The TSPASS System

**TSPASS** is an implementation of clausal temporal resolution based on the (classical) first-order theorem prover **SPASS**.

**TSPASS** actually implements clausal temporal resolution for *monodic* first-order temporal logic, but we will here deal only with the PTL aspects.

**TSPASS**
- uses the *simplified* clausal temporal resolution calculus
- is based on *ordered* resolution.
- uses **SPASS** underneath, not only to carry out step resolution and simplification operations, but also to automatically search for loops that can be used in the temporal resolution operation (as described above).

### Availability

**TSPASS** was implemented by Michel Ludwig and is currently available via

[http://www.csc.liv.ac.uk/~michel/software/tspass](http://www.csc.liv.ac.uk/~michel/software/tspass)

as well as at


### TSPASS in Action: Translation

We will first look at one component of TSPASS, namely **fotl-translate**, which translates arbitrary PTL formulae into SNF (actually into the temporal problem form).

Applying **fotl-translate** to "p => (always q)" gives (after some pre-amble) the output

```plaintext
list_of_symbols.
  predicates(_P,0),(_R,0),(p,0),(q,0).
end_of_list.
list_of_formulae(axioms).
  formula(or(not(p),_P)).
  formula(always(implies(_P,_R))).
  formula(always(implies(_R,next(_R)))).
  formula(always(implies(_R,q))).
end_of_list.
end_problem.
```
**TSPASS in Action: Translation**

- `list_of_symbols` defines the propositions we will see in later clauses. As well as `p` and `q`, both with arity 0, we see two new propositions, `_P` and `_R`.
- The translated SNF clauses are given in `list_of_formulae(axioms)`. Here, these four formulae correspond to:

  
  
<table>
<thead>
<tr>
<th>Clause</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code></td>
<td><code>¬p ∨ _P</code></td>
</tr>
<tr>
<td><code>_P</code></td>
<td><code>_R</code></td>
</tr>
<tr>
<td><code>_R</code></td>
<td><code>♢_R</code></td>
</tr>
<tr>
<td><code>_R</code></td>
<td><code>q</code></td>
</tr>
</tbody>
</table>

**Translating ‘Until’**

Next we translate an ‘U’ formula, by using “m => (p until q)” as input to

```
fotl-translate --extendedstepclauses
```

Note that the “--extendedstepclauses” flag here allows step-clauses to contain more than one literal on each side of the implication (the default is just one literal on each side).

Again, after some pre-amble, we get

```
list_of_symbols.
predicates{(_P,0),(_S,0),(_waitforq,0),(m,0),(p,0),(q,0)}.
end_of_list.
```

Then, the clauses...

```
list_of_formulae(axioms).
  formula(or(not(m),_P)).
  formula(always(implies(_P,or(p,q)))).
  formula(always(implies(_P,or(_S,q))))).
  formula(always(implies(_S,next(or(p,q))))).
  formula(always(implies(_waitforq,next(or(p,q))))).
  formula(always(sometime(not(_waitforq))).
end_of_list.
```

Or, in a more recognisable form:

```
<table>
<thead>
<tr>
<th>Clause</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start</code></td>
<td><code>¬m ∨ _P</code></td>
</tr>
<tr>
<td><code>_P</code></td>
<td><code>p ∨ q</code></td>
</tr>
<tr>
<td><code>_P</code></td>
<td><code>_S ∨ q</code></td>
</tr>
<tr>
<td><code>_S</code></td>
<td><code>♢(p ∨ q)</code></td>
</tr>
<tr>
<td><code>_S</code></td>
<td><code>♢(_S ∨ q)</code></td>
</tr>
<tr>
<td><code>_P</code></td>
<td><code>_waitforq</code></td>
</tr>
<tr>
<td><code>_waitforq</code></td>
<td><code>♢(_waitforq ∨ q)</code></td>
</tr>
<tr>
<td><code>true</code></td>
<td><code>♢false</code></td>
</tr>
</tbody>
</table>
```

**Simple Translation Example**

If we input `◊e` to `fotl-translate`, we get

```
list_of_formulae(axioms).
  formula(always(sometime(e))).
end_of_list.
```

