metaSMT: Focus On Your Application Not On Solver Integration

Finn Haedicke Stefan Frehse Görschwin Fey Daniel Große Rolf Drechsler
Institute of Computer Science
University of Bremen, 28359 Bremen, Germany
{finn,sfrehse,fey,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Decision procedures are used as core technique in many applications today. In this context, automated reasoning based on Satisfiability Modulo Theories (SMT) is very effective. However, developers have to decide which concrete engine to use and how to integrate the engine into the application. Even if file formats like SMT-LIB standardize the input of many engines, advanced features remain unused and the integration of the engine is left to the programmer.**

This work presents *metaSMT*, a framework that integrates advanced reasoning engines into the program code of the respective application. *metaSMT* provides an easy to use language that allows engine independent programming while gaining from high performance reasoning engines.

State-of-the-art solvers for satisfiability and other theories are available for the user via *metaSMT* with minimal programming effort. For two examples we show how *metaSMT* is used in current research projects.

I. Introduction

In recent years, *formal methods* have become attractive to solve complex computational hard problems. Decision procedures are applied in many applications, like e.g., *Model Checking* [1], [2], *Synthesis* [3], [4] and, *Automatic Test Pattern Generation* (ATPG) [5].

Despite the successful research and application of decision procedures, the increasing complexity of software and hardware systems demands for more effective reasoning engines to overcome complexity issues. In the last years, solvers for *Satisfiability Modulo Theories* (SMT) have been developed. Different theories are combined to formulate the problem. Various works gave empirical evidence that SMT reasoning engines increase the efficiency of formal methods [6], [7], [8].

The performance of SMT reasoning engines remains an active research topic. Annual SMT competitions [9], [10] show their advances. However, SMT reasoning engines have different strengths on different problem instances. Therefore, evaluating different engines with respect to a given problem instance allows to find the best performing engine.

When using SMT in a concrete algorithm, the most common way is to generate a problem instance in SMT-LIB format [11]. Taking a user created SMT-LIB file as input, an SMT solver decides whether the instance is satisfiable or unsatisfiable. However, many solvers additionally have custom native interfaces. These interfaces are used to pass the instance to the engine and check for satisfiability. Furthermore, advanced features are available, e.g., computing interpolants which are utilized in *SAT-based Model Checking* [2]. Moreover learnt information generated while reasoning can be reused very efficiently in consecutive reasoning processes to prune the search space [12]. Usually this can only be done by calling native functions which access the learnt information.

This work presents *metaSMT* a publicly available, easy to use and powerful tool[1] which provides an integration of the native *Application Programming Interface* (API) of modern reasoning engines into C++ code. The advantages of *metaSMT* are: (1) engine independence through efficient abstraction layers (2) simple use of various decision procedures (3) extensibility in terms of input language and reasoning engines (4) customizability in terms of optimization and infrastructure (5) translation of the input language into native engine calls at compile time.

The remaining work is structured as follows: Section II gives a basic introduction into SMT and the programming methods used in *metaSMT*. Afterwards an example of a *metaSMT*-based application is given before in Section IV the architecture of *metaSMT* is described. Section V describes how this architecture is implemented in *metaSMT* and Section VI gives an empirical evaluation of *metaSMT* including its use in current research projects. The work closes with conclusions.

II. Preliminaries

This section provides background information. However, basic knowledge of C++ is assumed.

A. Satisfiability Modulo Theories

Boolean satisfiability is a decision problem, also known as the SAT problem. The problem asks whether there exists an assignment of Boolean variables such that the Boolean function evaluates to true. The problem has been proven NP-complete [13]. In spite of the huge complexity of the problem, sophisticated algorithms and clever heuristics help to solve instances with many thousands variables and clauses very efficiently. Usually, SAT solvers work on a *Conjunctive Normal Form* (CNF) of a Boolean function that is a disjunction of conjunctions of literals, where each literal is variable or its negation.

Satisfiability Modulo Theories is also a decision problem but with more complex theories rather than only propositional logic. A detailed introduction is given

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088.

[1] Available online at http://www.informatik.uni-bremen.de/agra/eng/metasmt.php

Listing 1. SMT instance for $a \cdot b = 21466342967$

```
(benchmark factorization.smt
  :logic QF_BV
  :extrafuns ((a BitVec[32]))
  :extrafuns ((b BitVec[32]))

  :assumption (not (= a bv1[32]))
  :assumption (not (= b bv1[32]))

  :formula (=
    bv21466342967[64]
    (bvmul
      (zero_extend[32] a)
      (zero_extend[32] b)
  ))
)
```

in [14]. Already available SMT-solvers handle complex formulas. In addition to the logics the SMT-LIB standard also specifies a textual format that is commonly used for input files of the solvers. Listing 1 shows an example of such an SMT file: Two variables, `a`, `b` are declared as bit-vectors and constrained to be the two factors of a product resulting in the 64 bit number $21,466,342,967$. When called with this input, an SMT-QF_BV solver outputs satisfiable and e.g., the assignment $a = 740,218,723$ and $b = 29$.

