Converting SMTInterpol Proofs to Isabelle

Christian Schilling

March 21, 2013

1 Outline

We describe the conversion of proofs of unsatisfiability given by `SMTInterpol`. The proofs are converted to the interactive theorem prover `Isabelle`, which then can be used to check the correctness of the proof. The document is structured as follows: In Section 2 the general approach is exposed. The low-level conversion is explained in Section 3. Section 4 covers the more detailed conversion of sub-proofs.

TODO

Describing the proofs, we use certain variables with fixed sorts. These are:

sort	names
Bool	p, q, r
Int	x, y, z
Real	u, v, w

TODO

Proof examples will be given in a listing manner. On the left-hand side will be the keyword for the next step and on the right-hand side will be the resulting formula (either in `SMT-LIB` or `Isabelle` syntax). Some recurring keywords are *Input* for the input formula, *Proof* for an excerpt of the `SMTInterpol` proof, *Isabelle* for the translation into `Isabelle` syntax, and *Result* for the resulting formula of the recent proof step in `Isabelle` (together with instantiations of pattern variables).

TODO

When considering proof rules, we often want to talk about a specific term. We will then call this the *target term*, or *TT* for short.

2 General Proceeding

`Java` class file: `ProofChecker`

If a formula of any supported theory is passed to `SMTInterpol` and turns out to be unsatifiable, `SMTInterpol` can generate a proof.

The syntax follows `SMTInterpol`'s internal syntax, which again is closely related to the `SMT-LIB` 2.0 syntax. But for simplification many redundant symbols are not used, such as $\wedge, >, \geq$, and multiple $\neg$. Therefore formulae (and hence *ProofNodes*) always appear as (possibly negated) disjunctions (clauses).

TODO

Figure 1: proof tree example

The input file is given to the converter, which checks if there exists a name collision (SMT-LIB supports naming input formulae via the `:named` annotation) in the input formula with the conversion-internal names passed to `Isabelle` or with general `Isabelle` identifier syntax (special characters are barely supported). If so, the names are changed accordingly (an additional prefix is added and formulae with invalid syntax or without a name at all are assigned a standard name). The modified formulae are then passed to the solver, which hopefully generates a proof. The proof is received and then translated and written to a file. In the end `Isabelle` can read and check the file.

3 Basic Conversion

`Java` class files: `ProofConverter, TermConverter, LetHandler`

The proof is given as a binary tree of proof nodes (*ProofNodes*), where the leaves are (theory or logic) lemmata and the inner nodes are justified by resolution (see Figure 3 for an example). Their respective conversion is explained later in Section 4, while in this section we describe the "infrastructure".

The proof tree is traversed from the root to the leaves. But since the `Isabelle-ISAR` syntax suggests forward proofs, we use a stack. By pushing inner *ProofNodes* of a resolution *ProofNode*, we can delay its conversion and start the proof with the children. When they have been converted, the resolution *ProofNode* itself is converted.

3.1 Let and Lemmata

TODO

4 Main Proof Elements

Having explained the general approach in the last section, the conversion of the single steps (or nodes) in the proof tree are given here. They are called *ProofNodes* in `SMTInterpol` and currently there exist only four (rather six) different types. Each of these has its individual conversion, since they differ too much to be handled similarly. The types of proof nodes are:

1. resolution (`@res`)
2. lemma (`@lemma`)
    (a) equality theory (`:CC`)
    (b) linear arithmetic theory (`:LA`)
    (c) trichotomy (`:trichotomy`)
3. assertion (`@magic`)

4. tautology (`@tautology`)
5. substitution (`@eq`), using
   (a) rewrite equality (`@rewrite`)
   (b) internal rewrite equality (`@intern`)
6. splitting (`@split`)

We now explain the conversion approaches in full details.

4.1 Resolution

`Java` class file: `ResolutionConverter`

The resolution rule is the only rule with an argument and hence the "constructive" rule for creating the proof tree. It takes two proof nodes and returns another proof node:

$$\texttt{@res} : \mathit{ProofNode} \times \mathit{ProofNode} \to \mathit{ProofNode}$$

We can derive that a tree with $n > 1$ leaves always contains exactly $n - 1$ resolution steps, so half of the sub-proofs in the whole proof tree are resolution proofs. To put it differently: The proof tree is always a binary tree, where the inner nodes are the resolution nodes and the leaves are other proof nodes (without any arguments) (see Figure 3 for an illustration).

