`Smt-Switch`: a solver-agnostic C++ API for SMT solving
(Extended Abstract)

Makai Mann[1], Amalee Wilson[1], Cesare Tinelli[2], and Clark Barrett[1]

[1] Stanford University
Stanford, USA
[2] The University of Iowa
Iowa City, USA

Abstract

This extended abstract describes work in progress on `Smt-Switch`, an open-source, solver-agnostic API for SMT solving. `Smt-Switch` provides an abstract interface, which can be implemented by different SMT solvers. `Smt-Switch` provides simple, uniform, and high-performance access to SMT solving for applications in areas such as automated reasoning, planning, and formal verification. The interface allows the user to create, traverse, and manipulate terms, as well as to dynamically dispatch queries to different underlying SMT solvers.

1 Introduction

`Smt-Switch` is an open-source, solver-agnostic API for interacting with various SMT solvers in C++. While SMT-LIB [2] provides a standard textual interface for SMT solving, there are limitations. In particular, applications that interact closely with a solver or its expressions could benefit from an API. For example, tools that perform specialized term rewriting or dynamically adjust queries based on satisfiability results. The two most common approaches used by such tools now is to either build the tool on a specific SMT solver's API, or write a custom expression representation which is then translated to SMT-LIB and communicated to arbitrary SMT solvers through pipes as needed. `Smt-Switch` provides a convenient, in-memory representation that hides the underlying SMT solver from the user. This allows projects to easily swap out underlying SMT solvers, without developing a custom expression representation.

There already exists a well-established, solver-agnostic Python API: PySMT [8]. While this tool is excellent for prototyping, many performant tools are not written in Python due to the overhead of an interpreted language. Furthermore, PySMT keeps its own representation of terms which is translated to underlying SMT solvers on demand. While this is extremely convenient, it adds memory overhead and makes adding new theories more involved. `Smt-Switch` is a light wrapper on the underlying SMT solvers. This helps keep the memory overhead low, and the maintenance of the tool straightforward. Adding support for a new SMT solver to `Smt-Switch` requires implementing an abstract interface using the new solver's C or C++ API. The current version has implementations for Boolector[14], CVC4[1], MathSAT[4], and Yices2[6]. It supports quantifier-free formulas over booleans, bitvectors, integers, reals, uninterpreted functions and arrays. The API can easily be extended with new sorts and features supported by the underlying solvers. The tool is available at https://github.com/makaimann/smt-switch.

2   Design

`Smt-Switch` is designed to be a lightweight wrapper on the C or C++ API of the underlying SMT solvers. It intentionally delegates as much of the functionality to the underlying solvers as possible. This reduces redundancy, and results in simpler implementations and lower memory overhead. The API is implemented in C++11 and also provides Python bindings using Cython [3]. The Python bindings can be used directly in a Python project, but the intended usage is for tools that use `Smt-Switch` in C++, but want to expose `Smt-Switch` functionality in Cython-generated Python bindings for that tool.

Architecture. `Smt-Switch` provides abstract classes which must be implemented by a wrapper for each underlying solver. The implementations satisfy the interface requirements of the abstract classes and wrap the relevant objects needed for that underlying solver's own API. Throughout this abstract, we use *underlying* solver to refer to the SMT solver we are wrapping and *backend* to refer to the `Smt-Switch` wrapper. Any solver with a C or C++ API can be added to `Smt-Switch`. At the `Smt-Switch` API level, the user interacts with smart pointers to the abstract classes. The `virtual` method functionality of C++ allows the pointer to dynamically call the relevant method of the derived class (the backend) that is pointed to. This architecture allows the interface to be agnostic to the underlying solver. The three primary abstract classes are: (i) `AbsSort`; (ii) `AbsTerm`; and (iii) `AbsSmtSolver`. The interface and method names are based on SMT-LIB version 2 [2]. Figure 1 shows a representative selection of the `AbsSmtSolver` header file which details the abstract interface. Many methods were removed for space. The `Op` class is not abstract and does not need to be implemented by the backend. However, the backend's `AbsSmtSolver` implementation must interpret an `Op` when building terms.