Moreover, in addition to the SMT-LIB format many solvers provide an *Application Programming Interface* (API) that exposes features like incremental solving, where the SMT instance can be changed after the satisfiability check. Learnt information about the instance is kept and reused. Consequently, using the API may increase the overall performance.

An SMT solver can be utilized within an application in different ways. Each of the following options has certain advantages and disadvantages:

1) Generate an instance file according to SMT-LIB specification and call the solver. Different SMT solvers can easily be evaluated. However, the instance generation and result retrieval requires handling of files and text within the application.
2) Use the solver specific API to call an SMT solver's functions directly. In particular, incremental satisfiability can be exploited. But the application is restricted to a specific solver.
3) Introduce an abstraction layer specific to the application. This technique combines high performance using incremental satisfiability with the ability to evaluate different solvers. However a custom layer is not portable to different applications.

This work separates the programming model from the reasoning engine. With *metaSMT* the application specifies the instance in a simple, common notation. Many reasoning engines are available without modifications of the algorithm.

The Java package jSMTLIB [15] provides an interface for the usage of different SMT solvers. The package reads, checks and generates SMT-LIB files and executes the respective SMT-solver executables. However, jSMTLIB currently does not provide an embedded language for instance generation directly from the application.

B. Boost.Proto

The Boost project [16] is a collection of libraries that cover many features not included in the C++ standard library. In particular, Boost.Proto [17] provides tools to integrate *Domain Specific Embedded Languages* (DSEL) into C++. Given a programming language, a DSEL in that language is dedicated to a domain, e.g., parsing [18] or vector arithmetic [19]. The DSEL provides a syntax designed for this domain and, therefore, is easier to handle than the original language. Due to space constraints an in-depth description of Boost.Proto is omitted, the reader is referred to the Boost documentation.

Technically, this work uses Boost.Proto to implement a domain specific language for SMT logics in *metaSMT*.

III. Motivating Example

Before the subsequent sections give a complete description of *metaSMT* this section demonstrates how *metaSMT* is used in a complete example application.

The usual integration of reasoning engines iteratively calls API functions to construct the problem instance. However, this reduces the readability of the source code and makes it difficult to understand the program. A typical example can be constructed using the Boolector API for the C programming language. Figure 1 (a) shows the code to construct the simple constraint $c = a \cdot b$ using the Boolector C API (not including memory management). The same expression, written quite concise in the SMT-LIB format is shown in Figure 1 (b). The goal of *metaSMT* is to allow this compact syntax to be used in C++ programs. Due to the limitations of C++ this requires adaptions. Most notably, the expressions cannot easily be written in a symbolic expression (S-expression) syntax as in the SMT-LIB format, where each function is enclosed in parenthesis. The syntax for calling functions is used instead. Moreover, the solver is passed into the expression using the context e.g., as `btor_ctx` in Figure 1 (c).

In order to illustrate the programming interface of *metaSMT* an example written in C++ code is presented in Listing 2. The listing shows the factorization of an integer into two integers. More precisely, given an integer in bit-vector representation $\vec{c} \in \mathbb{B}^{2n}$, compute two integers $\vec{a}, \vec{b} \in \mathbb{B}^n$, such that $\vec{a} \cdot \vec{b} = \vec{c}$. To enforce non-trivial factorization, both integers a and b may not be 1. As the bit-vectors $\vec{a}$ and $\vec{b}$ are zero-extended, no overflows are possible and as a result a valid assignment to the constraint either gives a valid factorization of $\vec{c}$ or proves that it is prime. The integer value of $\vec{c}$ is randomly chosen and changed in each iteration of a loop for 10,000 iterations. A similar example is reconsidered in the empirical evaluation.

The first line in Listing 2 defines the solver context, which is not explained here but described later in this work. The context specifies which reasoning engine to use and how the input is handled by *metaSMT*. The next line is a user parameter, which defines the bit-vector width of the operands. The algorithm is therefore scalable to an abitrary bit-width. In lines 4-6 the bit-vectors `a`, `b` and `c` are declared and initialized.

Lines 8 and 9 constrain the operands to be different from 1 in bit-vector representation. In line 11 the

```
boolector_assume( btor,
  boolector_eq(btor, c,
   boolector_mult(btor, a, b)
  ))
```

(a) Boolector API calls

```
:assumption
    (= c (bvmul a b))
```

(b) SMT-Lib format

```
assumption( btor_ctx,
    equal(c, bvmul(a, b)) )
```

(c) metaSMT C++ Code

Fig. 1. Examples for $c = a \cdot b$

Listing 2. *metaSMT* factorization and prime test

```
1  Context ctx;
2  const unsigned width = <parameter>;
3
4  bitvector a = new_bitvector(width);
5  bitvector b = new_bitvector(width);
6  bitvector c = new_bitvector(width);
7
8  assertion(ctx, nequal(a, bvuint(1,width)) );
9  assertion(ctx, nequal(b, bvuint(1,width)) );
10
11 assertion( ctx, equal( zero_extend(width, c)
       , bvmul( zero_extend(width, a),
       zero_extend(width, b)) ));
12
13 for (unsigned i=0; i < 10000; ++i) {
14   unsigned r = random_number ( 2, 2^width -
         1  );
15   assumption( ctx, equal(c, bvuint(r, 2*
         width)) );
16
17   if( solve( ctx ) ) {
18     unsigned a_value = read_value( ctx, a );
19     unsigned b_value = read_value( ctx, b );
20
21     printf("factorized %d into %d * %d\n", r
           , a_value, b_value);
22   } else {
23     printf("%d is prime.", r);
24   }
25 }
```

multiplication $\vec{a} \cdot \vec{b} = \vec{c}$ is constrained. However the multiplication is done in double width to avoid overflows.

