1 Property-irrelevant predicates

Loops (and recursions) are major hurdles in scalability of property verification tools (verifiers). Although slicing removes loops which have no bearing on assert statement in terms of its value and reachability, sliced programs still have loops challenging scale up of the verifiers. If we can transform a program by eliminating some of such loops, it is more likely that a given verifier succeeds on transformed program. Of course the transformation will be useful only if results on transformed program can be used to get results on original program.

Loops existing in a (backward) sliced program, may compute value of some variables that impact outcome of assert expression in following ways:

1. Impact on value of assert expression, possibly through a chain of assignments.

2. Impact on value of some predicates, possibly through a chain of assignments, that
   (a) Value impact the assert statement
   (b) Influence the reachability of assert statement

Obviously, loops of type (1) can not be eliminated, as they directly impact the value of assert expression. However, loops of type (2) can be eliminated if property can be checked even after abstracting the predicates which these loops are impacting. Since we want the transformed program to be useful in deciding the outcome on original program, ideally we would like the transformed program to be property equivalent to original program. So we focus on what kind of predicates can be abstracted so that loops and computations contributing to value of such predicates can be eliminated.

Let $C$ be a predicate in a given program $P$ in which a property is encoded through an assert statement $A$. Let $P'$ be an abstract program obtained from $P$ by replacing right hand side of reaching definitions used for $C$, with a non deterministic value, denoted by ‘*’.

Following observations are obvious:

1. If predicate $C$ is loop invariant in concrete program $P$ then so it will be in abstract program $P'$ too.
2. If property holds in abstract program \( P' \) then it will hold in concrete program \( P \) also

3. If property gets violated in abstract program \( P' \) then it may or may not get violated in concrete program \( P \).

Predicate \( C \) is called irrelevant to property (ITP), if abstract program \( P' \) is property equivalent to concrete program \( P \).

Actually if one picks up any predicate \( C \) from concrete program \( P \) and generate abstract program \( P' \) in the manner mentioned above then case (1) and (2) will always hold. It is the case (3) which differentiates an arbitrary predicate from an ITP predicate. For predicate \( C \) to be ITP, in case (3), property should get violated in concrete program \( P \) also.

2 Example

In figure [1] we show a program annotated as (Concrete). If we abstract the predicate \( (t > 100) \) we will see that in resulting abstract program, annotated as (AbsProg1), property will hold and therefore \( (t > 100) \) is an ITP predicate.

Considering predicate \( (st==1) \), we observe that in concrete program, \( st \) is loop invariant for outer while loop. We abstract this predicate also as explained above and
resulting abstract program is shown in Figure 1 with annotation (AbsPrg2). It is obvious that the predicate \((st==1)\) is loop invariant in abstract program too. The property holds on this abstract program also and therefore \((st==1)\) is an ITP predicate.

Suppose we change the assignment to \(j\) at line 16 to \(j+=3\) in concrete program. Now the assert will be violated in modified abstract program (AbsProg2), when one assigns a suitable value at non-deterministic assignment to \(st\) which makes predicate \((st==1)\) as false. And if an input exists with which \(st\) is assigned value 0 then it will get violated in concrete program also. It will not get violated in concrete program only if for no input \(st\) gets value 0. Which means predicate \(st==1\) always remains true in concrete program.

3 A refined definition of ITP predicates

To ensure that loops computing the value of an ITP predicate get eliminated, it will be good to place the non-deterministic assignments at only one point for a predicate rather than doing so at all reaching definitions individually. We choose this point as the nearest common post dominator of these reaching definitions. We claim that such a point exists and is unique. Without loss of generality, we assume that this point will be on a straight line segment of CFG(single entry, single exit). Let us call such a point as computing point of predicate \(C\) and denote it as \(\hat{C}\). In earlier example, we showed that when property holds in concrete program due to the predicate \(C\) being constant, say true, it may not hold in abstract program because the predicate \(C\) can be made false also in abstract program. Such predicates can never be ITP as per the earlier definition. Considering this fact, we refine our definition of ITP predicates using the modified abstraction strategy.

