Interpolation in SMTLIB 2.0

Jürgen Christ Jochen Hoenicke

2012/06/30

We propose one new option `:produce-interpolants` and one new command `get-interpolants` for SMTLIB version 2 script format. The option should be set before the `set-logic` command to tell the solver that the user is interested in interpolants. The `get-interpolants` command is only supported if the option is set to **`true`**.

Rationale. For performance reasons a solver can omit the proof tracking necessary to compute interpolants unless this option was set. To ease solver implementation we do not allow this option to be changed after the `set-logic` command.

To compute Craig interpolants the interpolation problem must first be given via `assert` commands and checked for unsatisfiability using `check-sat`. The asserted formulas should have a `:named` annotation that provides a name to the formula, i.e., the formulas are asserted in the form

```
(assert (! formula :named some_name))
```

The name is referenced in the `get-interpolants` call. The simplest form of this call is

```
(get-interpolants A B)
```

where `A` and `B` are the names of the formulas for which the interpolant should be computed. The solver should reply with a parenthesized *term* that represents an interpolant for A and B, i. e., a formula I with

- $A \wedge \neg I$ is unsatisfiable.
- $I \wedge B$ is unsatisfiable.
- I contains only the symbols shared between A and B or theory defined symbols.

Rationale. This syntax is similar to the way the command `get-unsat-core` works. In both cases, the formulas are given `:named` annotations and asserted and the unsatisfiability is checked before-hand using `check-sat`. The difference is that for unsat cores the names are used in the result, while for interpolation they are used as input to the command.

Separating `check-sat` and `get-interpolants` also allows to compute several interpolants (with different partitions) for the same formula.

The parenthesis around the answer become clear when considering the extension for computing a sequence of interpolants.

Theory extensions

All unnamed formulas and all named formulas that are not mentioned in the `get-interpolants` command are considered to be theory extensions. Thus the unsatisfiability of the formulas $A \wedge \neg I$ and $I \wedge B$ need only hold in the context of the other formulas. Also I may contain any symbol occurring in an unnamed formula.

Inductive Sequences of Interpolants

As a simple extension `get-interpolants` can also be called with more than two named formulas. The reply to the command

```
(get-interpolants F1 F2 ... Fn)
```

should be a parenthesized sequence of $n-1$ interpolants `(I1 I2 ... In-1)`, where

- $F_1 \wedge \neg I_1$ is unsat,
- $I_i \wedge F_i \wedge \neg I_{i+1}$ is unsat for $1 \leq i \leq n-2$, and
- $I_n \wedge F_n$ is unsat.
- I_i contains only the symbols that occur in one of the formulas $F_1, \ldots, F_{i-1}$ and in one of the formulas $F_i, \ldots, F_n$. Again, symbols that occur in any unmentioned formula (theory extension) are also allowed.

Tree Interpolants

Let $T = (V, E)$ be a tree with a labeling $F : V \to \mathit{term}$ that assigns to each vertex a formula. McMillan defines the tree interpolants [dMB] as a second labeling $I : V \to \mathit{term}$ that assigns to each vertex an interpolant. The condition for the interpolants are:

- If the children of v_i are $v_{j_1}, \ldots, v_{j_n}$, then

  $$I(v_{j_1}) \land \ldots \land I(v_{j_n}) \land F(v_i) \land \neg I(v_i) \text{ is unsat,}$$

  i. e., the formula of a vertex and the interpolants of its children imply the interpolant of the vertex.

- The root vertex r is labeled with the interpolant $I(r) = \texttt{false}$.

- The interpolant of a vertex v may only contain symbols that occur in some input formula in the subtree starting with node v and in some input formula that is not in the subtree starting with node v.

Tree interpolants are a generalization of nested interpolants [HHP10] and are useful for programs with procedures. They can also be useful for more general class of programs where verification conditions are expressible by Horn clauses. This includes parallel programs, object oriented programs and many more.

A sequence of interpolants can be seen as a special case of a tree interpolant for the tree

$$F_1 \longleftarrow F_2 \longleftarrow \cdots \longleftarrow F_n,$$

where F_n is the root of the tree and I_i is the interpolants of the vertex labeled with F_i. This suggests that the encoding of the trees should be such that the above tree is encoded by the sequence `F1 F2 ... Fn`. We propose the following EBNF to encode a tree:

$$\begin{aligned} \textit{tree} &::= \textit{symbol} \mid \textit{subtrees symbol} \\ \textit{subtrees} &::= \textit{tree} \mid \textit{tree}(\textit{subtrees}) \end{aligned}$$