These constraints are identical for each iteration of the loop starting in line 13, therefore they are declared outside the loop as an `assertion`, which is permanent.

Inside the loop, in lines 14-15 `c` is set equal to a random number from 2 to $2^{\text{width}} - 1$ using an `assumption`, which is only valid for the next satisfiability check of the solver, i.e., for one loop iteration.

After setting up the SMT instance, the satisfiability check is performed in line 17. If the instance is satisfiable, the values of `a` and `b` are determined using `read_value`. Both operands are printed out in line 21. Otherwise the instance is unsatisfiable, the else branch is executed, which outputs `c` is prime.

IV. Architecture

In the following sections the architecture of *metaSMT* is described. At first the basic layers are introduced. Then each layer is described in detail. The terms *frontend* for the input languages, *middle-end* for the intermediate layer and *backend* for the solvers are taken from compiler design to denote the *metaSMT* layers.

A. metaSMT *Layers*

metaSMT consists of three basic layers depicted in Figure 2. The frontend layer provides primitives of

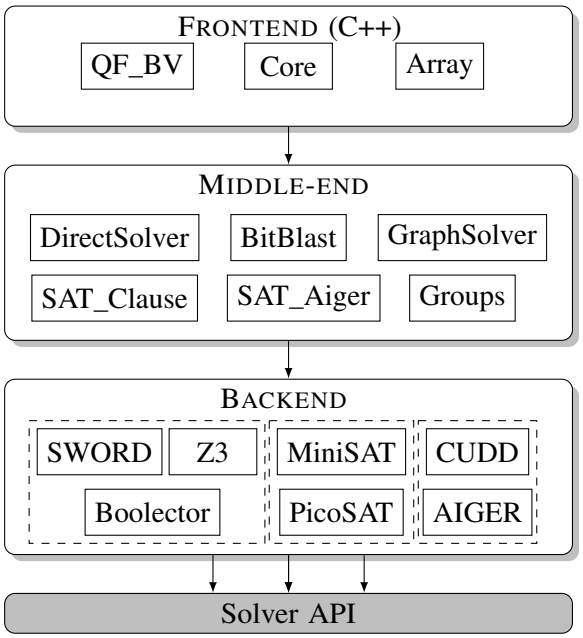

Fig. 2. *metaSMT* layer Architecture

the input languages, defined in the SMT-LIB format (e.g., Core Boolean logic, QF_BV). The middle-end layer provides translations, intermediate representation and optimizations of the input expressions. Optionally, expressions can directly be passed to the the backend layer where the solvers are integrated via their native API. Various configurations of middle-ends with backends are possible. The frontends allow to combine each translation middle-end with any compatible backend. However, not every backend supports every logic. Therefore, some middle-ends supply emulation or translations to other logics, e.g, a bit-vector expression can be translated into a set of Boolean expressions.

The frontends are independent from the underlying two layers and have no semantics attached. To evaluate frontend expressions, a *context* is used that defines their meaning. The context is the combination of at least one middle-end and one backend, where the middle-end defines how the input language is mapped to function calls of the backend.

B. Frontends

The frontends define the input languages for *metaSMT*. This includes Core Boolean logic and SMT QF_BV as well as a version of Array logic over bit-vectors. Each frontend defines its own set of available functions as well as public datatypes.

The Core Boolean logic defines the public datatype `predicate` which describes propositional variables.

Furthermore, Boolean constants are available, i.e., `true` and `false`. This logic also defines primitive Boolean functions, e.g., `Or`, `And`. The frontend creates a static syntax tree for the expression described in the code. This syntax tree is passed to the middle-end.

C. Middle-ends

The core of *metaSMT* are basic optimizations and translations from the frontend to the backend. While the frontends provide languages and the backends provide solver integrations, the middle-ends allow the user to customize *metaSMT*, i.e., on how the input language is mapped to the backend. Even in the middle-end itself, several modules can be combined.

1) DirectSolver: To enable a low-overhead translation from a frontend to a backend the `DirectSolver` is provided. All the elements of the input expression are directly evaluated in the backend. Variables are guaranteed to be constructed only once and are stored in a lookup table. For example, given a multiplication operation in QF_BV logic directly corresponds to a multiplication operation in the SMT solver Boolector.

The direct middle-end is very lightweight and allows the compiler to inline all function calls. For a modern compiler the resulting executable should perform equally well to a hand-written application using the same backend.