Building and Linking. `Smt-Switch` uses CMake [11]. The build infrastructure is designed to be modular with respect to backend solvers. By default, the configuration script does not add any solvers. `Smt-Switch` will build but has no functionality. To add a solver, the user must first obtain the underlying solver's library. `Smt-Switch` provides convenience scripts for obtaining several of the underlying solvers and instructions for the rest. Then, a command line flag to the `Smt-Switch` configuration script will add the solver to the build. Each solver backend results in a separate library, e.g. on Linux there will be a `libsmt-switch.so` file as well as a `libsmt-switch-cvc4.so` file if configured with `--cvc4`. This allows the user to build `Smt-Switch` once, but only link solver backends to their project as needed. The configuration script also has options to enable static and debug builds.

Testing. We use *GoogleTest* [9] for the C++ test infrastructure and *Pytest* [12] for the Python test infrastructure. Tests are parameterized by solver so that one test can easily be run over all solvers.

Undefined Behavior. Each backend implementation of a solver needs to recover its relevant objects from a generic abstract object. This is done with a `static_pointer_cast`. This results in *undefined behavior* if the cast is not valid. In particular, this means that sharing `Sorts` and `Terms` between different `SmtSolvers` results in undefined behavior. To move a `Sort` or `Term` between solvers, it must explicitly be transferred. There is a class provided in the API which can transfer terms between solvers. `Smt-Switch` is intentionally lightweight and thus does not perform much error checking. This design decision was made to reduce overhead and redundancy, since most SMT solvers do error checking already. However, the class hierarchy allows users to extend classes or build wrappers if they would like to insert their own error checking at the `Smt-Switch` level.

```
1   /** Abstract solver class to be implemented by each supported solver. */
2   class AbsSmtSolver
3   {
4    public:
5     /* Add an assertion to the solver
6      * SMTLIB: (assert <t>)
7      * @param t a boolean term to assert */
8     virtual void assert_formula(const Term & t) = 0;
9     /* Check satisfiability of the current assertions
10     * SMTLIB: (check-sat)
11     * @return a result object - see result.h */
12    virtual Result check_sat() = 0;
13    /* Get the value of a term after check_sat returns a satisfiable result
14     * SMTLIB: (get-value (<t>))
15     * @param t the term to get the value of
16     * @return a value term */
17    virtual Term get_value(const Term & t) const = 0;
18    /* Create a sort
19     * @param sk the SortKind (BOOL, INT, REAL)
20     * @return a Sort object */
21    virtual Sort make_sort(const SortKind sk) const = 0;
22    /* Create a sort
23     * @param sk the SortKind (BV)
24     * @param size (e.g. bitvector width for BV SortKind)
25     * @return a Sort object */
26    virtual Sort make_sort(const SortKind sk, uint64_t size) const = 0;
27    /* Create a sort
28     * @param sk the SortKind (FUNCTION)
29     * @param sorts a vector of sorts (last sort is return type)
30     * @return a Sort object
31     * Note: This is the only way to make a function sort */
32    virtual Sort make_sort(const SortKind sk, const SortVec & sorts) const = 0;
33    /* Make a boolean value term
34     * @param b boolean value
35     * @return a value term with Sort BOOL and value b */
36    virtual Term make_term(bool b) const = 0;
37    /* Make a symbolic constant or function term
38     * SMTLIB: (declare-fun <name> (s1 ... sn) s) where sort = s1x...xsn -> s
39     * @param name the name of constant or function
40     * @param sort the sort of this constant or function
41     * @return the symbolic constant or function term */
42    virtual Term make_symbol(const std::string name, const Sort & sort) = 0;
43    /* Make a new term
44     * @param op the operator to use
45     * @param terms vector of children
46     * @return the created term */
47    virtual Term make_term(const Op op, const TermVec & terms) const = 0;
48  };
```

50   Figure 1: Representative set of methods in the AbsSmtSolver interface.

Custom Exceptions. `Smt-Switch` defines its own set of exceptions that inherit from `std::exception`. Each of them take a `std::string` message for describing the problem that can be accessed with the `what` method.

