p_t = formula_at_time(vars, Not(prop), t)
            res = s.solve(assumptions=[not_p_t])
            if res:
                return "Bug found at depth
            s.add_assertion(formula_at_time(vars, trans, t))
    return "Safe up to
```

Figure 3: BMC Example

Bounded Model Checking A textbook example of using SAT/SMT technology is Bounded Model Checking (BMC) [6]. Although conceptually simple, an implementation of BMC usually requires dealing with substitution of terms. Substitution is a common operation when performing encodings, therefore PySMT provides a simple way to perform it. Given a transition system expressed as a set of variables, an initial relation $I(x)$ and a transition relation $T(x, x')$, we perform the unrolling of the transition relation for k steps, and check if any states in the unrolling intersects a state in which a given property p is violated. Formally, for a given k we check whether the following is satisfiable:

$$I(0) \wedge \bigwedge_{i=0,k-1} T(i, i+1) \wedge \bigvee_{i=0,k} \neg p(i)$$

This check can be done incrementally: we keep the initial relation (instantiated at time 0) and the unrolling of the transition relation up to t as assertions in the solver, and at each step we check if the formula conjoined with the negated property at time $t+1$ is satisfiable. If it is, we have found a counterexample (that is encoded in the model of the formula), otherwise we extend by one step the unrolling of the transition relation asserted in the solver. We continue until we reach the bound k, thus showing that the property is satisfied for at least k steps. To perform the unrolling, we use an auxiliary function `var_at_time` (not in Figure 3) that takes a symbol and a time i and returns the matching symbol (e.g., x^i for x). The auxiliary function `formula_at_time` instantiates the given formula `f` at time `t`, substituting each variable with its matching symbol at time t and each next variable with its matching symbol at time $t+1$. we use the `substitute` method, that takes a map from terms to be replaced to terms to be used as replacements. The core function is `bmc_inc` that takes the initial states `init`, transition

```python
def efsmt(y, phi, logic=AUTO, maxloops=None, ename=None, fname=None):
    # Solves exists x. forall y. phi(x, y) returning a model for x
    # if SAT, otherwise None
    y = set(y)
    x = phi.get_free_variables() - y

    with Solver(logic=logic, name=ename) as esolver:
        esolver.add_assertion(Bool(True))
        loops = 0
        while maxloops is None or loops <= maxloops:
            loops += 1

            eres = esolver.solve()
            if not eres:
                return False
            else:
                tau = {v: esolver.get_value(v) for v in x}
                sub_phi = phi.substitute(tau).simplify()
                model = get_model(Not(sub_phi),
                                  logic=logic, solver_name=fname)
                if model is None:
                   return tau
                else:
                   sigma = {v: model.get_value(v) for v in y}
                   sub_phi = phi.substitute(sigma).simplify()
                   esolver.add_assertion(sub_phi)
        raise SolverReturnedUnknownResultError
```

Figure 4: EF-SMT Example

relation `trans` and an invariant property to be checked `prop`, as well as the maximum depth `k`. In Line 6, we ask PySMT to give us a `Solver` for solving the logic specified in input. Note the `with` statement: PySMT takes care of constructing the solver object and the resources are automatically released when leaving the scope of the `with` statement. We assert the initial formula at time 0 in Line 7. The main loop of the algorithm iterates to all times from 0 to $k-1$ (Line 8). At each step, we check if the negated property is reachable at time t (Lines 9-10) by using solving under assumptions capabilities of PySMT: we specify a set of formulae to be asserted in the solver, solved and then retraced. If the formula conjoined with the assumptions is satisfiable, it means that we have found a path from the initial states to a state that violates the invariant property, otherwise we can go on extending the paths by asserting another step of the transition relation (Line 14). Note that in this example we do not make any assumption on the type of the variables that we are using. The same algorithm works for pure Boolean expressions, or for any theory supported by the library, thus we allow the user to specify the correct logic to be used, i.e., by using the `get_logic` function to compute the logic of a given formula. In this way we can use a SAT-solver, when we are in the purely Boolean case.