2) GraphSolver: Instead of adding the frontend expressions directly to the solver, they are first inserted into a directed graph. The graph models the explicit syntax tree of the expression as a *Directed Acyclic Graph* (DAG). Formally a node in the graph is a tuple $(\text{Operation}, \text{Static Arguments})$ where the SMT command and its static arguments are captured (e.g. `extract` and the range to extract). The edges point from an operation to the SMT expressions used in this command. A label on the edges stores the position of the subexpression in the command. Each time a new expression is evaluated it is first searched in a lookup table before a new node is created, when the node is not found. When the instance is checked for satisfiability, the graph is traversed and evaluated in the backend.

The graph-based translation provides a way to automatically detect common subexpressions and efficiently handle them to create smaller SMT instances which potentially increases performance of the reasoning process. This is especially useful if the user wants to automate this process, but either does not want to manually optimize the SMT instance or does not know the instance in advance because it is created dynamically inside the program.

3) Groups: This middle-end provides an interface and implementation of *constraint groups* for solvers that do not have native support for groups. A group is a set of constraints that belong together. The user can create groups, add expressions to them and delete them at any time. The solver will then disable all expressions in the group. Groups are emulated using *guard* variables and temporary assumptions, e.g., the expression $x \wedge y$ in group 1 is transformed to $g_1 \rightarrow (x \wedge y)$ using the guard variable g_1 and an implication. Depending on the solver deleting a group can either lead to the removal of the constraints or to the constraint just being disabled permanently.

4) BitBlast: This emulation of a QF_BV bit-vector backend uses only Core Boolean logic operations to allow the transparent use of SAT or BDD solvers with bit-vector expressions. The translation is performed in a standard way: Given only the Core Boolean logic, each bit-vector is transformed into a vector of Boolean variables. The bitwise operations can be applied easily, e.g., an exclusive-or over two bit-vectors is a bitwise exclusive-or for each pair of Boolean variables. The bit-vector predicates (`equal`, `less-than`, etc.) are mapped to a sequence of Boolean predicates, e.g., a conjunction of exclusive-nors for `equal`. Arithmetic operations are reduced to an equivalent Boolean expression.

D. Backends

The respective solvers and other constraint solving techniques are integrated as backends. For each reasoning engine a dedicated backend is created that maps from the internal *metaSMT* API to the API of the engine. Backends do not have an explicit interface to inherit from. They implement the required methods for the languages they support using C++ template mechanisms to integrate them into a context. This allows the compiler to optimize the code and, in the case of `DirectSolver`, produces code that is close to a hand-coded implementation using the same API.

This section gives an overview of the backends integrated into *metaSMT*. They are grouped by the input language they support. The compatibility of the solvers is also summarized in Table I.

1) Core Boolean logic backends: Several core logic backends as well as higher level backends are available. Core logic is directly supported by backends that accept all common Boolean operations. For example, the *Binary Decision Diagram* (BDD) package CUDD [20] supports all Boolean operations and is integrated in *metaSMT*. Furthermore, with some minor transformations based on De-Morgan *And-Inverter-Graph*s (AIGs) are also able to handle Boolean operations. Those AIGs are internally represented by the AIGER package [21]. SAT solvers can receive Boolean logic expressions either via the *SAT_Clause* adapter that creates one or more clauses per logic operation or via the *SAT_Aiger* adapter, that builds an AIG for the expression using the AIGER backend. Afterwards, the AIG is translated into a set of clauses. This infrastructure allow the usage of any SAT solver supporting CNF as input language either by an API or externally through files. PicoSAT [22] as well as MiniSAT [23] are directly supported as Core logic backends via their APIs. Other solvers are supported by generating CNF files and calling the executable of the SAT solvers.

Furthermore, all SMT QF_BV backends natively support Core logic as a subset of the language.

2) SMT QF_BV backends: Native SMT bit-vector solvers like Boolector [24], SWORD [25] and Z3 [26] are directly connected through their API for QF_BV support. Furthermore, the BitBlast middle-end provides an emulation for QF_BV using only basic logic operations. This emulation permits using QF_BV expressions in solvers that do not support them natively but support Core Boolean logic e.g., CUDD or SAT-solvers.

TABLE I
BACKEND COMPATIBILITY

BACKEND	CORE	QF_BV	ARRAY (BV)	SAT
AIGER [21]	yes	*emulated*	no	no
Boolector [24]	yes	yes	yes	no
CUDD [20]	yes	*emulated*	no	no
MiniSAT [23]	*emulated*	*emulated*	no	yes
PicoSAT [22]	*emulated*	*emulated*	no	yes
SWORD [25]	yes	yes	no	no
Z3 [26]	yes	yes	yes	no
BitBlast	yes	yes	no	no
SAT_Aiger	yes	*emulated*	no	no
SAT_Clause	yes	*emulated*	no	no

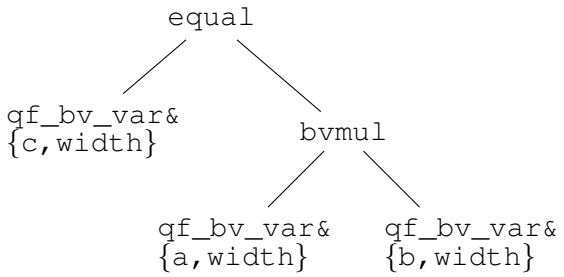

Fig. 3. Syntax Tree for `equal(c, bvmul(a, b))`.