©Michael Fisher  An Introduction to Practical Formal Methods Using Temporal Logic  | TEMPORAL RESOLUTION: IMPLEMENTATION | p.5/28
Translating ‘\( t \Rightarrow \Diamond s \)’

\[
\begin{align*}
\text{list_of_symbols.} \\
\text{predicates([}_P,0],[_\text{waitfor}f,0],[s,0],[t,0]).} \\
\end{align*}
\]

\[
\begin{align*}
\text{end_of_list.} \\
\text{list_of_formulae(axioms).} \\
\text{formula(or(not(t),}_P)). \\
\text{formula(always(implies(and(}_P,not(s)),_\text{waitfor}f))). \\
\text{formula(always(implies(}_\text{waitfor}f,\text{next(or(}_\text{waitfor}f,s))).} \\
\text{formula(always(sometime(not(_\text{waitfor}f)))).} \\
\text{end_of_list.} \\
\end{align*}
\]

This corresponds to the clause set

\[
\begin{align*}
\text{start} & \Rightarrow \neg t \lor \neg \_P \\
(\_P \land \neg s) & \Rightarrow _\text{waitfor}f \quad \_\text{waitfor}f \Rightarrow \Diamond(\_\text{waitfor}f \lor s) \\
\text{true} & \Rightarrow \Diamond \neg _\text{waitfor}f \\
\end{align*}
\]

Note: _\text{waitfor}f_ is generated even in simple examples.

Then, the clauses.... (1)

\[
\begin{align*}
\text{formula(always(and(}_P,\neg _P))). \\
\text{formula(always(implies(or(}_P,\neg(}_P),\_\text{waitfor}f))). \\
\text{formula(always(implies(}_\text{waitfor}f,\text{next(or(}_\text{waitfor}f,\_P))).} \\
\text{formula(always(implies(}_\text{waitfor}f,\text{next(}_\text{waitfor}f)).) \\
\text{formula(always(sometime(not(_\text{waitfor}f)))).} \\
\end{align*}
\]

Bigger Translation

Our final translation example is more complex.

We begin by taking a formula involving two eventualities (\( \Diamond f \) and \( \Diamond g \)) and produce a new set of clauses with only one (global) eventuality.

So, beginning with \( \square(\Diamond f \land \Diamond g) \) and running

\text{fotl-translate --extendedstepclauses --singleeventuality}

on input “always ((sometime \( f \) & (sometime \( g \)))”

we get a set of additional propositions:

\[
\begin{align*}
\{ & \_P,\_P1,\_P2,\_P3,\_P4,\_P5,\_P6,\_P7,\_U,\_l,\_\text{since_waitfor}f, \\
& \_\text{since_waitfor}f_1,\_\text{waitfor}_P6_2,\_\text{waitfor}_P6_1,\_\text{waitfor}_P6 \} \\
\end{align*}
\]

Then, the clauses.... (2)

\[
\begin{align*}
\text{formula(always(implies(}_P,\_P1))). \\
\text{formula(always(implies(or(}_P,\neg(}_P),\_\text{waitfor}f))). \\
\text{formula(always(implies(or(_P,\text{next(}_\text{waitfor}f)),\_\text{waitfor}f))).} \\
\text{formula(always(implies(}_\text{waitfor}f,\_\text{waitfor}f_1)).} \\
\text{formula(always(implies(}_\text{waitfor}f_1,\text{next(or(}_\text{waitfor}f_1,\_P))).} \\
\text{formula(always(implies(}_\text{waitfor}f_1,\_\text{waitfor}f))).} \\
\text{formula(always(implies(}_P,\_P))). \\
\text{formula(always(implies(}_P,\text{next(or(}_P,\_P))).} \\
\text{formula(always(implies(}_\text{waitfor}f,\_\text{waitfor}f)).} \\
\text{formula(always(implies(}_P,\text{next(or(}_P,\_P))).} \\
\text{formula(always(implies(}_P,\text{next(or(}_P,\_P))).} \\
\text{formula(always(implies(}_P,\_P))). \\
\text{formula(always(implies(}_P,\_P))). \\
\text{formula(always(implies(}_P,\_P))). \\
\text{formula(always(implies(}_P,\_P))). \\
\text{formula(always(implies(}_P,\_P))). \\
\text{formula(always(implies(}_P,\_P))). \\
\end{align*}
\]