1. `SmtException` : the generic `Smt-Switch` exception - all other custom exceptions inherit from it.

2. `NotImplementedException` : the exception for an unimplemented feature in a backend solver.

3. `IncorrectUsageException` : an exception that is thrown when incorrect usage on the users' part is detected.

4. `InternalSolverException` : an exception that is thrown when there is an error in the underlying solver.

3   Abstract Interface

We now describe the abstract interface in more detail. We start by describing the `AbsSort` class which illustrates the supported theories. This is followed by the non-abstract struct for representating operators: `Op`. Next, we describe the other two important abstract classes: `AbsTerm` and `AbsSmtSolver`. Figure 2 depicts the class hierarchy for an `AbsSmtSolver`. The `AbsSort` and `AbsTerm` classes have analogous architectures. Finally, we describe the `Result` struct which is returned after a satisfiability query, and the solver factories that are used to instantiate solvers.

3.1   AbsSort

The `AbsSort` abstract class represents the type of `Terms` created in `Smt-Switch`. Sorts are characterized by an `enum`, `SortKind`, which categorizes it and additional parameters based on the `SortKind`. `AbsSort` has `virtual` methods for querying the sort for its `SortKind` and parameters. This abstract class is implemented by each backend solver and wraps the backend's representation of a sort. For example, the CVC4 backend has a `CVC4Sort` class which wraps a sort object from CVC4's C++ API. A `Sort` is a pointer to an `AbsSort`. Currently supported `SortKinds` and associated parameters are the following:

1. `BOOL`: booleans

2. `INT`: integer numbers

3. `REAL`: real numbers

4. `BV`: fixed-width bitvectors

    (a) parameter: positive integer width

5. `FUNCTION`: uninterpreted function sort

    (a) parameter: vector of domain `Sorts`
    (b) parameter: the codomain `Sort`

6. `ARRAY`: arrays parameterized by index and element sorts

(a) parameter : index `Sort`

(b) parameter : element `Sort`

7. `UNINTERPRETED`: an uninterpreted sort

(a) parameter: non-negative integer arity

3.2 Op

`Op` is a struct used to represent builtin operators for constructing terms in `Smt-Switch`. These can be split into two categories: primitive operators and indexed operators. For unity of representation, an `Op` is used for both. An `Op` stores a `PrimOp` enum and up to two integer indices. Primitive operators are defined only by the `PrimOp` and take no indices. Indexed operators have both a `PrimOp` and one or two indices. For convenience, the `Op` constructor can be applied implicitly by the compiler. Thus, building terms with a primitive operator can be accomplished by using a `PrimOp` directly, without explicitly building an `Op` from it. `Smt-Switch` uses a straightforward naming scheme such that the `PrimOp` enums have one-to-one correspondence with SMT-LIB operators.

Ops can be null. All `Terms` that are a symbol or value have a null operator. Note that terms with null operators are not necessarily leaf nodes. For example, one such edge case is constant arrays (arrays where every element has been set to a given value). Constant arrays are themselves values and thus have a null operator. However, they still have one child which is the value assigned to every element.

3.3 AbsTerm

The `AbsTerm` abstract class represents expressions built through the API. Constructing the term uses methods from the `SmtSolver` because most underlying solvers use a solver or context object for registering terms. However, once the term is created, the `Smt-Switch` interface assumes that it can be queried for information. For example, a term has `virtual` methods for accessing the `Sort`, the `Op` used to create it, and the children of the `Term`. Querying the children is implemented as an iterator. Thus, `Terms` have a `begin` and `end` method that each return a `TermIter`, which is a class that wraps an abstract class, `TermIterBase`. Each backend solver must implement a derived class for `AbsTerm` that wraps its relevant expression representation(s). If the backend solver supports accessing the children of a term, it must also implement a derived class for `TermIterBase`. For example, the CVC4 backend has both a `CVC4Term` and `CVC4TermIter` object that implement the `Term` and `TermIterBase` interfaces and store the relevant term and term iteration objects from the CVC4 C++ API. A `Term` is a pointer to an `AbsTerm`. `Smt-Switch` also has type declarations for convenient data structures such as `TermVec` which is a vector of `Terms`.