3) SMT QF_ABV backends: In addition to Core Boolean logic and bit-vector logic the Boolector and Z3 backends also support arrays in the form of QF_QBV logic. Therefore *metaSMT* supports declaring and working with arrays over bit-vectors.

V. IMPLEMENTATION

This section describes how the architecture is implemented in *metaSMT* and how *metaSMT* is integrated in C++ programs.

A. Syntax and Semantics

For the evaluation of *metaSMT* expressions a context is used which defines syntax and semantics. The context concept and different kinds of contexts are described in this section.

The syntax component is provided by Boost.Proto. An expression like `equal(c, bvmul(a, b))` is created from the custom Boost.Proto functions `equal` and `bvmul` as well as the variables `a`, `b` and `c`. From the expression the syntax tree in Figure 3 is created. The nodes are labeled with the C++ *type* and strings inside the curly braces denote the content of the respective nodes. For *metaSMT* the tree is used as static type of the expression. The expression and the syntax tree are data, i.e., they neither have semantics attached nor trigger any actions.

The semantics for the expression is introduced by the *metaSMT context*, that defines how the syntax tree is evaluated and transformed for a specific solver. The evaluation of Boost.Proto-based expressions is performed in the *metaSMT* translation middle-end (e.g., GraphSolver or DirectSolver) so that the backends do not need to handle Boost.Proto expressions directly. This reduces the overhead to implement new backends.

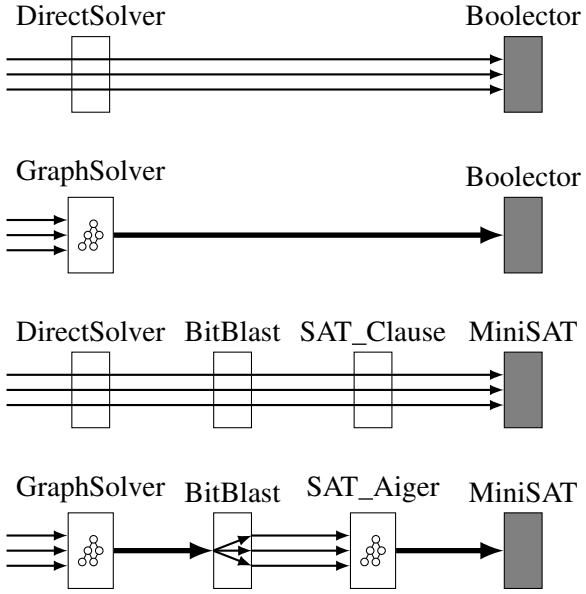

Fig. 4. Data flow in different contexts

Listing 3. *metaSMT* command grammar

```
command     ::= assert_cmd | assume_cmd
              | eval_cmd
              | solve_cmd  | result_cmd
assert_cmd  ::= 'assertion('  context ','
                expression ');'
assume_cmd  ::= 'assumption(' context ','
                expression ');'
eval_cmd    ::= 'evaluate('   context ','
                expression ');'
solve_cmd   ::= 'solve(' context ');'
result_cmd  ::= 'read_value(' context ','
                variable ');'
variable     ::= boolean_variable
               | bitvector_variable
expression  ::= <expression in metaSMT DSEL>
```

Figure 4 gives some example contexts and visualizes the data flow inside. This illustrates how different contexts can change the evaluation of a constraint. The first context defines a solver using Boolector without intermediate representation (DirectSolver). The context directly supports Core Boolean logic and QF_BV. In contrast, in the last example, MiniSAT is used. QF_BV as well as Core Boolean logic are emulated for this clause based backend. Furthermore this context uses a graph and an AIG as intermediate representations.

The GraphSolver-based and AIGER-based context first create an internal representation and pass the complete expression directly before solving. When using approaches without intermediate representation, the requests are forwarded to the next layer until they reach the backend. Only explicit transformations are applied before passing the expression (e.g., BitBlast, SAT_Clause).

B. Usage and API

The example from Listing 2 contains most of the core commands of *metaSMT*. These are summarized in Listing 3.

Listing 4. Programmatic constraint construction using temporary Variables

```
1 bitvector x = new_bitvector(8);
2 for( ... ) {
3   bitvector tmp = x;
4   x = new_bitvector(8);
5   assertion(ctx, equal( x, bvmul(tmp, ...)));
6 }
7 ...
8 solve(ctx)
```

The first three functions accept frontend expressions, however they have different effects. The functions `assertion` and `assumption` create the constraint instance where the first adds a constraint permanently to the (incremental) solver context while the latter adds the constraint for the next call of the solver only. In both cases the expression needs to have a Boolean result. The third function `evaluate` does not change the instance but returns a context specific representation of the input expression only.

