Temporal Specification

[Programming Language Semantics]  
Michael Fisher

Department of Computer Science, University of Liverpool, UK

[MFisher@liverpool.ac.uk]

An Introduction to Practical Formal Methods Using Temporal Logic

Simple Programming Language

Consider a (very) simple imperative programming language:

\[ \text{x:=v} \]

\[ \text{...... after this statement, variable x has the value v} \]

\[ v \text{ comprises integers, variables or arithmetic expressions} \]

\[ S1; S2 \]

\[ \text{.......... execute statement } S2 \text{ after executing } S1 \]

\[ \text{if Test then } S1 \text{ else } S2 \]

\[ \text{...... if Test is true, execute } S1 \text{, otherwise execute } S2 \]

\[ \text{while Test do } S \]

\[ \text{...... while Test is true, continue executing } S \]

\[ \text{end} \]

\[ \text{......... terminate the program} \]

Example Assignments

The following assignments are legal statements in our programming language, principally because

- \( y, \text{zz}, \text{and wow} \) are all legal variables, and
- \( 2, y-1, \text{and zz+y} \) are all legal expressions.

\[ y := 2 \]

\[ \text{zz := y-1} \]

\[ \text{wow := zz+y} \]

Example Statements

\[ \text{if (y>3) then } \text{zz := y else } \text{zz := 2} \]

\[ \text{while ( (y<3) & (zz=2) ) do } \text{zz := y-1} \]

\[ \text{if (wow=7) then} \]

\[ \text{................. (while (wow>6)} \]

\[ \text{do } \text{wow := wow-1)} \]

\[ \text{else } \text{wow := y+7} \]
Example Program

\[
x := 0;
\]
\[\text{while } (x < 4) \text{ do } x := x + 1; \]
end

What should happen to the value of ‘\(x\)?’

Semantics (end)

Now we will look at some temporal semantics for each of the constructs within our language, beginning with ‘end’.

\[[\text{end}] = \text{true}\]

Simply finish the program and allow unconstrained behaviour afterwards.

Aside: In reality we would likely need to say that all variables retain their values once the program has finished.

Semantics (if...then...else)

Semantics of “if...then...else” is simply

\[
[(\text{if Test then } S_1 \text{ else } S_2); S_3] \equiv 
\text{(Test } \Rightarrow [S_1;S_3]) \land ((\neg \text{Test} \Rightarrow [S_2;S_3])
\]

We here assume that Test is a simple Boolean test that is easily evaluated.

So, if the Test is true, do the ‘then’ branch \((S_1;S_3)\).

If the Test is false, do the ‘else’ branch \((S_2;S_3)\).

Example: if...then...else

Consider:

\[
(\text{if } (x > 2) \text{ then } y := 3 \text{ else } y := 5); \ S
\]

Using the earlier rule, the semantics of this is

\[
[(\text{if } (x > 2) \text{ then } y := 3 \text{ else } y := 5); \ S] \\
= \\
((x > 2) \Rightarrow [y := 3; \ S]) \land ((x \leq 2) \Rightarrow [y := 5; \ S])
\]
**Semantics (while)**

The semantics of a `while` statement is reduced to that of a recurring `if` statement, with the original `while` statement embedded within it.

\[
[(\text{while Test do S1); S2}] = 
[(\text{if Test then S1;(while...)else S2);end}]
\]

**Note.** Examining the semantics of ‘while’ above, we see that we will generate a temporal fixpoint formula.

Indeed, the temporal semantics of both loops and recursive procedures are intimately linked to fixpoints.

---

**Semantics of Assignment**

Perhaps surprisingly, the semantics of this intuitively simple operation is quite subtle and requires a complex semantic description.

Why is this the case? Well, in `x:=v` if `v` is a simple value, then the semantics is straightforward, for example

\[
[x:=2] = \Box(x = 2)
\]

However, if the expression `v` involves the current value of `x` then we must describe how this is stored/accessed.

In implementing `x:=x+1`, we must record the previous value of ‘x’, add one to it, and then assign this to the variable ‘x’.

---

**Example: while**

Consider:

\[
(\text{while (x<4) then x:=x+1); S}
\]

The semantics of this is

\[
[(\text{while (x<4) then x:=x+1); S}] = 
((x < 4) \implies [x:=x+1; ((\text{while ...})]) \land
((x \geq 4) \implies [S])
\]

---

**Semantics (full assignment, 1)**

\[
[x:=v; S] = \exists w. x = w \land \Box(x = v(x/w) \land [S])
\]

Note here that `v(x/w)` is the expression `v` with occurrences of `x` replaced by `w`.

Thus, if `v` is `x + 3 + (2x)`, then `v(x/w)` is `w + 3 + (2w)`.