EF-SMT The previous two examples were focused on expressions manipulation, with limited interaction with the solver. We now show an algorithm that requires the use of multiple instances of a solver, and show how we can seamlessly combine different solvers to solve it. The algorithm has been presented in [14], and it is an approach to solve quantified SMT expressions of the form:

$$\exists \vec{x}.\ \forall \vec{y}.\ \varphi(x, y)$$

without performing explicit quantifier elimination. This type of problem appears in several contexts and we refer to [14] for an in-depth discussion. The idea of the algorithm is to alternate an existential solving phase, with an universal solving phase. Two solvers are instantiated: the existential and the universal. The existential picks a model for φ. The model obtained is used to fix the value of the $\vec{x}$ variables in φ, and a new formula, with only $\vec{y}$ free variables, is given to the universal solver, that tries to find a counter-example. If a counter-example is found, it is used to build an additional constraint for the existential solver, that then starts looking for a new model. The algorithm terminates when one of the solvers is not able to find models anymore. In particular, the formula is unsatisfiable if the existential solver cannot find a model, while it is satisfiable when the universal solver fails to find a counter-example. The code in Figure 4 implements the above algorithm.

By design, we require the user to specify only the $\vec{y}$ variables, and we compute the $\vec{x}$ variables by complement of the free-variables of φ (Line 5-6). We then create a context for the existential solver (L8). A context is a Python construct that is used to take care of initialization and destruction of objects. `esolver` is defined within the context, but whenever we leave the context, through a nominal or exceptional flow, the library will take care of destroying the solver and freeing associated resources. Note that the `Solver` constructor takes a logic and a solver name as parameters. Indeed, in this example we can experiment with different solvers for the existential and universal part, e.g., MathSAT could play the role of existential solver, while Z3 could play the role of universal solver. Line 14 performs the solving for the existential solver, and we extract a model (if one exists) in L17-18. We then call the universal solver on the problem with the partial model over the $\vec{x}$ variables. If a counter-example is found, we create a constraint to generalize the counter-example and assert it in the existential solver (L24-26). We proceed until either one of the solvers is not able to find models, or we reached a maximum number of iterations. This example shows a more fine-grained interaction with the underlying solver exploiting incrementality, and explains how different solvers can be used simultaneously in the same program.

4 Related Work

The need of providing a simple and uniform access across SMT solvers has been demonstrated first by the SMT-Lib initiative, and then by a series of libraries similar to PySMT. PySMT is unique, however, in the fact that it provides services to easily perform common operations, and a solver-agnostic API that allows developers to use multiple solvers in the same code-base, without the need of learning ad-hoc APIs.

Some solvers, e.g. Z3 and MathSAT, offer a Python API to allow high-level interaction with the solver. In particular, Z3 has a very sophisticated and powerful API that strongly resembles some of the primitives that PySMT offers. These APIs are limited by the fact that they are solver specific, and are often too close to the C API of the package, sometimes leaving the implementation of common procedures, such as substitution or computation of the free variables of a formula, to the users.

Apart for the SMT-Lib initiative itself, few projects allow the use of multiple solvers in a coherent way. Some libraries, such as jSMTLib [20] for Java, Hsmtlib [28] for Haskell, Scala SMT-LIB [7] and Assumptio [13] for Scala, and NSolv [30] for (multi-processing) C++, provide a programmatic access to SMT-Lib features, using solvers as back-ends by communicating SMT-Lib commands via pipe. This approach has some limitations: first, it is limited to the standard features offered by SMT-Lib, leaving out useful features offered by several solvers such as quantifier elimination; second, the communication via pipe is typically slower than direct solver integration, because it requires the serialization and de-serialization of possibly large formulae.

The work closest to ours is MetaSMT [25], a C++ library that allows the use of different solvers with a unifying C++ API. A first difference comes form the set of supported logics: PySMT supports $\mathcal{BV}$ and $\mathcal{UFLIRA}$ while MetaSMT is concerned with $\mathcal{QF\_ABV}$. MetaSMT is an efficient library that provides low-level solver-agnostic interface to a few SMT solver, but does not provide high-level services for expression manipulation and fast prototyping. Clearly, this comes with a performance trade-off: the native C++ linking and the use of templates can provide benefits when implementing algorithms with performance requirements.

SMT-KIT [27] is a library similar to MetaSMT providing solver-agnostic access to CVC4, MathSAT and Z3. Although, SMT-KIT supports more logics than MetaSMT, it is still limited to the quantifier free fragments. Moreover, neither MetaSMT nor SMT-KIT provide a solver-agnostic representation of a model, thus relying always on the availability of a solver.