To query a context for satisfiability, the `solve` function is provided. The result is a Boolean value directly representing `SAT` (true) or `UNSAT` (false). After a call to `solve` the assumptions are discarded while the assertions are still valid for the subsequent calls.

Getting a `SAT` result for `solve(ctx)`, i.e., the instance is satisfiable, a model is generated. The model can be retrieved with the `read_value` function. The function takes a context and a variable and returns the assignment of this variable in the given context. The result of `read_value` is automatically convertible to many C++ datatypes, including strings, bit-vectors (vector of `bool`, `tribool`, `bitset`) or integers.

In addition to these core commands, custom middle-ends may provide additional extensions. The *Group* middle-end for example provides functions to add groups, change the active group and delete groups. These functions cannot be used in any other context.

C. Expression Creation

Typically it is necessary to create the *metaSMT* expression at run time, e.g., in a loop. As *metaSMT* syntax trees are statically typed, an extension of the syntax tree is not possible. To work around this limitation, *metaSMT* provides two options. The first option is to create a partial expression and constrain equality to a temporary variable that is later reused to create the complete expression. This would allow strict grammar checking but introduces a temporary variable and a constraint, see Listing 4.

The second option is the use of the `evaluate(Ctx, Expr)` function and the context's `result_type`. The function takes a context and a frontend expression and returns the context specific representation of the expression. The result of the evaluation is of the backend specific type `Ctx::result_type`. This expression can be stored and later be used in other expressions. Note however that the return value is solver specific and therefore not portable or reusable in other contexts, not even contexts of the same type.

A powerful exception to this rule is the `result_type` of a `GraphSolver`-based context, where the result is a node in the internal graph.

Listing 5. Using a shared graph for different contexts

```
1 GraphSolver_Context<SWORD> sword;
2 GraphSolver_Context<Boolector> btor(sword);
3
4 GraphSolver_Context<SWORD>::result_type x =
    evaluate(sword, bvuint(0, 8));
5
6 for( ... ) {
7   x = evaluate(sword, equal( x, bvmul(x,
      ...)));
8 }
9 assertion(sword, x);
10 assertion(btor, x);
11 solve(sword) == solve(btor);
```

When a `GraphSolver` is constructed using the copy constructor, a shared graph is internally used by the contexts. The newly created solver also copies all assertions and assumptions, so that both solvers have the same internal state. In this setup the results of `evaluate` can be shared among the solvers. Each backend will only evaluate the parts of the graph that are required as parts of assertions or assumptions. The application of `evaluate` is demonstrated in Listing 5. This can be used for example when building multiple instances from the same base. At a specific point the context can be copied and from there both contexts can diverge into seperate instances.

VI. Emperical Evaluation

This section presents two different applications of *metaSMT*. Furthermore, a comparison of *metaSMT* with the native API of an SMT solver is presented. The experiments have been performed on a system with AMD Opteron 2222 SE processor, 32 GB RAM and a Linux operating system. In the following, the run times are always given in CPU seconds.

A. Exact Synthesis

This section presents examples from exact synthesis of reversible circuits [28], [29]. A reversible circuit computes a bijective function, i.e., there is a unique input and output mapping. Those circuits purely consist of reversible gates – here, the universal Toffoli gate and basic quantum gates are considered. The synthesis problem searches for a reversible circuit consisting of reversible gates which computes the function specified by a truth table. There are several exact approaches to synthesize those circuits. We considered the approach from [28], which translates the problem into a decision problem and asks for a circuit realization for a given number of gates. The size of the problem instances grows exponentially with number of input variables, because the entire truth table has to be taken into account. This usually results in hard problem instances even for small truth tables.

The underlying problem formulation has been encoded in QF_BV and an incremental algorithm searches for a valid circuit implementation. Using *metaSMT* sixteen different configurations have been evaluated. The configurations consist of four internal API backend solvers, i.e. Boolector, Z3, MiniSAT, and PicoSAT. Additionally, metaSMT is used to generated CNF files to run the external solvers PicoSAT, MiniSAT, PrecoSAT, and Plingeling [30]. All eight backends are used