The arguments are two clauses, and for discrimination we call them (I) and (II). The only assertion known about them is the fact that one particular disjunct – the so-called *pivot* – is contained, at which negated in (I). With a set-based view on clauses we have this structure:

$$(I) = \{\neg p, \dots\}, \quad (II) = \{p, \dots\}$$

The resolution uses this fact: We can "unite" two disjunctions with a literal (the *pivot*) positive in one and negative in the other disjunction, while removing these literals. That gives us the following rule:

$$\frac{\neg p \vee x_1 \vee \cdots \vee x_k \qquad p \vee x_{k+1} \vee \cdots \vee x_\ell}{x_1 \vee \cdots \vee x_\ell} \text{ pivot } = p$$

To be more exact, recall the set-based view of a clause. The rule translates to:

$$\frac{(I) \equiv \{\neg p, \bigvee_{i=1}^{k} x_i\} \qquad (II) \equiv \{p, \bigvee_{i=k+1}^{\ell} x_i\}}{((I) \setminus \{\neg p\}) \cup ((II) \setminus \{p\})} \text{ pivot } = p$$

Note that there is no limitation that the x_i have to be pairwise disjunct. In fact, the more literals occur in both clauses, the shorter the result gets (which is desirable), because we only have to keep them once in the clause. Further note the following invariant:

Invariant: All clauses in the proof tree contain their literals exactly once.

This is automatically assured by the leaves in the tree. By guaranteeing that the result of the resolution rule does not violate it, it must always hold. So the first version of the resolution rule is only useful if the clauses do not share any x_i (except for the pivot literals). This gives rise to a more specific rule that is closer to the characteristics of a set containing elements only once.

$$\frac{\neg p \vee \bigvee_{i=1}^{j} x_i \vee \bigvee_{i=j+1}^{k} x_i \qquad p \vee \bigvee_{i=j+1}^{k} x_i \vee \bigvee_{i=k+1}^{\ell} x_i}{\bigvee_{i=1}^{j} x_i \vee \bigvee_{i=j+1}^{k} x_i \vee \bigvee_{i=k+1}^{\ell} x_i} \text{pivot} = p$$

Here the x_1 to x_j are only contained in (I), the x_{j+1} to x_k are contained in both clauses and the x_{k+1} to x_ℓ are only contained in (II). Note that this formulation may suggest, that the pivot does not occur in these three groups, but this is not the case.

TODO
- ResolutionWalker
- general structure, example, own result format
- proof rules for pivot and movement
- case distinction (8)
- substitution
- special cases

4.2 Lemma CC

`Java` class file: `LemmaCCConverter`

TODO

4.3 Lemma LA

`Java` class file: `LemmaLAConverter`

The proven formula is a disjunction of one of the following inequality literals $\bar{\ell_i}$:

$$\begin{aligned} a_1x_1 + \cdots + a_nx_{k_i} + c_i \preceq 0 &\qquad \text{where } \preceq \in \{<, \leq\} \\ \neg(a_1x_1 + \cdots + a_nx_{k_i} + c_i \preceq 0) &\qquad \text{where } \preceq \in \{<, \leq, =\} \end{aligned}$$

The proof goes by contraposition, hence we assume $\neg(\bar{\ell_1} \vee \cdots \vee \bar{\ell_n})$. After applying de Morgan's rule we receive a conjunction of negated literals ℓ_i. For the remaining proof description, we only refer to these negated literals. Note that equality can only appear in positive manner now.