In a clearer form:

\[
I = \{ \neg _\text{since_waitfor}f, \neg _\text{since_waitfor}f_1, \_l \} \\
E = \{ \Diamond _\text{since_waitfor}f, \Diamond _\text{since_waitfor}f_1, \_l \} \\
S = \{ \Diamond \neg _\text{waitfor}_P6_2 \}
\]
Then, the clauses.... (3)

\[
U = \\
(P \land P_1) \\
(P \land \neg f) \Rightarrow \_\text{waitforf} \\
(P_1 \land \neg g) \Rightarrow \_\text{waitfor}_1 \\
\_\text{waitforf} \land ((\_l \lor \neg \_\text{since_waitforf}) \lor \_P) \\
((\_\text{waitforf} \lor (\_\_l \land \_\text{since_waitforf})) \lor \_P) \\
(\_\text{waitfor}_1 \land ((\_l \lor \neg \_\text{since_waitforf}_1)) \lor \_P) \\
(\_\text{waitfor}_1 \land (\_\_l \land \_\text{since_waitforf}_1)) \lor \_P) \\
\_l \Rightarrow \_P_7 \\
\_U \Rightarrow (\_\text{since_waitforf}_1 \land \_\text{since_waitforf} \land \_l) \\
(\_P_7 \land \neg \_P_6) \Rightarrow \_\text{waitfor}_P_6 \_2
\]

This example does exhibit how quickly the translated version becomes difficult to understand.

TSPASS Step Resolution

We will show the validity of the PTL axiom

\[
\vdash (a \Rightarrow b) \Rightarrow (\Diamond a \Rightarrow \Box b).
\]

by negating, translating to SNF, and applying step resolution, all through TSPASS

Applying fotl-translate --extendedstepclauses to

\[
\Diamond (a \Rightarrow b) \land (a \land \neg \Diamond b)
\]
gives new propositions \_P, \_P1, and \_P2, and the temporal problem

\[
\begin{align*}
\text{formula(always (and (P, and (P_1, P_2))))}. \\
\text{formula(always (implies (P, next (or (not (a), b))))).} \\
\text{formula(always (implies (P_1, next (a))).)} \\
\text{formula(always (implies (P_2, next (not (b))))).}
\end{align*}
\]

Aside: Grouping Step Clauses.

Note that adding “--regroupnext” as an additional flag to the fotl-translate procedure produces the following set of clauses (the flag tries to group several 'next' operators together into one):

\[
\begin{align*}
\text{list_of_formulae(axioms).} \\
\text{formula(always (implies (true, next (_U))).)} \\
\text{formula(always (implies (_U, and (or (not (a), b)), and (a, not (b))))).}
\end{align*}
\]

Back to the proof....

If we simply feed the above (un-grouped) temporal problem to TSPASS, we get (amongst a lot of other detail)

\[
\begin{align*}
\text{TSPASS 0.92} \\
\text{SPASS beiseite: Unsatisfiable.} \\
\text{Problem: translate_step_axiom.out} \\
\text{TSPASS derived 5, backtracked 0, and kept 10 clauses.} \\
\text{Number of input clauses: 6} \\
\text{Number of eventualities: 0} \\
\text{Total number of generated clauses: 11}
\end{align*}
\]

So, the original set of clauses is unsatisfiable.
DocProof

Slightly more informative output can be generated using the "--DocProof" flag.

Here, output includes:

Here is a proof with depth 5, length 11:
1. [0:Inp:LS] || _P(U) *.
2. [0:Inp:LS] || _P1(U) *.
3. [0:Inp:LS] || _P2(U) *.
4. [0:Inp:LS] || _P1(U) -> a(temp_succ(U)) *.
5. [0:Inp] || _P2(U) b(temp_succ(U)) * -> .
6. [0:Inp:LS] || a(temp_succ(U)) * -> b(temp_succ(U)).
7. [0:Res:4.1,6.1:LS] || _P1(U) _P(U) b(temp_succ(U)) * -> .
9. [0:Res:1.0,8.1] || _P1(U) * _P2(U) -> .
11. [0:Res:3.0,10.0] || _P(U) a(temp_succ(U)) * -> b(tempsucc(U)).

The Refutation

1. true => _P
2. true => _P1
3. true => _P2
4. _P1 => Oa
5. _P2 => O~b
6. _P => O(−a v b)
7. (_P1 ^ _P) => Ob [4, 6 Step Resolution]
8. true => _P1 v _P v _P2 [5, 7 Step Resolution: Simplification]
9. true => _P1 v _P2 [1, 8 Resolution]
10. true => _P2 [2, 9 Resolution]
11. true => false [3, 10 Resolution]

Running TSPASS

Rather than running fotl-translate and TSPASS separately, a shell script, run-tspass.sh runs both while passing appropriate flags on to each one.

TSPASS Temporal Resolution (1)

Now we will move on to some examples involving temporal resolution, and therefore some loop detection.