The non-terminal *symbol* matches a name of a named formula. The first rule encodes a tree as either a single symbol (for a single-node tree) or as the encoding of the subtrees of the root vertex followed by the symbol labeling the root vertex. The encoding of the first child requires no parenthesis (thus yielding in the simple encoding for a sequence), but its siblings need to be parenthesized to encode the tree structure. Note that the grammar above is LALR(1) and should be easy to parse. The formulas appear in the same order as in a post-order traversal of the tree. E. g., to compute the interpolants for the tree depicted in Figure 1, the command is

```
(get-interpolants phi1 phi2 (phi3 phi4) phi5 phi6)
```

$$\begin{tikzcd}
\phi_1 & \phi_2 \arrow[l] & & & \phi_5 \arrow[lll] \arrow[dl] & \phi_6 \arrow[l] \\
& & \phi_3 & \phi_4 \arrow[l] & &
\end{tikzcd}$$

Figure 1: Example Formula Tree

The solver replies to this command with a parenthesized sequence of interpolants `(I1 I2 ...In-1)` that correspond to the post-order traversal of the tree. The last interpolant, which is the interpolant `false` annotated to the root vertex, is omitted from the returned sequence. The tree structure is also omitted from the output.

Combining Partitions

Sometimes it may be useful to combine two named formulas into a single partition without computing an intermediate interpolant between these formulas. Instead of a single name of a formula, SMTInterpol also allows to combine several formulas with `and`. E.g. to compute the interpolant for $A = \phi_1$ and $B = \phi_2 \wedge \phi_3$, the following command can be used:

```
(get-interpolants phi1 (and phi2 phi3))
```

Caveat. This may lead to confusion with the parenthesis used in tree interpolants. Since `and` cannot be used as name of a formula, the grammar is still unambiguous, but it is a bit more tricky to build an LALR grammar. Also note that this requires `and` to be a keyword, which it is not in the current SMTLIB 2 standard.

For this reason I will not strongly vote for including the `and`-extension in the standard. The use-cases for which we needed them are super-seeded by the tree interpolants.

Example

Next example shows the SMTLIB 2.0 counterpart of the previous example. Again, we compute two interpolants modulo the theory of integer linear arithmetic and uninterpreted functions extended with a two element sort U. The example also shows the intermixing of theory extension and interpolation problem. Answers from the solver are given below the command and are preceded by a semicolon.

```
(set-option :print-success false)
(set-option :produce-interpolants true)
(set-logic QF_UFLIA)
(declare-fun x_1 () Int)
(declare-fun xm1 () Int)
(declare-fun x2 () Int)
(declare-fun res4 () Int)
(declare-fun resm5 () Int)
(declare-fun xm6 () Int)
(declare-fun x7 () Int)
(declare-fun res9 () Int)
(declare-fun resm10 () Int)
(declare-fun res11 () Int)
(assert (! (<= x_1 100) :named M1))
(assert (! (= xm1 (+ x_1 11)) :named M2))
(assert (! (> x2 100) :named S11))
(assert (! (= res4 (- x2 10)) :named S12))
(assert (! (and (= x2 xm1) (= resm5 res4)) :named S1RET))
(assert (! (= xm6 resm5) :named M3))
(assert (! (> x7 100) :named S21))
(assert (! (= res9 (- x7 10)) :named S22))
(assert (! (and (= x7 xm6) (= resm10 res9)) :named S2RET))
(assert (! (= res11 resm10) :named M4))
(assert (! (and (<= x_1 101) (distinct res11 91)) :named ERR))
(check-sat)
;unsat
(get-interpolants M1 M2 (S11 S12) S1RET M3 (S21 S22) S2RET M4 ERR)
;((<= x_1 100)
; (<= xm1 111)
; true
; (<= res4 (- x2 10))
; (<= resm5 101)
; (<= xm6 101)
; (>= x7 101)
; (and (>= res9 91) (<= res9 (- x7 10)))
; (= resm10 91)
; (= res11 91))
```

Interpolation Info

It might be helpful to users to easily get information about the interpolation methods supported by the solver. Since we proposed three different methods, we also propose an information flag `:interpolation-method`. Possible return values are shown in the following table.

Return Value	Meaning
`unsupported`	Interpolation is not supported by the solver
`basic`	Only the binary `get-interpolants`-call is supported
`sequence`	`get-interpolants` can be used to compute sequences
`tree`	Tree interpolation is available

Rationale. If a solver does not support one of the extensions, users might be able to use a fallback. For example, if tree interpolants are used to model check programs with function calls, a series of sequence interpolation calls on different interpolation problems can be used to compute tree interpolants.

References

[dMB] Leonardo de Moura and Nikolaj Bjørner. Z3: Theorem prover.

[HHP10] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *POPL*, 2010.