Two examples of the assignment semantics are as follows.

1. \[
[x:=x+1] = \exists w. (x = w) \land \Box(x = (w + 1))
\]

If `x` has the value `w` in the current moment, it will have the value `w + 1` in the next moment.

Here, `w` is used to store the current value of the variable so it is not lost in the over-writing process.
Semantics (full assignment, 2)

\[ x := v; S \]  \[=\] \[ \exists w. x = w \land (x = v(x/w) \land [S]) \]

2. \( [x := 2] = \exists w. x = w \land \bigcirc(x = 2) \)

As \( v \) is a variable it must always have a value, i.e.,

\[ \square(\exists a. v = a) \]

so the above can be reduced simply to

\[ [x := 2] = \bigcirc(x = 2) \]

i.e. in the next moment in time, \( x \) will have the value 2.

Example: assignment

Consider

\[ x := 5; x := x \ast 4; \text{ end} \]

In stages, we develop the semantics for this as

\[ [x := 5; x := x \ast 4; \text{ end}] \]
\[ \circ((x = 5) \land [x := x \ast 4; \text{ end}]) \]
\[ \circ((x = 5) \land \exists w. (x = w) \land \bigcirc((x = (w \times 4)) \land [\text{end}])) \]
\[ \circ((x = 5) \land \exists w. (x = w) \land \bigcirc((x = (w \times 4)) \land \text{true}) \]
\[ \circ((x = 5) \land \bigcirc((x = (5 \times 4)) \land \text{true}) \]
\[ \circ((x = 5) \land \bigcirc(x = 20)) \]

Full Semantic Expansion!

\[ \circ((x = 0) \land [\text{while ...}]) \]
\[ \circ((x = 0) \land ((x < 4) \Rightarrow [x := x + 1; \text{ while ...}]) \land (x \geq 4) \Rightarrow [\text{end}])) \]
\[ \circ((x = 0) \land [x := x + 1; \text{ while ...}]) \]
\[ \circ((x = 0) \land \bigcirc((x = 1) \land [\text{while ...}])) \]

Semantics of Earlier Example

Recall the program:

\[
\begin{align*}
x &:= 0; \\
\text{while } (x < 4) \text{ do } x &:= x + 1; \\
&\text{end}
\end{align*}
\]

Now the semantics of \[ [x := 0; \text{ while ...}] \] is as follows:

\[ \bigcirc((x = 0) \land [\text{while ...}]) \]
\[ \bigcirc((x = 0) \land ((x < 4) \Rightarrow [x := x + 1; \text{ while ...}]) \land (x \geq 4) \Rightarrow [\text{end}])) \]
\[ \bigcirc((x = 0) \land [x := x + 1; \text{ while ...}]) \]
\[ \bigcirc((x = 0) \land \bigcirc((x = 1) \land [\text{while ...}])) \]

.........................
Proving Properties

Once we have the temporal formula \([P]\) describing the behaviour of our program \(P\), then we might wish to prove properties of this.

In our temporal proof system, we might try to prove that the semantics of the program satisfies some property, \(\varphi\), i.e.

\[ \vdash [P] \Rightarrow \varphi. \]

If we are sure that our temporal semantics captures all the behaviours of programs in this language, then establishing the above means that

*all possible executions of \(P\) satisfy the property \(\varphi\).*

Example: Safety Property

Typical properties we might want to show for our temporal specification include *safety* properties, i.e.

*something bad will not happen*

For our example program,

\[
\begin{align*}
  & x := 0; \\
  & \text{while } (x < 4) \text{ do } x := x + 1; \\
  & \text{end}
\end{align*}
\]

a typical safety property might be that variable \(x\) never has a value greater than 8.

In this case, we try to prove

\[ \vdash [P] \Rightarrow \neg (x > 8) \]

Example: Liveness Property

We might also want to prove *liveness* properties, i.e.

*something good will happen*

For our example program,

\[
\begin{align*}
  & x := 0; \\
  & \text{while } (x < 4) \text{ do } x := x + 1; \\
  & \text{end}
\end{align*}
\]

a typical liveness property might be that variable \(x\) eventually reaches the value 4.

In this case, we try to prove

\[ \vdash [P] \Rightarrow \lozenge (x = 4) \]

Where To Next?

Now that we have some idea how to describe (at least simple) programs using temporal formulae, we will turn to:

1. *how we might use temporal formulae as specifications of programs and how we might refine these specifications as part of program development;*
2. *if we have specifications of several distinct programs, then what is the temporal formula describing the system where the two programs are run concurrently;*
3. *what temporal specification do we obtain once we allow such concurrent programs to communicate?*

So, throughout the next part, we must be conscious of what varieties of communication and concurrency our specified programs aim to use.