Let \(C\) be a predicate in a given program \(P\) and let \(A\) be an assertion encoding a property to be checked in program \(P\). Let \(P'\) be an abstract program obtained from (concrete) program \(P\) by inserting non-deterministic assignments at predicate computing point \(\hat{C}\), for variables used in predicate \(C\). The scheme is shown diagrammatically in Figure 2. As can be seen in Figure 2 in abstract program \(P'\), point \(\hat{C}\) is deemed to be after the placement of non-deterministic assignments. We claim that abstract program \(P'\) is a sound abstraction of concrete program \(P\). We make following observations regarding this abstraction mechanism.

1. Since abstract program \(P'\) is a sound abstraction of concrete program \(P\), if property holds in abstract program \(P'\), it will hold in concrete program \(P\) also.

2. If the property does not hold in abstract program \(P'\) then the counter example must follow one of the paths labeled as 2,5',6',7'.

3. Execution traces which bypass predicate computing point \(\hat{C}\) will be same in concrete as well as abstract program (shown by paths labels 1 and 2 in diagram). Consequently, if there is a counter example in abstract program \(P'\) bypassing \(\hat{C}\), like paths labeled (2) in Figure 2 then the same counter example will apply to concrete program \(P\) also.
When we have a counter example trace passing through $\tilde{C}$, then the suffix of the trace starting with last occurrence of $\tilde{C}$ will be called as violating-suffix. Obviously all the occurrences of predicate $C$, if any, on the violating-suffix will evaluate to same value. A counter example trace will have a violating-suffix if and only if the trace passes through $\tilde{C}$.

**Definition 1** (Irrelevant to Property (ITP) Predicates) Predicate $C$ is said to be ITP for the property encoded with assertion $A$ when one of the following holds:

1. If the property gets violated in abstract program $P'$, with a counter example trace having a violating-suffix $\pi$ then the property gets violated in concrete program $P$ also with a counter example trace having a violating-suffix same as $\pi$.

2. Property gets violated in abstract program $P'$ with a counter example trace having a violating-suffix with $C$ evaluating to $b$ within the violating-suffix, and predicate $C$ never evaluates to $b$ in concrete program $P$.

### 4 A sufficient criterion for ITP predicates

From the Figure 2 it is clear that we need to identify the predicates for which, the value of assert expression as well as paths labeled as (5’,6’ and 7’) are not dictated by values of variables used in predicate $C$. Let $Y$ denote set of such variables. In addition, the variables which dictate these paths and value of assert expression, should have same values at $\tilde{C}$ in abstract program $P'$ as well as in concrete program $P$.

To formalise the idea we proceed as follows. In the rest of the discussion, we assume that program $P$ is already sliced with respect to assertion $A$. Suppose we abstracted a given program $P$ to abstract program $P'$ with respect to a predicate $\tilde{C}$, as per the strategy mentioned earlier. Let the property be violated in abstract program $P'$. We want to know under what conditions we can say that it will be violated in concrete program $P$ also. We need to consider only the case where counter example found in abstract program passes through the predicate computing point $\tilde{C}$. Let such a counter
example trace $\tau'$ have program state $\sigma'$ at the last occurrence of predicate computing point $\hat{C}$ in $\tau'$. To be precise, $\sigma'$ is the program state just after the sequence of non-deterministic assignments placed at $\hat{C}$. Let $Y$ be set of variables used in predicate $C$. Let $\pi'$ be the violating-suffix of trace $\tau'$, as shown in Figure 2.

Let $X'$ be set of variables whose value in $\sigma'$ determined the value of $A$ and the violating-suffix $\pi'$, excluding the control points corresponding to predicate $C$. Intuitively, $X'$ is the set of live variables at predicate computing point $\hat{C}$, computed as follows:

1. Start at $A$ with variables used in $A$ as initial set of live variables.
2. Proceed along path $\pi'$ up to $\hat{C}$ computing live variables at different nodes as per the traditional approach.
3. Treat nodes corresponding to $C$ as identity.

Now, let us see what will it require to get a counter example in concrete program $P$ also. There are two cases to be considered.