A given `Term` can be a symbol (uninterpreted constant or function), a value (theory values such as the number 1), or an expression built from symbols and values using operators. *All* SMT expressions in `Smt-Switch` are represented as terms. In particular, we take a higher-order logic perspective and represent uninterpreted functions as terms. Thus, applying an uninterpreted function uses an "`Apply`" operator on the function and the arguments. Some SMT solvers are starting to add higher-order logic features, and this representation is often convenient. For example, an invariant maintained by `Smt-Switch` is that any term with a non-null operator can always be rebuilt using its operator and children. For a given `Term` t

with a non-null operator, the backend solver implementation should guarantee that the following always returns `true`:

```
t == solver->make_term(t->get_op(), TermVec(t->begin(), t->end())))
```

If the API instead had different methods for applying an uninterpreted function versus creating terms with a builtin operator, this invariant would not be maintained.

3.4 AbsSmtSolver

`AbsSmtSolver` declares the main interface that a user interacts with. It has methods for declaring `Sorts`, building `Terms`, asserting formulas and checking for satisfiability. `SmtSolver` is a pointer to an `AbsSmtSolver`. The bulk of the implementation for a backend solver will likely be its implementation of `AbsSmtSolver`. The interface methods closely match the commands of SMT-LIB. The naming scheme simply replaces "-" with "_". For example, some methods of `AbsSmtSolver` include `set_opt`, `check_sat`, and `get_value`. One notable exception is that assertions are added with the `assert_formula` method (as opposed to "assert" as in SMT-LIB), to avoid clashing with the C `assert` macro.

Each backend solver must implement a derived class for `AbsSmtSolver`. For example, the CVC4 backend has a `CVC4Solver` object that inherits `AbsSmtSolver`. This derived class must store the relevant information for that solver and implement each of the `virtual` `AbsSmtSolver` methods in the underlying solver's API. The `AbsSmtSolver` methods all operate over pointers to the abstract objects as shown in Figure 1. For example, the method at line 47 has signature: `make_term(Op, const TermVec &)`. The derived class implementation must cast each of the `Term` pointers in the `TermVec` to its own derived class implementation of `AbsTerm` to be able to access relevant wrapped members. Thus, the `CVC4Solver` implementation of that method would cast each of the `Terms` to a `CVC4Term` with `static_pointer_cast<CVC4Term>(t)` for `Term t`. Then, it would intepret the `Op`, perform the corresponding operation over the CVC4 objects, and finally wrap the result in a `CVC4Term` and return it as a `Term`.

3.5 Result

`Result` is a struct for representing the return value of a call to `check_sat` or `check_sat_assuming`. It currently has three possible values: `SAT`, `UNSAT`, and `UNKNOWN`. In the case of `UNKNOWN`, the backend solver can optionally provide a `std::string` with an explanation of the reason for the unknown result.

3.6 Solver Factories

A solver factory is a simple class that defines a single static method: `create(bool logging)`. Each backend solver implementation defines a factory and has a dedicated header file. The `create` function is used to create a `SmtSolver` of that type. The single boolean parameter is to choose whether or not the term DAG is tracked at the `Smt-Switch` level.

If `logging` is set to `false`, the `SmtSolver` relies on the underlying solver API for querying a term for its `Op`, `Sort` and children (e.g. term traversal) by translating back to `Smt-Switch` objects.