Consider the following input to TSPASS describing an unsatisfiable set of clauses:

always(a => next a)
& always(a => next ~m)
& always(some(m) a)

Running TSPASS does indeed indicate unsatisfiability.
**TSPASS Temporal Resolution (2)**

Again, more informative output can be provided using the "--DocProof" flag. Here, the output includes:

```
Here is a proof with depth 1, length 3:
1[0:Inp:LS] || -> a(temp_zero)*.
15[0:LoopSearch::LS] || a(U)* -> .
17[0:Res:1.0,15.0] || -> .
```

Thus, the proof involves

1. 'a' is true at the start, thus a(temp_zero)
2. an 'a' anywhere leads to a loop in contradiction to our global eventuality, so we must add '¬a' to U, and
3. straightforward resolution between (1) and (15) gives false.

---

**Bigger Loops**

Examples with conjunctive loops, e.g., \((a \land b) \Rightarrow \lozenge \neg m\) are not so informative since TSPASS typically replaces "a \land b" by a new proposition and then, in the proof produced, we just see a loop in that new proposition.

Similarly, step clauses such as "x \Rightarrow \lozenge(b \lor c)" are typically renamed to "x \Rightarrow \lozenge y" with "y \Rightarrow (b \lor c)" being added to U.

So, it is sometimes hard to see exactly what is happening with the original propositions.

Thus, trying the negation of our induction axiom, i.e.,

\[ p \land \Box(p \Rightarrow \lozenge p) \land \Diamond \neg p \]

within TSPASS simply gives a loop in p, then a straightforward contradiction with "start \Rightarrow p".

---

**TSPASS Speed**

A final comment about TSPASS is that it is both fast and reliable, and so provides an appropriate tool both for learning about temporal proof and for tackling more sophisticated applications.

Of course, we mean “fast” for a temporal prover!

---

**Model Construction**

The idea with refutation-based systems is that we are usually hoping to find a contradiction.

In such a case, this means that the clauses are unsatisfiable and, typically, that the unnegated original formula is valid.

So, often, a contradiction is what we desire and, in such a case, TSPASS will provide a ‘proof’ showing how the contradiction was derived.

However, what if the expected contradiction is not found? What shall we do then? Fortunately, TSPASS also provides us with a way to visualize a satisfiable set of clauses.

Again, rather than explaining the underlying theory, let us look at an example.
Example (1)

We wanted to show how TSPASS established that the formula
\[(\Diamond p \land \Box(p \Rightarrow \Diamond p)) \Rightarrow \Box \Diamond p\]
is valid.

This was to be an example of the typical PTL induction schema: \(p\) would eventually be true; whenever \(p\) is satisfied then \(p\) would eventually be true again; and so we know \(p\) will be true infinitely often.

So, we input
\[(\text{sometime } p) \land \text{always}(p \Rightarrow \text{sometime } p) \land (\text{sometime always } \neg p)\]
to TSPASS, expecting a contradiction.

However, TSPASS reported that the formula is satisfiable!

Example (2)

Fortunately, TSPASS has the ability to construct (finite) models of satisfiable PTL formulae.

So, calling TSPASS with the flag “--ModelConstruction” the following output was produced describing a simple reduced model for the input formula:

```
Temporal model constructed:
------------------------ 0
{}
------------------------ 1
{p}
------------------------ 2
{}
------------------------> 2
```

From Models to Execution

Thus the *model construction* aspect of TSPASS is very useful.

As we will see elsewhere, constructing a model from a temporal formula is very productive even beyond the clausal resolution approach.

Indeed, such model construction can also be seen as *execution* of the base formula.

Example (3)

So, \(p\) can be false in state 0, true in state 1 and then false again in state 2. After that point the transitions loop on state 2. This tells us that \(p\) just occurring once (e.g. in state 1) is enough to satisfy the formula. When we look back at the formula this becomes clear.

\[(\Diamond p \land \Box(p \Rightarrow \Diamond p)) \Rightarrow \Box \Diamond p\]
is not valid after all. The sub-formula “\(p \Rightarrow \Diamond p\)” is trivially true, so we are left with “\(\Diamond p \Rightarrow \Box \Diamond p\)” which is certainly not valid!

Our mistake was that we missed out a “\(\Box\)” operator. The real formula we meant was

\[(\Diamond p \land \Box(p \Rightarrow \Box \Diamond p)) \Rightarrow \Box \Diamond p\]