Path condition, program point reachability

Anne Pacalet

June 13, 2006

1 Goal

The goal of this paper is to find a way to slice a program with respect to a program point reachability. The idea is to compute a precondition that is true whenever the given point is reached, and to use it to slice out the branches that are not used when this precondition is true.

Let's try to prove that this approch is correct.

2 Notations and definitions

S	:	a sequence of statements.
ϕ	:	a property.
P	:	a program point.
S_P	:	a sequence containing a program point P.
$S[P:\phi]$	:	ϕ is true each time that P is reached.
$S(x)$	:	execution of sequence S with inputs x.
$[\phi]S$	:	executions of sequence S with inputs satisfying ϕ.
	:	ie. $\{S(x).\phi(x)\}$.
$S(x) \twoheadrightarrow P:\phi$	:	$S(x)$ reach P and $S[P:\phi]$
$S(x) \twoheadrightarrow P$	:	$S(x)$ reach P (ie. $S(x) \twoheadrightarrow P : \mathit{true}$)
$S(x) \not\twoheadrightarrow P$	:	$S(x)$ doesn't reach P
$[\phi]S \twoheadrightarrow P$	:	$\forall x.\phi(x) \implies S(x) \twoheadrightarrow P$

We represent by . the program point which is at the end of a given sequence. So, with the above definitions :

$S(x) \twoheadrightarrow .$	:	$S(x)$ ends.
$S[.:\phi]$	:	ϕ is true after S if S ends (we will also write it $S[\phi]$)
$S(x) \twoheadrightarrow .:\phi$	:	$S(x) \twoheadrightarrow . \wedge S[\phi]$.

3 Path condition vs. reachability

To compute a precondition that is equivalent to the reachability of a program point, one can use WP as long as the considered sequence doesn't contain any loop, or abnormal terminaison.

In the other cases - ie. most of the cases - we will only be able to compute some over or under approximations.

Let's see :

- why WP doesn't fit our needs,
- and two ideas to compute something more appropriate.

3.1 Traditional *WP*

Usual *WP* definition tells that if S is executed with inputs that are satisfying the precondtion given by $WP(S, \phi)$, then ϕ is true if S terminates, ie. with the notations defined above :

$$[WP(S, \phi)]S[\phi]$$

Unfortunatly, we are not always able to compute the **weakest** precondition. We then try to compute an approximation such that the above property is true. In fact, *false* would be correct as a precondition ensuring ϕ, but it wouldn't be very usefull ! That is why we are trying to find a **weaker** precondition. But because we are doing an approximation, it might be the case that :

$$\exists x. \neg WP(S, \phi)(x) \wedge S(x)[\phi]$$

So, if we are interested in **all** the executions that reach a point, we cannot use traditional *WP* computation.

By the way, providing annotations on loops with respect to a point reachability would be difficult.

Example :

```
if (x > 0)
  S1;
else if (y > 0)
  S2;
while (c) S3;
if (x > 0 || y> 0)
  P: S4;
```

One can annotate the loop with an invariant stating that $x > 0$. In fact, it is true that if $x > 0$, the point P is reached. Using this precondition, we can decide to slice out the branch with $S2$. But in fact, there are some execution of this sequence with $x \leq 0 \wedge y > 0$ that also reach P...

What would be the definition of a *correct* annotation with respect to a point reachability ?

3.2 Path condition

WP can be used to compute a path condition. We define $MPC(S, P)$ as a precondition of S such that P is reached for all the executions that satisfy it, ie.

$$[MPC(S, P)]S \twoheadrightarrow P$$

Notice that some executions of S with inputs NOT satisfying $MPC(S, P)$ may also reach P.

$MPC(S, P)$ could be define like this :

$$\begin{aligned} MPC(S1; P : S2, P) &= WP_M(S1, \mathit{true}) \\ MPC(S; S_P, P) &= WP_M(S, MPC(S_P, P)) \\ MPC(\mathit{if}\ (c)\ \mathit{then}\ S_P\ \mathit{else}\ S, P) &= c \wedge MPC(S_P, P) \\ MPC(\mathit{if}\ (c)\ \mathit{then}\ S\ \mathit{else}\ S_P, P) &= \neg c \wedge MPC(S_P, P) \end{aligned}$$

with $WP_M(S, \phi)$ that must give a precondition of S such that :

$$[WP_M(S, \phi)]S \twoheadrightarrow P : \phi$$

This WP_M is a kind of traditional *WP* that deals with terminaison.