The proof annotation given by `SMTInterpol` gives a Farkas coefficient f_i for each literal, which is just an integer different from zero. The basic idea is to multiply the literals with the respective factor and sum up the resulting inequalities. Of course, then the inequality signs must be equal (or at least similar, i.e. only $<$ and $\leq$), but this is always the case for the proven formulae (for equality, see the explanation below). In case of a inequality, the Farkas coefficient always fits the sign, i.e., a negative literal has a negative coefficient and vice versa. In case of an equality, the coefficient can be arbitrary, so it looks like

$$c \cdot (a_1x_1 + \cdots + a_nx_{k_i} + c_i) = 0,$$

which is equivalent to

$$(c \cdot (a_1 x_1 + \cdots + a_n x_{k_i} + c_i) \leq 0) \wedge (c \cdot (a_1 x_1 + \cdots + a_n x_{k_i} + c_i) \geq 0).$$

We can safely drop the second conjunct, since this only weakens the formula (remember that it became a conjunction).

The only two kinds of literals we receive are thereby:

$$|c| \cdot (a_1 x_1 + \cdots + a_n x_{k_i} + c_i) \preceq 0 \qquad \text{where } \preceq \in \{<, \leq\}$$

There are three kinds of linear arithmetic logics supported by `SMTInterpol`: `integer`, `real` and `mixed`. In the integer case we can even assure that $<$ never occurs: Simply rewrite $x < 0$ by $x + 1 \leq 0$. For negated literals, this has to be done before we multiply with the coefficient, otherwise some proofs do not go through. This can be seen in the example: If we multiplied the third literal before doing this transformation, we would end up with $0 \leq 0$, which is no contradiction.

The `real` and `mixed` cases are not converted so far.

FIXME

4.3.1 Example

Let $x_1, x_2 \in \mathbb{Z}$. We prove the lemma given in line 1 of the following proof.

Lemma		$\neg(-3x_1 + 2x_2 + 2 \leq 0)$	$\vee$	$\neg(x_1 - 1 = 0)$	$\vee$	$(x_2 \leq 0)$	
assume	$\neg($	$\neg(-3x_1 + 2x_2 + 2 \leq 0)$	$\vee$	$\neg(x_1 - 1 = 0)$	$\vee$	$(x_2 \leq 0)$	$)$
de Morgan		$(-3x_1 + 2x_2 + 2 \leq 0)$	$\wedge$	$(x_1 - 1 = 0)$	$\wedge$	$\neg(x_2 \leq 0)$	
transformation =		$(-3x_1 + 2x_2 + 2 \leq 0)$	$\wedge$	$(x_1 - 1 \leq 0)$	$\wedge$	$\neg(x_2 \leq 0)$	
Farkas coefficients		$1 \cdot (-3x_1 + 2x_2 + 2 \leq 0)$	$\wedge$	$3 \cdot (x_1 - 1 \leq 0)$	$\wedge$	$-2 \cdot \neg(x_2 \leq 0)$	
transformation $\mathbb{Z}$		$1 \cdot (-3x_1 + 2x_2 + 2 \leq 0)$	$\wedge$	$3 \cdot (x_1 - 1 \leq 0)$	$\wedge$	$-2 \cdot (x_2 - 1 \geq 0)$	
calculation		$(-3x_1 + 2x_2 + 2 \leq 0)$	$\wedge$	$(3x_1 - 3 \leq 0)$	$\wedge$	$(-2x_2 + 2 \leq 0)$	
total sum		$(0x_1 + 0x_2 + 1 \leq 0)$					

So we end up with a contradiction: $1 \leq 0$. Hence the lemma is proven. $\square$

4.3.2 Proof in Isabelle

TODO

4.4 Lemma Trichotomy

`Java` class file: `LemmaTrichotomyConverter`

Description

Trichotomy is the three-partition of numbers by the usual order relation: In comparison to 0, a number x is either less, equal, or greater ($x < 0$, $x = 0$, or $x > 0$).

In `SMTInterpol` this becomes the following ternary disjunction:

```
(or E L G)
```

where the disjuncts are dependent on the variable sort (integer or real). Note that the order of the disjuncts can vary. They are already given in canonical form, so we have:

	integer case	real case
`E`	`(= x 0)`	`(= x 0.0)`
`L`	`(<= (+ x 1) 0)`	`(< x 0.0)`
`G`	`(not (<= x 0))`	`(not (<= x 0.0))`

Translation

The translation first detects the order of the disjuncts. This is trivial, since the `E` disjunct always has equality and the `G` disjunct always has negation as function symbol. In case the order deviates from `ELG`, we bring the disjunction to this form. This is possible, since the trichotomy is only followed by resolution steps. Then the proof rule just depends on the variable sort.