If `logging` is set to `true`, the method returns the backend solver but wrapped by a `LoggingSolver`. This is an implementation of `AbsSmtSolver` that forwards commands to
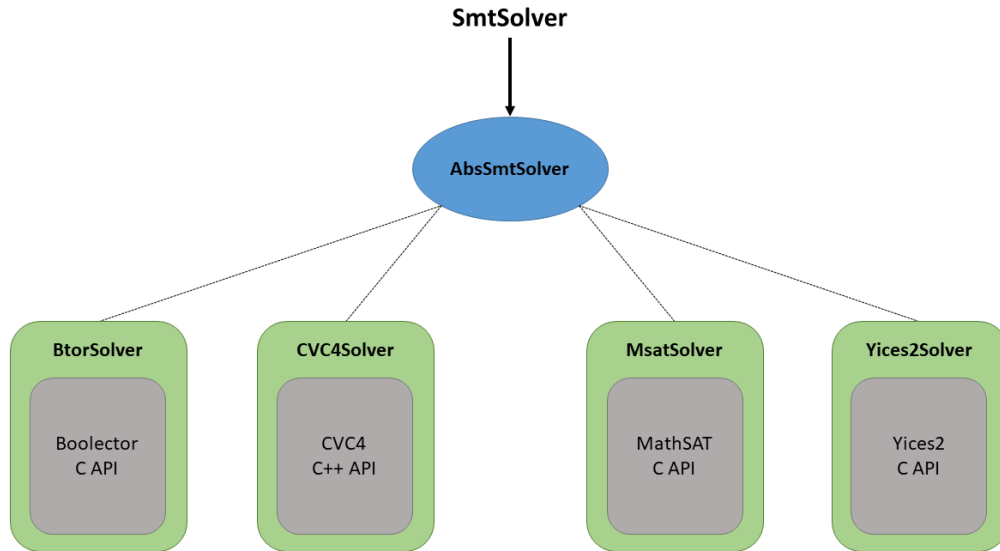
Figure 2: `AbsSmtSolver` Hierarchy. `SmtSolver` is a `typedef` for a pointer to an `AbsSmtSolver`. The dashed lines represent inheritance.

another backend `SmtSolver`. Additionally, it keeps a term DAG at the `Smt-Switch` level. This is accomplished by wrapping every `Sort` and `Term` created through the `LoggingSolver` in a `LoggingSort` and `LoggingTerm`, respectively. In addition to storing the underlying objects, these classes also keep relevant data for tracking the expression DAG as it was constructed. For example, a `LoggingSort` keeps the `SortKind` and parameters used to create it. A `LoggingTerm` keeps the `Op`, (Logging) `Sort`, and children (Logging) `Terms`. This optional feature can be useful for underlying solvers that perform on-the-fly rewriting or alias sorts (e.g., do not distinguish between bitvectors of width one and booleans). The logging infrastructure ensures that a created term has exactly the same `Op`, `Sort`, and children that were used to create it. The implicit assumption is that even though creating a term through a solver API might result in a rewritten term, creating the term again with the same `Op` and children will result in the same rewritten term. Thus, the logging infrastructure hides the underlying solver's rewriting from the user. To contend with sort aliasing, the `LoggingSolver` also performs sort inference to compute the expected sort when building terms. The logging infrastructure simplifies transferring terms between different solvers (which might not alias sorts in the same way) and can be more intuitive. Additionally, some solver backends (currently the Yices2 backend) rely on the logging infrastructure for term traversal. A Yices2 `SmtSolver` created without logging will create `Terms` that do not support iterating over the children.