But what about the loops ? Maybe something like this :

$$MPC(\mathit{while}(c)S_P, P) = c \wedge (MPC(S_P, P).....)$$

Anyway, path conditions are not directly useful for what we are trying to do.

3.3 Reachability 1

We define $RC(S, P)$ as a precondition of S that must be true for every input set x such that $S(x) \twoheadrightarrow P$, ie.

$$S(x) \twoheadrightarrow P \implies RC(S, P)(x)$$

In other words :

$$[H]S \twoheadrightarrow P \implies (H \implies RC(S, P))$$

Notice that $RC(S, P) = \mathit{true}$ satisfies the property. But of course, we will try to find the strongest condition that we can, because the stronger the property is, the more we will be able to delete.

3.4 Reachability 2

We can also define $nPC(S, P)$ as a precondition of S that garanties that P is NOT reached, ie.

$$[nPC(S, P)]S \nrightarrow P$$

So, $\neg nPC(S, P)$ can be used as $RC(S, P)$.

Is it easier to use ? to explore...

4 Slicing

Let's see how those functions can be used for slicing. But first of all, what do we means by slicing ?

4.1 Slicing with respect to a precondition

We define $\mathit{slicePre}(S, \phi)$ to be a slice of sequence S with respect to the precondition ϕ so that :

$$\forall x.\phi(x) \implies S(x) \equiv (\mathit{slicePre}(S, \phi))(x)$$

meaning that every execution of S with inputs that satisfy ϕ is equivalent to the execution of $\mathit{slicePre}(S, \phi)$ with the same inputs. By equivalent ($\equiv$), we mean that the two executions have the same trace.

So, for $S = s_1; ...s_i; ...s_n;$, we define $S' = \mathit{slicePre}(S, \phi) = s'_1; ...s'_i; ...s'_n;$ such that :

$$\forall i.s'_i = s_i \vee (s'_i = \texttt{skip} \wedge [\phi]S \nrightarrow P_{s_i})$$

What do we want to do for terminaison ?...

4.2 Slicing with respect to a program point

We want to prove that if we use RC as a precondition to slice a sequence, we remove statements that cannot be involved in an execution that reaches P.

Let :

```
Pc: if (c) Pt: St; else Pe: Se;
```

be a part of the sequence S such that :

$$\forall x.RC(S, P)(x) \implies S(x)[Pc : c]$$

Then :

$$\forall x.RC(S, P)(x) \implies S(x) \nrightarrow Pe$$

The definition of RC says that :

$$\forall x.S(x) \twoheadrightarrow P \implies RC(S,P)(x)$$

Then we can conclude that :

$$[RC(S,P)]S[Pc:c] \implies (\forall x.S(x) \twoheadrightarrow P \implies S(x) \nrightarrow Pe)$$

4.3 RC computation

Now that we have seen that RC could be useful for slicing, let's try to find an algorithm to compute it.

4.3.1 RC algorithm

Notice that $RC(S,P)$ is only defined if the program point P is in S.

Warning : we first only deal here with a simple language with loops, but without jumps. It means that when the sequence S is `{S1; P:S2; }`, P is reached each time that S1 terminates, and there is no way to go from S2 to P (except of course if S is embeded in a loop).

Let's define a RC computation and check that it meets RC property, which is :

$$S_P(x) \twoheadrightarrow P \implies RC(S_P,P)$$

In those definitions, we use $WP_m(S,\phi)$ that must give a precondition of S such that :

$$S(x) \twoheadrightarrow . : \phi \quad \implies \quad WP_m(S,\phi)(x)$$

and what we will define later on.

- $RC(\texttt{\{S1; P:S2\}},P) = WP_m(S1,true)$

$$\begin{array}{lll} \texttt{\{S1; P:S2\}} \twoheadrightarrow P & \implies & \texttt{\{S1; P:\}} \twoheadrightarrow P \\ & \implies & S1 \twoheadrightarrow . : true \\ & \implies & WP_m(S1,true) \\ & \implies & RC(S1;P:S2,P) \end{array}$$

- $RC(\texttt{\{S;S\_P\}},P) = WP_m(S,RC(S_P,P))$

$$\texttt{\{S;S\_P\}} \twoheadrightarrow P \quad \implies \quad \exists\phi.S \twoheadrightarrow . : \phi \wedge [\phi]S_P \twoheadrightarrow P$$