Case 1: Violating-suffix $\pi'$ bypasses condition $C$.
Suppose we get a program state $\sigma$ in concrete program $P$ at predicate computing point $\hat{C}$ such that it has same values of variables in $X'$ as that in $\sigma'$. Now the path taken in concrete program $P$, from $\hat{C}$ starting in state $\sigma$ will be same as $\pi'$ and consequently, assertion $A$ will get violated in concrete program $P$ also.

Case 2: Violating-suffix $\pi'$ passes through condition $C$.
Let $b$ be the violating-value of predicate $C$. Suppose we get a program state $\sigma$ in concrete program $P$ at predicate computing point $\hat{C}$ such that it has same values for variables in $X'$ as that in $\sigma'$ and predicate $C$ evaluates to $b$ in state $\sigma$. It may be noted that value of predicate $C$ will not change till we do not revisit the predicate computing point $\hat{C}$. Therefore, the path followed from $\hat{C}$ starting from state $\sigma$ will be same as $\pi'$ and consequently, it will definitely make $A$ get violated in concrete program $P$.

We generalise above observations along two lines:

1. We expand $X'$ to $X$ as set of live variables along all paths from predicate computing point $\hat{C}$ to assertion point $A$ (with the restriction that $\hat{C}$ does not repeat on these paths).
2. Rather than looking for a state $\sigma$ having same $X'$ restriction as that of $\sigma'$, we consider restriction of all states at $\hat{C}$ with respect to $X$ (generalisation of $X'$) in abstract program $P'$ as well as in concrete program $P$.

To formalise, let $\Sigma'$ and $\Sigma$ be set of program states at $\hat{C}$ in abstract program $P'$ and concrete program $P$ respectively. Let $\Sigma_b \subseteq \Sigma$ be set of states in which expression of predicate $C$ evaluates to $b \in \{true, false\}$. We claim as follows:

Claim 1  If for a predicate $C$, both of the following hold then $C$ is ITP.
(S1) $|\Sigma'|_X = |\Sigma|_X$

(S2) $\forall b \in \{\text{true}, \text{false}\}. \Sigma_b \neq \emptyset \implies |\Sigma_b|_X = |\Sigma|_X$

**Proof 1** We will consider two cases:

Case 1: The counter example in $P'$ bypasses the predicate $C$.

By the precondition (S1) of the claim, $|\Sigma'|_X = |\Sigma|_X$. Since $\Sigma' \neq \emptyset$, there will be a state $\sigma \in \Sigma$ at $\hat{C}$ in concrete program $P$ such that $X$ restriction of $\sigma'$ will be same as $X$ restriction of $\sigma$. Therefore path followed from $\sigma$ in concrete program $P$ will be exactly same as violating-suffix $\pi'$ and so assertion $A$ will get violated.

Case 2: The counter example in $P'$ passes through $C$.

Let $b$ be the violating-value of predicate $C$. If $C$ always evaluates to constant $\neg b$ in concrete program $P$ then $C$ is ITP as per the definition. So we assume that $C$ evaluates to $b$ also sometimes. Therefore $\Sigma_b \neq \emptyset$. And consequently, by (S2) and (S1), $|\Sigma_b|_X = |\Sigma'|_X$ By similar argument as that in case (1), we can show that assertion $A$ will get violated in concrete program $P$ also.

## 5 A computable criterion for ITP predicates

Given an assert $A$ in a program $P$, we want to identify predicates in program $P$ which satisfy the two conditions of claim [1]. In these conditions, we talk about values of $X$ and $Y$ in all program states at predicate computing point $\hat{C}$.

First we consider the condition (S1), $|\Sigma'|_X = |\Sigma|_X$. We observe that starting from same initial state, the program state, restricted to $X$, at $\hat{C}$ at its first occurrence in a trace will be same in abstract program $P'$ and the concrete program $P$, provided $X$ and $Y$ are disjoint. Subsequent changes to program states, restricted to $X$, at $\hat{C}$ will be result of its transformation along the looping paths from this occurrence of $\hat{C}$ to its next occurrence. We observe that the only difference on paths followed from one occurrence of $\hat{C}$ to its next occurrence in $P$ and $P'$ is new abstract assignments inserted for $Y$ at $\hat{C}$ in abstract program $P'$. Let $Z$ be the set of live variables at $\hat{C}$ after traversing all paths from one occurrence of $\hat{C}$ to its next occurrence, after starting with $X$ as set of live variables at $\hat{C}$. It is easy to see that if $Z$ is disjoint from $Y$ then the transformation of $X$ from one occurrence of $\hat{C}$ to its next occurrence will be in same manner in concrete program as well as abstract program.