4   Examples

In this section we demonstrate the `Smt-Switch` API with a simple example using both the C++ API and the Python bindings. Figure 3 uses `Smt-Switch` with the CVC4 backend to solve two simple queries over bitvectors and uninterpreted functions. Lines 1-2 include the

necessary `Smt-Switch` headers, and lines 3-8 are standard C++ includes, `using` declarations and main function. Line 9 declares a `SmtSolver` using CVC4 as the backend without logging. In line 10 the solver is set to incremental mode. Lines 11 and 12 declare a bitvector sort of width 9 and a function sort from a width 9 bitvector to a width 9 bitvector. Lines 13-17 declare two bitvector symbolic constants, an uninterpreted function and then applies the function to each of the bitvector symbols. Line 20 uses a C assert to check that the operator of an applied uninterpreted function term is `Apply`. Line 21 populates a vector with the children of term `fx`. Line 22 asserts that there are two children (in some solvers this term would have only one child: `x`). Lines 25 and 26 check that the children are the function, `f`, and the bitvector, `x`. Line 29 asserts to the `SmtSolver` that the returned values from the function applied to x and y are different. Lines 30 and 31 extract the bottom 8 bits from `x` and `y`, respectively. Line 33 asserts to the `SmtSolver` that the bottom bits of `x` and `y` are equivalent. Line 35 checks the satisfiability of the current assertions and line 36 checks that the solver found `SAT` as expected. The query is satisfiable because x and y can have different most significant bits, and thus the function applications could return different values. Line 38 asserts to the `SmtSolver` that the most significant bits of `x` and `y` are also equivalent. Checking satisfiability in line 41 and checking the result in line 42 confirms that the current set of assertions are now unsatisfiable because of uninterpreted function axioms. Figure 4 shows exactly the same example but through the Python API.

It is very simple to change the solver in these two examples, assuming that the relevant solver libraries have already been built. In the C++ example this amounts to including the relevant solver factory file (similar to line 2), updating line 9 to use a different solver factory create function, and finally recompiling and relinking with the new solver library. Assuming the Python bindings were built with the relevant solver, changing the solver in the Python example only requires using a different create function in line 5 of Figure 4.

5   License

The `Smt-Switch` code is distributed under the BSD 3-clause license. However, not all solvers have BSD-compatible licenses. For these solvers, the user must obtain the solver libraries themselves and ensure they meet all the requirements for the license. There are then instructions for how to properly build smt-switch with that solver as the backend, in which case the license assumes that of the solver. The BSD-compatible backend solvers are Boolector and CVC4.

6   Related Work

The most closely related work is `smt-kit` [10], another C++ API for SMT solving. This API utilizes templates to be solver agnostic, and has a term representation that is separate from the underlying solver, as opposed to `Smt-Switch` which only provides an abstract interface and a light wrapper around the underlying solvers. This design choice reduces overhead and keeps maintenance simple. `smt-kit` does not appear to be under active development since 2014.

Two other related tools are `PySMT` [8] and `sbv` [7]. `PySMT` is a solver-agnostic SMT solving API for Python. `PySMT` has its own term representation and translates formulas to the backend solvers dynamically once they are asserted. It also uses a class hierarchy to support swapping underlying solvers. `sbv` is a solver-agnostic SMT-based verification tool for Haskell. It provides its own datatypes for representing various SMT queries and communicates with solvers through SMT-LIB with pipes.

```
1   #include "smt-switch/smt.h"
2   #include "smt-switch/cvc4_factory.h"
3   #include "assert.h"
4   #include <iostream>
5   using namespace smt;
6   using namespace std;
7   int main()
8   {
9     SmtSolver s = CVC4SolverFactory::create(false);
10    s->set_opt("incremental", "true");
11    Sort bvsort9 = s->make_sort(BV, 9);
12    Sort funsort = s->make_sort(FUNCTION, {bvsort9, bvsort9});
13    Term x = s->make_symbol("x", bvsort9);
14    Term y = s->make_symbol("y", bvsort9);
15    Term f = s->make_symbol("f", funsort);
16    Term fx = s->make_term(Apply, f, x);
17    Term fy = s->make_term(Apply, f, y);
18
19    // Functions are terms
20    assert(fx->get_op() == Apply);
21    TermVec fx_children(fx->begin(), fx->end());
22    assert(fx_children.size() == 2);
23    // These equalities are structural e.g. the first child *is* f
24    // These are not SMT equalities
25    assert(fx_children[0] == f);
26    assert(fx_children[1] == x);
27
28    // (assert (distinct (f x) (f y)))
29    s->assert_formula(s->make_term(Distinct, fx, fy));
30    Term x_7_0 = s->make_term(Op(Extract, 7, 0), x);
31    Term y_7_0 = s->make_term(Op(Extract, 7, 0), y);
32    // (assert (= ((_ extract 7 0) x) ((_ extract 7 0) y)))
33    s->assert_formula(s->make_term(Equal, x_7_0, y_7_0));
34
35    Result r = s->check_sat();
36    assert(r.is_sat()); // the MSB of x and y can be different
37    // (assert (= ((_ extract 8 8) x) ((_ extract 8 8) y)))
38    s->assert_formula(s->make_term(Equal,
39                                   s->make_term(Op(Extract, 8, 8), x),
40                                   s->make_term(Op(Extract, 8, 8), y)));
41    r = s->check_sat();
42    assert(r.is_unsat());
43    return 0;
44  }
```