Proof

```
trichotomy_int    (y::int) = (x + 1) ==>
                    (x = 0) | (y <= 0) | ~ (x <= 0)
trichotomy_real   ((u::real) = 0) | (u < 0) | ~ (u <= 0)
```

For real x the lemma is proven with a single rule. For integer x the rule has an obligation $y = x+1$, where y is the left-hand side of the `<=` disjunct. This cannot be covered by the rule, because x itself can be a sum with a constant, which `SMTInterpol` already adds, so we really need a mathematical computation. This is finally done by the simplifier.

integer case	`(rule trichotomy_int, simp)`
real case	`(rule trichotomy_real)`

Example

```
Input      (not (or (= x y) (< x y) (> x y)))
Proof      (or (<= (+ y (- x) 1) 0) (= (+ y (- x)) 0)
             (not (<= (+ y (- x)) 0)))
Reorder    (or (= (+ y (- x)) 0) (<= (+ y (- x) 1) 0)
             (not (<= (+ y (- x)) 0)))
Isabelle   (y + - x = 0) | (y + - x + 1 <= 0) | ~ (y + - x <= 0)
rule trichotomy_int: ?x ↦ (y + - x), ?y ↦ (y + - x + 1)
Result     (y + - x + 1) = ((y + - x) + 1)
simp
```

4.5 Tautology

`Java` class file: `TautologyConverter`

FIXME

4.6 Assertion

In the partial proof mode this proof node just tells from which assertion a given formula is derived. This is only proven with the help of the built-in automatic tactic *auto*. In many cases this works perfectly, but there are (even short) examples where this tactic does not find the proof. That is the price for having short proofs.

In the extended proof mode, this proof node only introduces the assertion from the input as an axiom. The translation is just by `note F`, where `F` is the name associated with the assertion (cp. TODO).

4.7 Substitution

`Java` class file: `SubstitutionConverter`

FIXME

4.8 Splitting

`Java` class file: `SplitConverter`

Description

The splitting starts with the canonical term form (the DAG) and afterwards (and after some further rewrites such as elimination of double negation) the resulting formula is in CNF and hence passed to the solver/to the resolution level. The name splitting may be disturbing: There is only one application of this proof node that really does a split (but it is also by far the most often occurring one). The other applications are rather used as rewrites.

The several possible applications are indicated by an annotation. We now list them with the respective proofs, (cp. section 4.2 [1]).

annotation	description	SMTInterpol proof rule
`:notOr`	split conjunct from conjunction	$\dfrac{\texttt{(not (or}_{i\in I}\ \texttt{t}_i\texttt{))}}{\texttt{(not t}_j\texttt{)}}\ j \in I$
`:=+1`	positive Boolean equality 1	$\dfrac{\texttt{(= F}_1\ \texttt{F}_2\texttt{)}}{\texttt{(or F}_1\ \texttt{(not F}_2\texttt{))}}$
`:=+2`	positive Boolean equality 2	$\dfrac{\texttt{(= F}_1\ \texttt{F}_2\texttt{)}}{\texttt{(or (not F}_1\texttt{) F}_2\texttt{)}}$
`:=-1`	negative Boolean equality 1	$\dfrac{\texttt{(not (= F}_1\ \texttt{F}_2\texttt{))}}{\texttt{(or F}_1\ \texttt{F}_2\texttt{)}}$
`:=-2`	negative Boolean equality 2	$\dfrac{\texttt{(not (= F}_1\ \texttt{F}_2\texttt{))}}{\texttt{(or (not F}_1\texttt{) (not F}_2\texttt{))}}$
`:ite+1`	positive if-then-else 1	$\dfrac{\texttt{(ite F}_1\ \texttt{F}_2\ \texttt{F}_3\texttt{)}}{\texttt{(or (not F}_1\texttt{) F}_2\texttt{)}}$
`:ite+2`	positive if-then-else 2	$\dfrac{\texttt{(ite F}_1\ \texttt{F}_2\ \texttt{F}_3\texttt{)}}{\texttt{(or F}_1\ \texttt{F}_3\texttt{)}}$
`:ite-1`	negative if-then-else 1	$\dfrac{\texttt{(not (ite F}_1\ \texttt{F}_2\ \texttt{F}_3\texttt{))}}{\texttt{(or (not F}_1\texttt{) (not F}_2\texttt{))}}$
`:ite-2`	negative if-then-else 2	$\dfrac{\texttt{(not (ite F}_1\ \texttt{F}_2\ \texttt{F}_3\texttt{))}}{\texttt{(or F}_1\ \texttt{(not F}_3\texttt{))}}$