We know that :

$$[\phi]S_P \twoheadrightarrow P \implies (\phi \implies RC(S_P,P))$$

and :

$$S \twoheadrightarrow . : \phi \implies WP_m(S,\phi)$$

WP_m has the following property :

$$WP_m(S,\phi_1) \wedge (\phi_1 \implies \phi_2) \implies WP_m(S,\phi_2)$$

because :

$$S \twoheadrightarrow . : \phi_1 \wedge (\phi_1 \implies \phi_2) \quad \implies \quad S \twoheadrightarrow . : \phi_2$$

We then have :

$$(\phi \implies RC(S_P, P)) \wedge WP_m(S, \phi) \implies WP_m(S, RC(S_P, P))$$

So finally :

$$\begin{array}{rcl} \texttt{\{S;S\_P\}} \twoheadrightarrow P & \implies & WP_m(S, RC(S_P, P)) \\ & \implies & RC(S; S_P, P) \end{array}$$

- $RC(\texttt{if (c) S\_P else S}, P) = c \wedge RC(S_P, P)$

$$\begin{array}{rcl} \texttt{\{ if (c) S\_P else S \}} \twoheadrightarrow P & \implies & c \wedge S_P \twoheadrightarrow P \\ & \implies & c \wedge RC(S_P, P) \\ & \implies & RC(\texttt{if (c) S\_P else S}, P) \end{array}$$

The proof is similar for the sequence `if (c) S else S_P`.

- $RC(\texttt{while (c) S\_P;}, P)$?

This is the most difficult part because P can be reached at any execution of S_P. A quite weak condition to reach P is first to enter the loop. But let's try to find a stronger one.

Let :

- W be the sequence `{ while (c) S_P; }`
- v be the set of the variables involved (read or write) in the loop,
- v_0 be the values of these variables when we enter the loop,
- and v_i their values at the i-th iteration of the loop.

We know that, by definition, $RC(S_P, P)$ is a precondition of S_P that is true each time P is reached. Let p be an iteration where P is reached. We then have $(RC(S_P, P))(v_p)$:

$$\begin{array}{rcl} \texttt{\{ while (c) S\_P; \}} \twoheadrightarrow P & \implies & c \wedge \exists p. S_P(v_p) \twoheadrightarrow P \\ S_P(v_p) \twoheadrightarrow P & \implies & (RC(S_P, P))(v_p) \end{array}$$

It can be the case that :

$$\forall i, j. RC(S_P, P)(v_i) = RC(S_P, P)(v_j)$$

This is true for instance when the variables involved in $RC(S_P, P)$ are not modified in the loop. In that case,

$$RC(S_P, P)(v_0) = RC(S_P, P)(v_p)$$

and then :

$$(\exists p. S_P(v_p) \twoheadrightarrow P) \implies RC(S_P, P)(v_0)$$

So, in that particular case, we can define :

$$RC(W, P) = c \wedge RC(S_P, P)$$

To deal with a more general case, let's define a function γ which take a property ϕ and a set of variables v and that returns ϕ_I such that :

$$(\phi \implies \phi_I) \wedge (\forall v_i, v_j . \phi_I(v_i) = \phi_I(v_j))$$

which means that $\gamma(v, \phi)$ provide a property that is not modified by the loop, and that is weaker than ϕ.

If we have a property ϕ that is true for the iteration p, the property $\gamma(w, \phi)$, where w is the set of variables that are modified in the loop, is true at any iteration, and for instance at the first one.

We then have :

$$\begin{array}{rcl} W \twoheadrightarrow P & \implies & (RC(S_P, P))(v_p) \\ & \implies & (\gamma(w, RC(S_P, P))(v_p) \\ & \implies & (\gamma(w, RC(S_P, P))(v_0) \end{array}$$

so we can define :

$$RC(W, P) = c \wedge \gamma(w, RC(S_P, P))$$

The difficult part will then be to find an algorithm to remove a part of a property that depends on some variables, without loosing to much information (see §4.3.3).

4.3.2 WP_m computation

We need to compute $WP_m(S, \phi)$ that is a precondition of S such that :

$$S(x) \twoheadrightarrow . : \phi \quad \implies \quad WP_m(S, \phi)(x)$$

or in other words :

$$[H]S \twoheadrightarrow . : \phi \quad \implies \quad (H \implies WP_m(S, \phi))$$

For statements without loops, we can use traditional WP :

- Assignment : $WP_m($`y = exp;`$, \phi) = \phi_{y \leftarrow exp}$
  is correct because :

  $$[H]\{\texttt{ y = exp; }\} \twoheadrightarrow . : \phi \implies (H \implies \phi_{y \leftarrow exp})$$

  and then :

  $$\{\texttt{ y = exp; }\} \twoheadrightarrow . : \phi \implies WP_m(\texttt{y = exp;}, \phi))$$

  Notice that this is correct even if the sequence doesn't terminate because the first part of the implication is then *false*.