**Claim 2** If following two conditions are satisfied then value of $X$ at $\hat{C}$ will be same in $P$ and $P'$.

(C1) $X$ and $Y$ are disjoint.

(C2) The set of live variables on paths from $\hat{C}$ to $\hat{C}$ with respect to $\langle X, \hat{C}\rangle$ is disjoint from $Y$.

Now we consider the criterion (S2) $|\Sigma_b|_X = |\Sigma|_X$. Intuitively, this condition requires that every $X$ restricted state of $\Sigma$ is some $X$ restricted state of $\Sigma_b$ also. Since
set of states $\Sigma_b$ will be decided by values of $Y$, intuitively, this criteria can be met when $X$ and $Y$ get their values in independent manner. That is they are not related in any manner when computation proceeds from ENTRY to $\hat{C}$.

Consider programs given in Figure 3. In program (a), values of $x$ and $y$ at line 5 get related (they will be same as $z$). In program (b) values of $x$ and $y$ get related as they change together in the enclosing loop. In program (c) values of $x$ and $y$ are not related.

Intuitively, we can achieve independence of $X$ and $Y$ by ensuring following:

1. No computation impacts value of both $X$ as well as $Y$.

2. Values of $(X, \hat{C})$ and $(Y, \hat{C})$ do not change together in different iterations of any loop enclosing $\hat{C}$.

So now we will derive computable criteria which will satisfy these two requirements individually and then show that together they are sufficient to ensure (S2).

### 5.1 Non interfering computations

We observe that only the computations in value slice of $\langle V, \ell \rangle$ may affect value of $\langle V, \ell \rangle$. So obviously we need to consider the common computations belonging to value slice of $\langle X, \hat{C} \rangle$ as well as to that of $\langle Y, \hat{C} \rangle$. But when we want to see if values of $\langle X, \hat{C} \rangle$ and $\langle Y, \hat{C} \rangle$ may get affected by a common computation, we need to look at computations which provide values to predicates that only control reachability of $\hat{C}$ but otherwise are not common to value slice of $\langle X, \ell \rangle$ and $\langle Y, \hat{C} \rangle$. This is because, a controlling predicate in a way restricts the values of variables participating in the predicate, along the true and false branches. If such variables, later take part in computations which affect the value of $\langle X, \hat{C} \rangle$ as well as $\langle Y, \hat{C} \rangle$ then the computations which provide value to such predicates, should be considered as candidates affecting the value of $\langle X, \hat{C} \rangle$ as well as $\langle Y, \hat{C} \rangle$.

To illustrate, consider programs of Figure 4. In both programs (a) and (b), predicates $Q_1$ at line 1 and $Q_2$ at line 4 are controlling the reachability of predicate computing point $\hat{C}$ at line 7. We notice that, in program (a), through value of variable $z$ at $Q_1$, 

---

(a) (computation-related) 
(b) (loop-related)  
(c) (un-related)
predicate $Q_1$ will restrict the value of $x$ at $\hat{C}$. Similarly, value of $z$ at $Q_1$ will also play a role in predicate $Q_2$ restricting the value of $y$ at $\hat{C}$ (assuming value of $z$ does not get reassigned from $Q_1$ to $Q_2$). As a result, in program (a), computation of $z$ before $Q_1$ is relating value of $x$ and $y$ at $\hat{C}$ through predicates $Q_1$ and $Q_2$. However, in program (b), it is value of variable $z_1$ which plays a role in predicate $Q_1$ restricting value of $x$ at $\hat{C}$ and it is value of variable $z_2$ which plays a role in predicate $Q_2$ restricting value of $y$ at $\hat{C}$. As a result value of $x$ and $y$ at $\hat{C}$ do not get related due to $Q_1$ and $Q_2$ in program (b).