PySMT tries to get the best out of both types of approaches. Thanks to our generic SMT-Lib wrapper, we can use any SMT-Lib compliant solver. However, for some solvers, we have native Python bindings that overcome the limitations of the pure SMT-Lib interface.

5 Conclusions

In this paper we presented PySMT, a Python library that simplifies and abstracts the use of SMT solvers, allowing fast and easy prototyping of algorithms using the SMT technology. PySMT is a production-ready, open-source project and we would like to invite anyone to try it and contribute. Several directions are possible for the future development of PySMT. We would like to extend the set of supported logics to include the theory of arrays and non-linear arithmetic. This would provide a good opportunity to start experimenting with different paradigms of solvers: exact solvers (e.g., Z3) and approximation solvers (e.g., dReal [23]). Finally, there are several more solvers that we would like to integrate through their native APIs, for example, Boolector [9] and OpenSMT [10].

References

[1] Clark Barrett, ChristopherL. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification*, volume 6806, pages 171–177. 2011.

[2] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, pages 1–35, 2012.

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

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT Workshop*, 2010.

[5] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[7] Regis Blanc. Scala SMT-LIB. `https://github.com/regb/scala-smtlib`, 2015.

[8] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Andrea Micheli. Smt-based validation of timed failure propagation graphs. In *AAAI*, 2015.

[9] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, pages 174–177. 2009.

[10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *TACAS*, pages 150–153. 2010.

[11] Jrme Cantenot, Fabrice Ambert, and Fabrice Bouquet. Test generation with satisfiability modulo theories solvers in model-bas ed testing. *Software Testing, Verification and Reliability*, 24(7):499–531, 2014.

[12] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *CAV*, pages 334–342, 2014.

[13] Adrien Champion and Pierre-Loc Garoche. Assumptio: Actor-based Smtlib-2 Scala Unified Malleable Package for real-Time Interaction On-demand. `https://cavale.enseeiht.fr/redmine/projects/assumptio`, 2013.

[14] Chih-Hong Cheng, Natarajan Shankar, Harald Ruess, and Saddek Bensalem. EFSMT: A logical framework for cyber-physical systems. *CoRR*, abs/1306.3456, 2013.

[15] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *TACAS*, 2014.

[16] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, pages 93–107, 2013.

[17] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. An smt-based approach to weak controllability for disjunctive temporal problems with uncertainty. *Artificial Intelligence*, 224(0):1–27, 2015.

[18] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Solving strong controllability of temporal problems with uncertainty using SMT. *Constraints*, 20(1):1–29, 2015.

[19] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Micha Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, volume 5674, pages 23–42. 2009.

[20] David R. Cok. The jSMTLIB User Guide. `http://www.grammatech.com/resource/smt/jSMTLIBUserGuide.pdf`, 2013.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[22] Bruno Dutertre. Yices 2.2. In *CAV*, pages 737–744, 2014.

[23] Sicun Gao, Soonho Kong, and Edmund M Clarke. dreal: An smt solver for nonlinear theories over the reals. In *Automated Deduction–CADE-24*, pages 208–214. 2013.

[24] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. HSF(C): A software verifier based on horn clauses - (competition contribution). In *TACAS*, pages 549–551, 2012.

[25] Finn Haedicke, Stefan Frehse, Grschwin Fey, Daniel Groe, and Rolf Drechsler. metaSMT: Focus on your application not on solver integration. In *DIFTS*, 2012.

[26] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.

[27] Alex Horn. SMT-Kit. `https://github.com/ahorn/smt-kit`, 2015.

[28] Nuno Laranjo and Rogerio Pontes. Hsmtlib. `https://github.com/MfesGA/Hsmtlib`, 2015.

[29] M. Leino and K. Rustan. Automating theorem proving with smt. In *Interactive Theorem Proving*, volume 7998, pages 2–16. 2013.

[30] Dan Liew. Nsolv. `https://github.com/delcypher/nsolv`, 2012.

[31] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, volume 4144, pages 419–423. 2006.

[32] Fabio Somenzi. Efficient manipulation of decision diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):171–181, 2001.

[33] Nikolai Tillmann and Jonathan de Halleux. Pex - white box test generation for .net. In *TAP*,

volume 4966, pages 134–153, 2008.

[34] Cesare Tinelli. Smt-based model checking. In *NFM*, 2012.