Translation

The translation depends on the annotation. The only important rule is `:notOr`, the others are straight-forward transformations (no pitfalls).

For this rule we split a conjunct from a conjunction (which is fine). Since `SMTInterpol` uses the clauses as negated disjunctions, we really split a disjunct (TT, $\texttt{t}_j$ in the rule) from a negated disjunction and negate the result.

Disjunction is right-associative in Isabelle, so we only consider binary disjunctions. The proof goes by elimination tactics (*elim*), that is, the rules are repeatedly applied until either none is available or the goal is closed. If the first (negated) disjunct is the TT, the proof is finished with a binary split rule. If not, the left-hand side of the disjunction is dropped and the search goes on recursively within the right-hand part. Note that this relies on the fact that the TT is at the lowest level of the disjunction (no flattening applied).

If the TT is the rightmost (negated) disjunct, the proof will end up with the obligation $\sim\ \texttt{p} \implies\ \sim\ \texttt{p}$. Normally, this is automatically solved by the *by* command in Isabelle. But `p` itself can be a disjunction. To prevent *elim* to go on searching there, another rule is necessary for this special case.

We look at the last disjunct and depending on if it is the TT we insert the according finishing rule. Note that this is not necessary, we could just insert both finishing rules, but the check is cheap and this way the Isabelle proof works faster.

Proof

```
split_notOr_elim   [|~ (p | q); ~ q ==> r|] ==> r
split_notOr_finL   ~ (p | q) ==> ~ p
split_notOr_finR   ~ (p | q) ==> ~ q
```

The first rule is the elimination step: If we have a negated disjunction, we can drop the first disjunct, assuming that the rest will still show the goal.

The second rule stops the elimination if the first disjunct is the TT.

The third rule is necessary for the special case that the TT is the right-most disjunct. It stops the elimination when there is only a binary disjunction left.

normal case	`(elim split_notOr_finL split_notOr_elim)`
special case	`(elim split_notOr_finR split_notOr_elim)`

The other rules are just listed, for they are obvious. The proof always goes like `(rule RULE)`, where `RULE` is the name of the rule from the table.

`split_eqP1`	`p = q ==> p | ∼ q`
`split_eqP2`	`p = q ==> (∼ p) | q`
`split_eqM1`	`p ∼= q ==> p | q`
`split_eqM2`	`p ∼= q ==> (∼ p) | ∼ q`
`split_iteP1`	`if c then t else e ==> (∼ c) | t`
`split_iteP2`	`if c then t else e ==> c | e`
`split_iteM1`	`∼ (if c then t else e) ==> (∼ c) | ∼ t`
`split_iteM2`	`∼ (if c then t else e) ==> c | ∼ e`

Example

Since all the other rules are straight-forward, we only give an example for the `:notOr` rule. However, the examples file also contains applications for the other cases.

```
Input     (and (not p) (not q) (or (and p q) q))
Proof     (not (or p q (not (or (not (or (not p) (not q))) q))))
            split (not q)
Isabelle  ∼ (p | [q | (∼ ((∼ ((∼ p) | (∼ q))) | q))]) ==> ∼ q
elim split_notOr_finL split_notOr_elim
try       rule split_notOr_finL – not applicable
try       rule split_notOr_elim – applicable
  ?p ↦ p, ?q ↦ (q | (∼ ((∼ ((∼ p) | (∼ q))) | q))), ?r ↦ q
Result    ∼ (q | (∼ ((∼ ((∼ p) | (∼ q))) | q))) ==> ∼ q
try       rule split_notOr_finL – applicable
```

5 Summary

TODO

References

[1] Jürgen Christ, *Proof System for SMTInterpol 2.0*, 2012