- `if-then-else` : $WP_m($`if(c) S1; else S2;`$, \phi) =$ *if* c *then* $WP_m(S1, \phi)$ *else* $WP_m(S1, \phi)$
  is obviously correct because :

  $$\begin{array}{rcl} (\texttt{if(c) S1; else S2;} \twoheadrightarrow . : \phi & \implies & ([c]S1 \twoheadrightarrow . : \phi) \wedge ([\neg c]S2 \twoheadrightarrow . : \phi) \\ & \implies & (c \implies WP_m(S1, \phi)) \wedge (\neg c \implies WP_m(S2, \phi)) \end{array}$$

- for a sequence W with a loop : `{ while (c) S; }`, we want to find a precondition that is true each times W ends satisfying ϕ.

  To have $(W \twoheadrightarrow . : \phi)$:

– either c is initially false and ϕ initially true,

– or the loop is executed and there exists an iteration p (the last one) such that $S \twoheadrightarrow . : (\neg c \wedge \phi)$

$$S(v_p) \twoheadrightarrow . : (\neg c \wedge \phi)(v_p) \quad \Longrightarrow \quad WP_m(S, (\neg c \wedge \phi)(v_p)$$

The idea is to find a property which is invariant ϕ_I in the loop, and which is implied by this WP_m. So, let's use $\gamma(w, WP_m)$ again.

We can then define :

$$WP_m(W, \phi) = (\neg c \wedge \phi) \vee (c \wedge \gamma(w, WP_m(S, (\neg c \wedge \phi))))$$

4.3.3 The γ function

As I said before, the difficult part with this γ function is to find a transformation that doesn't lose to much information. Let's remind the property that γ has to satisfy :

$$(\phi \implies \gamma(w, \phi)) \wedge (\forall w_i, w_j . \gamma(w, \phi)(v_i) = \gamma(w, \phi)(v_j))$$

We write $\mathcal{V}(\phi)$ the set of the variables involved in ϕ. So γ property can be rewritten :

$$(\phi \implies \gamma(w, \phi)) \wedge \mathcal{V}(\gamma(w, \phi) \cap w = \emptyset)$$

We know that the property γ is applied to is the result of a WP_m computation (can we use that ?).

But let's consider a general property composed with expressions E, and logical operators $\wedge$, $\implies$, $\neg$, $\vee$.

$$\forall \phi . \mathcal{V}(\phi) \cap w = \emptyset \implies \gamma(w, \phi) = \phi$$

In the other cases :

$$\begin{aligned} \gamma(w, E) &= \mathit{true} \\ \gamma(w, \phi_1 \wedge \phi_2) &= \gamma(w, \phi_1) \wedge \gamma(w, \phi_2) \\ \gamma(w, \phi_1 \vee \phi_2) &= \gamma(w, \phi_1) \vee \gamma(w, \phi_2) \\ \gamma(w, \neg\phi) &= \mathit{true} \end{aligned}$$

We can deduce that :

$$\begin{aligned} \gamma(w, \phi_1 \implies \phi_2) &= \gamma(w, \neg\phi_1 \vee \phi_2) \\ &= \gamma(w, \neg\phi_1) \vee \gamma(w, \phi_2) \\ &= \phi_1 \implies \gamma(w, \phi_2) \text{ if } (\mathcal{V}(\phi_1) \cap w = \emptyset) \\ &\mathit{or} \quad \mathit{true} \text{ otherwise} \end{aligned}$$

4.4 Slicing algorithm

The point is now to see how to compute a slice. The precondition gives a context of execution, and we want to remove the branches that are NEVER used in that context.

4.4.1 How to slice using a precondition

4.4.2 How to use the point to get better results

The previous method can be used with any precondition. But in the special case when the precondition is computed with RC to ensure a point reachability, one can use this information to get better results. For instance, if we want to reach P in the sequence :

```
if (c) P: S1; else S2;
```

the `else` branch can be removed without any computation if the whole `if` is not in a loop.

... to continue...

4.4.3 Proof of correctness

5 How to use more than one program point