To find out whether some computation relates values of $(X, \hat{C})$ and $(Y, \hat{C})$, we will extend the concept of value impacting statement and call it extended value impacting. A predicate which controls the reachability of point of interest and uses same definition of a variable which is used by a value impacting node, is also treated as a value impacting node. For example, in program(a) of Figure 8, predicate $Q_2$ uses same value of $y$ as that used in value impacting assignment $y=y+1$ for $(y, \hat{C})$. Therefore, predicate $Q_2$ is also considered as value impacting for $(y, \hat{C})$. Similarly predicate $Q_1$ will be considered as value impacting for $(x, \hat{C})$.

**Definition 2** (Extended Value-impacting node) A node $s$ extended-value-impacts $\Upsilon = (l, V)$, if any of the following conditions hold:

1. $s$ is an assignment in $DU(\Upsilon)$. 
2. $s$ is an assignment, and there exists a node $t$ such that $t$ extended-value-impacts $\Upsilon$ and $s$ is in $DU(LV(t))$. 
3. $s$ is a predicate $c$ from which there exist paths $\pi_1$ and $\pi_2$ starting with the out-edges of $c$ and ending at the first occurrence of $l$. Further, there exists a node $t \neq c$ such that $t$ extended-value-impacts $\Upsilon$, and (a) $t$ is the first value-impacting node along $\pi_1$ (b) $t$ is not the first value-impacting node along $\pi_2$. 
4. $s$ is a predicate $c$ which transitively controls $l$. Further, there exists an extended-value-impacting node $t \neq c$ and an assignment $d$ such that $d$ is in $DU(LV(t))$ and $d$ is in $DU(LV(c))$. 

A slice made up from extended value impacting statements, will be called extended value slice. We know that in a program which is already (backward) sliced with respect to $(V, \ell)$, only variables which are live at ENTRY point dictate the value of $(V, \ell)$ and reachability of $\ell$. Similarly, in an extended value slice with respect to $(V, \ell)$, variables live at ENTRY point will dictate value of $(V, \ell)$, whenever $\ell$ is reached. For extended value slice we will call these variable as value base of $(V, \ell)$ and denote them as $VB(V, \ell)$. We will denote the set of extended value impacting nodes for $(V, \ell)$ by $EVI(V, \ell)$.

We claim that if set of value base variables for $(X, \hat{C})$ and $(Y, \hat{C})$ are disjoint then no computation affects value of both when $\hat{C}$ is not enclosed in a loop.

(C3) $VB(X, \hat{C}) \cap VB(Y, \hat{C}) = \emptyset$

**5.2 Avoiding relation through loops**

To find a criterion which ensures that value of $(X, \hat{C})$ and $(Y, \hat{C})$ do not become related due to they changing together in a loop $L$, we need to see how to identify if some
variable is changing in a loop.

Suppose we observe value of a set of variables $Z$ at a point $\ell$ which is inside a loop $L$. Let $[L]$ represent the body of loop $L$ and $VB(L)$ denote the value base variables of loop controlling condition of loop $L$. Let $EVI(Z, \ell)$ be set of extended value impacting statements for $\langle Z, \ell \rangle$. We claim that value of $Z$ at $\ell$ may change with different iterations of loop $L$ only if at least one extended value impacting statement from $EVI(Z, \ell)$ is outside the body of loop $L$ and at least one such statement is inside the loop body. Let us call such loops as value changing loops for $\langle Z, \ell \rangle$. That is if $EVI(Z, \ell) \cap [L] \neq \emptyset$ and $EVI(Z, \ell) \subseteq [L]$ then only $\langle Z, \ell \rangle$ can change its value in different iterations of loop $L$.

So to ensure that value of $\langle X, \hat{C} \rangle$ and $\langle Y, \hat{C} \rangle$ do not change together in any enclosing loop of $\hat{C}$, following criterion (C4) should be satisfied.

(C4) For all loop $L$ enclosing $\hat{C}$, one of the following should hold.