Figure 3: C++ API Example

```python
1  import smt_switch as ss
2  from smt_switch.primops import Apply, Distinct, Equal, Extract
3
4  if __name__ == "__main__":
5      s = ss.create_cvc4_solver(False)
6      s.set_opt('incremental', 'true')
7      bvsort9 = s.make_sort(ss.sortkinds.BV, 9)
8      funsort = s.make_sort(ss.sortkinds.FUNCTION, [bvsort9, bvsort9])
9      x = s.make_symbol('x', bvsort9)
10     y = s.make_symbol('y', bvsort9)
11     f = s.make_symbol('f', funsort)
12     fx = s.make_term(Apply, f, x)
13     fy = s.make_term(Apply, f, y)
14
15     assert fx.get_op() == Apply
16     fx_children = [c for c in fx]
17     assert len(fx_children) == 2
18     assert fx_children[0] == f
19     assert fx_children[1] == x
20
21     s.assert_formula(s.make_term(Distinct, fx, fy))
22     x_7_0 = s.make_term(ss.Op(Extract, 7, 0), x)
23     y_7_0 = s.make_term(ss.Op(Extract, 7, 0), y)
24     s.assert_formula(s.make_term(Equal, x_7_0, y_7_0))
25
26     r = s.check_sat()
27     assert r.is_sat()
28     s.assert_formula(s.make_term(Equal,
29                          s.make_term(ss.Op(Extract, 8, 8), x),
30                          s.make_term(ss.Op(Extract, 8, 8), y)))
31     r = s.check_sat()
32     assert r.is_unsat()
```

Figure 4: Python Bindings Example

7 Conclusion

We have presented work in progress on `Smt-Switch`, a solver-agnostic API for performant prototyping with SMT solvers in C++. This system is available for use on GitHub and has already been used in projects [13, 15]. Future work includes adding a backend for Z3 [5], adding support for quantifiers and inductive datatypes, as well as investigating possible performance improvements, such as optimized data structures and alternatives to smart pointers.

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[3] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[4] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The Math-SAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[6] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[7] Levent Erkok. Sbv: Smt based verification in haskell. http://leventerkok.github.io/sbv/.

[8] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 373–384, 2015.

[9] Google. Googletest.

[10] Alex Horn. smt-kit: C++11 library for many sorted logics. http://ahorn.github.io/smt-kit/.

[11] KitWare. Cmake. https://cmake.org.

[12] Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. pytest 5.4.2, 2004.

[13] Makai Mann, Ahmed Irfan, Florian Lonsing, and Clark Barrett. Pono: an smt-based model checker. https://github.com/upscale-project/pono.

[14] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[15] Yoni Zohar, Ahmed Irfan, Makai Mann, Andres Nötzli, Andrew Reynolds, and Clark Barrett. lazybv2int. https://github.com/yoni206/lazybv2int.