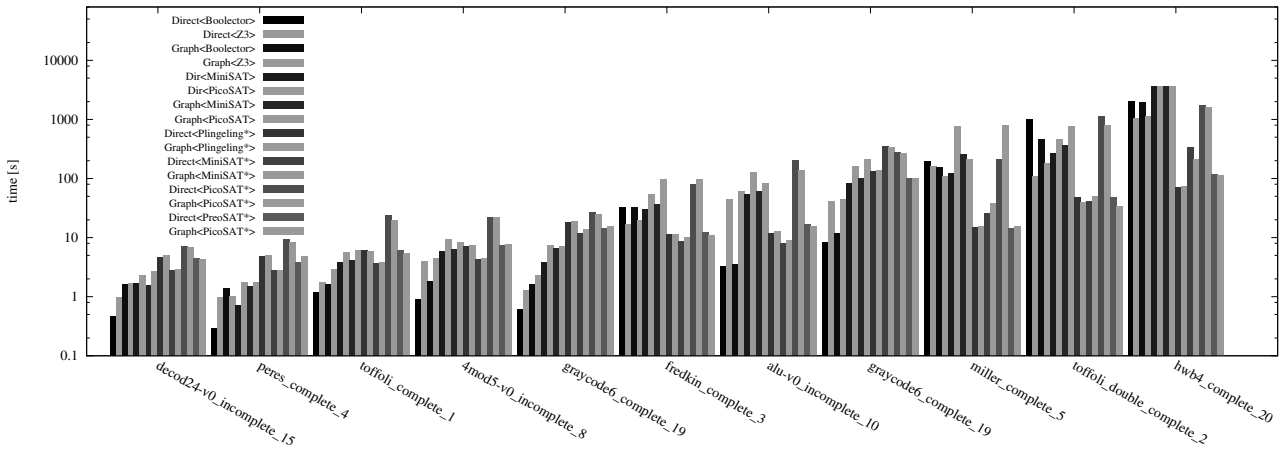

Fig. 5. Run times of reversible circuit synthesis for hard instances using a *metaSMT*-based version of RevKit [27]

with the `DirectSolver` middle-end as well as the `GraphSolver` middle-end.

For the synthesis 11 specifications were used which are publicly available from [31]. A timeout of 3,600 seconds was applied. The results are presented in Figure 5. On the x-axis the respective benchmark is shown, whereas the y-axis denotes the run time in seconds for each configuration in logarithmic scale. Externally called solvers are marked with $*$.

From Figure 5 it is clear that no single solver dominates the benchmarks. For example, for the benchmarks from `decode24-v0_incomple_5` to `graycode6_complete_19` Boolector performs much better than any other solver. But for the benchmarks from `miller_complete_5`, Boolector is outperformed by the SAT solvers. MiniSAT as well as PicoSAT are evaluated as internal and external versions. The accumulated run times of the solvers MiniSAT and PicoSAT over all benchmarks are 18,790 seconds for the internal version and 8,989 seconds for the external version. Surprisingly, the externally called solvers are much better here than the internal called solvers, though the internal solvers may benefit from learnt information. Overall, the externally called PrecoSAT solver needs 326.24 seconds over all benchmarks and Boolector needs 3,285.05 seconds.

To summarize, all the presented results can be achieved very easily by using *metaSMT*. Only one parameter needs to be switched to run a different solver.

B. Mutation Testing

Given a program, by mutation several faulty variants of the program are created and combined into a single program called meta-mutant. Using *metaSMT* the meta-mutant is encoded into a set of constraints. Each satisfying assignment yields a test case that discovers at least one fault. Experiments of [32] where executed on a set of *metaSMT* contexts. The results are shown in Table II. Comparing the Boolector backend with the MiniSAT backend using DirectSolver as well as GraphSolver middle-ends.

The results significantly vary with the difficulty of the instance. For easy instances the directly connected Boolector contexts and the graph-based MiniSAT perform better. However, for difficult instances with a run time over 1 minute, the graph-based Boolector context is fastest while MiniSAT-based contexts require significantly more time. For these harder instances the GraphSolver middle-end outperforms the direct variant of the same backend. This effect is most likely due to the removal of redundancies in the graph. For MiniSAT this amounts to run time reductions of around 50%. With *metaSMT* as abstraction layer it is easy to evaluate the effects of different contexts or optimizations. When changes are only in the abstraction layer no application code needs to be changed and only little effort is required.

TABLE II
RESULT FOR MUTATION TESTING USING DIFFERENT CONTEXTS

Name	Time [s]			
	Direct Boolector	Graph Boolector	Direct MiniSAT	Graph MiniSAT
isl	0.05	0.22	0.34	0.47
min	0.34	0.48	0.82	0.65
mid	4.73	5.07	7.87	4.36
fmin3	5.02	5.85	4.45	2.55
fmin5	21.08	25.96	14.92	9.56
fmin10	310.52	260.38	997.65	550.94
tri	207.69	193.99	1596.99	652.64

C. Comparison with direct API

For the factorization algorithm from Listing 2 a hand coded implementation using the Boolector C API is compared to a *metaSMT* DirectSolver-based implementation with the Boolector backend. The resulting application is available as part of the *metaSMT* package.

The experiment has the following setup: The algorithm from Listing 2 was changed to work on sequences instead of generating random numbers. A sequence of 10,000 random 64 bit numbers is generated and the algorithm it applied to it 10 times. The same sequence is used for both the hand coded and the *metaSMT*-based implementation of the algorithm. The complete experiment was repeated 5 times with Boolector being executed first and 5 times with *metaSMT* being executed first. Altogether each solver was forced to solve 1,000,000 constraints of 64 bit numbers factorized into two 32 bit numbers.

The results showed no significant difference caused by the *metaSMT* abstraction layer: $1,736s$ for plain Boolector compared to $1,729s$ for *metaSMT* with Boolector, i.e., 1.7 seconds for 10,000 factorizations.

D. Other applications

In addition to the aforementioned projects *metaSMT* is also used in a Constraint Random Stimuli generation Library. In the library elements of the SystemC Verification Library [33] and techniques from [34] are combined.