(a) $(EVI(X, \hat{C}) \cap [L]) = \emptyset$ or $(EVI(X, \hat{C}) \subseteq [L])$

(b) $(EVI(Y, \hat{C}) \cap [L]) = \emptyset$ or $(EVI(Y, \hat{C}) \subseteq [L])$

We observe that, if value of $\langle V, \ell \rangle$ changes in different iterations of a loop $L$ then loop controlling conditions of all outer loops (enclosing loop $L$) will be value impacting node of $\langle V, \ell \rangle$. Based on this observation and criterion (C4) about value changing loops, we have following properties for value changing loops of $\langle X, \hat{C} \rangle$ and $\langle Y, \hat{C} \rangle$.

(P1) if $\langle X, \hat{C} \rangle$ changes in a loop $L$ then for all outer loops $L'$ of $L$, $VB(L') \cap VB(X, \hat{C}) = \emptyset$ or $VB(L') \subseteq VB(X, \hat{C})$.

(P2) if $\langle Y, \hat{C} \rangle$ changes in a loop $L$ then for all outer loops $L'$ of $L$, $VB(L') \cap VB(Y, \hat{C}) = \emptyset$ or $VB(L') \subseteq VB(Y, \hat{C})$.

(P3) if $\langle X, \hat{C} \rangle$ changes in a loop $L_1$ and $\langle Y, \hat{C} \rangle$ changes in a loop $L_2$ then $VB(L_1)$ and $VB(L_2)$ are disjoint.
Claim 3 If criteria (C3) and (C4) hold for a predicate \( C \), then following will hold:
\( \forall b \in \{ \text{true}, \text{false} \}. \Sigma_b \neq \emptyset \implies [\Sigma_b]_X = [\Sigma]_X \)

Proof 2 Let \( \bar{X} \) and \( \bar{Y} \) be the set of value base variables for \( \langle X, \hat{C} \rangle \) and \( \langle Y, \hat{C} \rangle \) respectively. In addition let \( V \) be the set of all variables. Let \( \bar{Z} = V - (\bar{X} \cup \bar{Y}) \). Obviously, \( \bar{X}, \bar{Y} \) and \( \bar{Z} \) are disjoint.

Assume that \( b \) is \( \text{true} \) and \( \Sigma_{\text{true}} \neq \emptyset \). Suppose \( \sigma \in \Sigma \) and \( \sigma_t \in \Sigma_{\text{true}} \). Let input state \( I_1 \) produce \( \sigma \) and \( I_2 \) produce \( \sigma_t \). We partition \( I_1 \) into \( [I_1]_{\bar{X}}, [I_1]_{\bar{Y}} \) and \( [I_1]_{\bar{Z}} \).

Similarly, we partition \( I_2 \) into \( [I_2]_{\bar{X}}, [I_2]_{\bar{Y}} \) and \( [I_2]_{\bar{Z}} \).

Let \( VB(L) \) denote the base variables for loop controlling conditions of loop \( L \). We construct an input \( I_3 \) such that values of \( \bar{X} \) come from \( I_1 \) and those for \( \bar{Y} \) come from \( I_2 \).

For the remaining values we proceed as follows:

1. If \( \langle X, \hat{C} \rangle \) changes in \( L \) then use values of \( VB(L) \) from \( I_1 \).
2. If \( \langle Y, \hat{C} \rangle \) changes in \( L \) then use values of \( VB(L) \) from \( I_2 \).
3. If none of \( \langle X, \hat{C} \rangle \) and \( \langle Y, \hat{C} \rangle \) changes in \( L \) then use values of \( VB(L) \) from \( I_3 \).

By the properties (P1) to (P5), such assignments will be possible without any conflict. That is no variable will be required to have it value from \( I_1 \) as well as from \( I_2 \). By the above step, some of variables from \( \bar{Z} \) would have got their input values. For the remaining variables of \( \bar{Z} \), if any, take their values from \( I_3 \). Obviously starting with input \( I_3 \), \( C \) will be reachable, with value of \( X \) at \( \hat{C} \) same as that with \( I_3 \) and value of \( Y \) at \( \hat{C} \) same as that with \( I_2 \). Therefore, one of the states produced with input \( I_3 \) at \( \hat{C} \) will have same values of \( X \) as that in \( \sigma \) and same values of \( Y \) as that in \( \sigma_t \). Let \( \sigma_3 \) be such a state. Obviously \( C \) will evaluate to \( \text{true} \) in state \( \sigma_3 \) and therefore \( \sigma_3 \in \Sigma_{\text{true}} \).

Moreover \( [\sigma]_X = [\sigma_3]_X \). Therefore \( [\Sigma]_X \subseteq [\Sigma_{\text{true}}]_X \). similarly we can show that \( [\Sigma]_X \subseteq [\Sigma_{\text{false}}]_X \).

It is obvious that checking the criteria (C1), (C2), (C3) and (C4) is computable for a given predicate \( C \).

6 Property checking with ITP predicates

Assume that we identified an ITP predicate \( C \) in program \( P \) as per the definition [1]. We abstract \( P \) to \( P' \) using the abstraction strategy mentioned earlier. Now we run a property checker on program \( P' \). We consider following cases:

(i) Property holds in \( P' \).

(ii) Property gets violated with a counter example bypassing \( \hat{C} \).
(iii) Property is violated in $P'$ with counter example passing through $\hat{C}$.

For case (i), we are done as the property will hold in program $P$ also. For case (ii), the property will get violated in $P$ also with same counter example as that of $P'$, as per the property of the abstraction mechanism. The case (iii) needs to be analysed further and we proceed as follows:

Let input $I'$ be the counter example, producing trace $\tau'$, for abstract program $P'$.

Since the counter example trace passes through $\hat{C}$, it will have a violating-suffix, say $\pi$. We execute program $P$ with same input $I'$ to get trace $\tau$ and consider following possible outcomes for trace $\tau$.

1. Assertion is violated.
2. Assertion is not violated and trace $\tau'$ bypasses the condition $C$.
3. Assertion is not violated in trace $\tau$ and trace $\tau'$ passes through $C$ with violating-value of $C$ as $b \in \{true, false\}$.

If it is case (1) then we are done as we found a counter example in concrete program $P$ also. For case (3) we consider following sub cases:

3(a) Trace $\tau$ never passed through $C$.
3(b) Trace $\tau$, passed through $C$ and $C$ evaluated to $b$ at least once.
3(c) Trace $\tau$ passed through $C$ but $C$ always evaluated to $\neg b$.

If it is case 2, 3(a) or 3(c), we want to check the possibility of $C$ always evaluating to $\neg b$ in program $P$. For this, we create a program $\hat{P}$ from $P$ by placing a new assert expression $(C == \neg b)$ at $\hat{C}$ and removing the old assert. We also replace predicate $C$ with $\neg b$. We claim that assertion in $\hat{P}$ will hold if and only if $C$ always evaluated to $\neg b$ in program $P$. We solve the new property checking problem $\hat{P}$ and consider following possible outcomes.

(A) New property does not hold in $\hat{P}$, implying that the predicate $C$ evaluates to $b$ also sometimes.

(B) New property in $\hat{P}$ holds implying $C$ always evaluates to $\neg b$.

In case (B), we create a new property checking problem $\tilde{P}$ from $P$ by replacing $C$ with $\neg b$. Problem $\tilde{P}$ will be property equivalent to $P$. Solving $\tilde{P}$ will give solution to $P$.

For the cases (A) and 3(b), by definition [1] of ITP predicates, we know that, for program $P$, there exists a counter example trace which has a violating-suffix matching with violating-suffix $\pi$ of counter example trace of $P'$. We compute a weakest-precondition $\psi$ of $\neg A$ for the path $\pi$. Obviously, $\psi$ must be satisfiable at $\hat{C}$ in $P$. So there must exist an input for program $P$ which satisfies $\psi$ at $\hat{C}$ and the same will indeed be a counter example in $P$ for assertion $A$.

So we just need to find a counter example which violates the assertion $\neg \psi$ at $\hat{C}$. So, we create an input generation problem as property checking for the assertion $\neg \psi$ in program $P$ at $\hat{C}$. Obviously, this assertion must get violated and if verifier finds an input for the same, it will be counter example for the original property checking problem.