VII. Conclusions

metaSMT is a library that abstracts details of reasoning engines. Based on *metaSMT* very little programming effort is required to integrate formal methods into a user's application. Once this has been done, a wide range of solvers as well as optimization techniques can be selected.

Various research projects already integrate *metaSMT*. Future work on *metaSMT* includes the development of the following features: New frontend logics will complete the support for SMT logics (e.g. uninterpreted functions, integer arithmetic), while new middle-ends will increase solving performance (e.g. portfolio or multi-threaded contexts) and new backends will provide access to additional SMT solvers.

VIII. Acknowledgments

We would like to thank Hoang M. Le, Heinz Riener and Fereshta Yazdani for the helpful discussions, their proposals for improvements, testing and contributions to *metaSMT*.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[2] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, ser. LNCS. Springer Berlin / Heidelberg, 2003, vol. 2725, pp. 1–13.

[3] E. Arbel, O. Rokhlenko, and K. Yorav, "SAT-based synthesis of clock gating functions using 3-valued abstraction," in *Formal Methods in Computer-Aided Design, 2009*, 2009, pp. 198 –204.

[4] F. Haedicke, B. Alizadeh, G. Fey, M. Fujita, and R. Drechsler, "Polynomial datapath optimization using constraint solving and formal modelling," in *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, 2010, pp. 756 –761.

[5] R. Drechsler, S. Eggersgluss, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille, "On acceleration of SAT-based ATPG for industrial designs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 7, pp. 1329 –1333, 2008.

[6] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *International Conference on Computer-aided design*. New York, NY, USA: ACM, 2006, pp. 794–801.

[7] P. Bjesse, "A practical approach to word level model checking of industrial netlists," in *International Conference on Computer Aided Verification*, 2008, pp. 446–458.

[8] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, pp. 69–83, 2009.

[9] "SMT-COMP 2009," http://www.smtcomp.org/2009, 2009.

[10] "SMT-COMP 2010," http://www.smtcomp.org/2010, 2010.

[11] S. Ranise and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," http://www.smtlib.org, 2006.

[12] O. Shtrichman, "Pruning techniques for the SAT-based bounded model checking problem," in *CHARME*, ser. LNCS, vol. 2144, 2001, pp. 58–70.

[13] S. Cook, "The complexity of theorem proving procedures," in *3. ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[14] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Satisfiability Modulo Theories*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185, ch. 26, pp. 825–885.

[15] D. R. Cok, "jSMTLIB: Tutorial, validation and adapter tools for smt-libv2," in *NASA Formal Methods*, ser. LNCS, 2011, vol. 6617, pp. 480–486.

[16] "Boost C++ libraries," http://www.boost.org/.

[17] E. Niebler, "Proto: A compiler construction toolkit for DSELs," in *Proceedings of the 2007 Symposium on Library-Centric Software Design*, ser. LCSD '07. New York, NY, USA: ACM, 2007, pp. 42–51.

[18] J. de Guzman and D. Nuffer, "The Spirit library: Inline parsing in C++," *C/C++ User Journal*, vol. 21, no. 9, 2003.

[19] T. L. Veldhuizen, "Arrays in Blitz++," in *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, 1998, pp. 223–230.

[20] F. Somenzi, *CUDD: CU Decision Diagram Package Release 2.4.1*. University of Colorado at Boulder, 2009.

[21] "Aiger," http://fmv.jku.at/aiger/.

[22] A. Biere, "Picosat essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.

[23] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[24] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009, pp. 174–177.

[25] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "Sword: A SAT like prover using word level information," in *VLSI of System-on-Chip*, 2007, pp. 88–93.

[26] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[27] M. Soeken, S. Frehse, R. Wille, and R. Drechsler, "RevKit: A toolkit for reversible circuit design," in *Workshop on Reversible Computation*, 2010, pp. 69 – 72.

[28] D. Große, R. Wille, G. Dueck, and R. Drechsler, "Exact multiple-control toffoli network synthesis with SAT techniques," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 5, pp. 703 –715, May 2009.

[29] R. Wille, D. Große, M. Soeken, and R. Drechler, "Using higher levels of abstraction for solving optimization problems by boolean satisfiability," in *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, 2008, pp. 411 –416.

[30] A. Biere, "Lingeling, plingeling, picosat and precosat at SAT race 2010," Tech. Rep., 2010. [Online]. Available: http://fmv.jku.at/papers/Biere-FMV-TR-10-1.pdf

[31] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *Int'l Symp. on Multi-Valued Logic*, 2008, pp. 220–225, RevLib is available at http://www.revlib.org.

[32] H. Riener, R. Bloem, and G. Fey, "Test case generation from mutants using model checking techniques," in *International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 388 – 397.

[33] *SystemC Verification Standard Specification Version 1.0e*, SystemC Verification Working Group.

[34] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC verification library," in *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's: Selected Contributions on Specification, Design, and Verification from FDL 2009*, ser. Lecture Notes in Electrical Engineering, D. Borrione, Ed. Springer Netherlands, 2010, vol. 63, pp. 227–244.