A Conditioned Program Slicer

Chris Fox, University of Essex

21st February 2005
1 Background
   - Slicing
   - Conditioning
   - Slicing and Conditioning
   - Constraining the Context

2 Program Conditioning
   - Symbolic Execution and Theorem Proving
   - Combining Symbolic States
   - Uninterpreted Constant Values

3 Examples of Conditioning
   - Conditional Statements
   - Conditioned “if”
   - While Loops
   - Conditioned “while”
Philosophy/Motivation

- Put a programmer-friendly face on software analysis
Philosophy/Motivation

• Put a programmer-friendly face on software analysis
• Use formal analysis to check the correctness of the analysis tools and their transformations,
Philosophy/Motivation

- Put a programmer-friendly face on software analysis
- Use formal analysis to check the correctness of the analysis tools and their transformations, not the programs under examination.
Philosophy/Motivation

- Put a programmer-friendly face on software analysis
- Use formal analysis to check the correctness of the analysis tools and their transformations, not the programs under examination.
- A pragmatic approach to formal methods.
Philosophy/Motivation

- Put a programmer-friendly face on software analysis
- Use formal analysis to check the correctness of the analysis tools and their transformations, not the programs under examination.
- A pragmatic approach to formal methods?
Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.
Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.

Usually specified in terms of a set of variables and program point of interest, with the variables constituting the “output” of the subcomputation (backward slicing), or its “input” (forward slicing).
Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.

Usually specified in terms of a set of variables and program point of interest, with the variables constituting the “output” of the subcomputation (backward slicing), or its “input” (forward slicing).

Usually using one transformation:
Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.

Usually specified in terms of a set of variables and program point of interest, with the variables constituting the “output” of the subcomputation (backward slicing), or its “input” (forward slicing).

Usually using one transformation: *statement deletion*. 
Slicing

- Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.
- Usually specified in terms of a set of variables and program point of interest, with the variables constituting the “output” of the subcomputation (backward slicing), or its “input” (forward slicing).
- Usually using one transformation: *statement deletion*.
- Need to keep statements that could change (or are changed by) the values of interest.
Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.

Usually specified in terms of a set of variables and program point of interest, with the variables constituting the “output” of the subcomputation (backward slicing), or its “input” (forward slicing).

Usually using one transformation: *statement deletion*.

Need to keep statements that could change (or are changed by) the values of interest.

1. Data dependence
2. Control dependence
Slicing can be thought of as projecting out part of a program involved in a subcomputation of interest.

Usually specified in terms of a set of variables and program point of interest, with the variables constituting the “output” of the subcomputation (backward slicing), or its “input” (forward slicing).

Usually using one transformation: *statement deletion*.

Need to keep statements that could change (or are changed by) the values of interest.

1. Data dependence
2. Control dependence

Computing either of these precisely is problematic, so we are obliged to accept conservative approximations.
Example
Example

Consider:

```plaintext
x := y;
w := x;
while (x > z) {
    w := w + 1;
x := x - 1;
}
```
Example

Consider:

\[
x := y; \\
w := x; \\
while (x > z) \{ \\
    w := w + 1; \\
    x := x - 1; \\
\}
\]

Slicing (backward) with respect to the value of \(x\) at the end of the program will give the code in red (the statements in gray can be “sliced away”.)
Consider:

\[
x := y;  \\
w := x;  \\
\text{while } (x > z) \{  \\
\text{ } w := w + 1;  \\
\text{ } x := x - 1;  \\
\text{ } \}
\]

Slicing (backward) with respect to the value of \( x \) at the end of the program will give the code in red (the statements in gray can be “sliced away”.)
Another Example
Another Example

Consider:

```plaintext
x := y;
w := x;
while (true) {
    w := 1;
x := x + 1;
}
```
Another Example

- Consider:

```plaintext
x := y;
w := x;
while (true) {
    w := 1;
x := x + 1;
}
```

- Slicing (backward) with respect to the value of \( w \) at the end of the program will give the code in red (the statements in gray can be “sliced away”.)
Another Example

Consider:

```plaintext
x := y;
w := x;
while (true) {
    w := 1;
x := x + 1;
}
```

Slicing (backward) with respect to the value of \( w \) at the end of the program will give the code in red (the statements in gray can be “sliced away”.)
Another Example

- Consider:

```plaintext
x := y;
w := x;
while (true) {
    w := 1;
x := x + 1;
}
```

- Slicing (backward) with respect to the value of `w` at the end of the program will give the code in red (the statements in gray can be “sliced away”.)

- This illustrates the impact that slicing can have on termination behaviour: it cannot be analysed as giving a simple projection of the (usual) semantics of the program.
Problematic Example

There is a statement/line in the following program that is not involved in determining the final value of $x$ in the following program.
There is a statement/line in the following program that is not involved in determining the final value of x in the following program.

```c
while (p(i)) {
    if (q(c)) {
        x := f();
        c := g();
    };
    i := h(i)
}
```
Problematic Example

- There is a statement/line in the following program that is not involved in determining the final value of $x$ in the following program.

```c
while (p(i)) {
    if (q(c)) {
        x := f();
        c := g();
    }
    i := h(i)
}
```

- No conventional slicing algorithm can find it.
Slicing is Not Trivial
Slicing is Not Trivial

- Even just using statement deletion, slicing real programs is non-trivial:
Slicing is Not Trivial

Even just using statement deletion, slicing real programs is non-trivial:

1. unstructured code (breaks and returns);
Slicing is Not Trivial

- Even just using statement deletion, slicing real programs is non-trivial:
  1. unstructured code (`breaks and returns`);
  2. arrays, union types and pointers (`& functions in C`);
Slicing is Not Trivial

Even just using statement deletion, slicing real programs is non-trivial:

1. unstructured code (breaks and returns);
2. arrays, union types and pointers (& functions in C);
3. difficult to analyse with standard state-based semantics.
Slicing is Not Trivial

Even just using statement deletion, slicing real programs is non-trivial:

1. unstructured code (breaks and returns);
2. arrays, union types and pointers (& functions in C);
3. difficult to analyse with standard state-based semantics (non-termination behaviour changes);
4. minimal slices are not computable.
Slicing is Not Trivial

Even just using statement deletion, slicing real programs is non-trivial:

1. unstructured code (breaks and returns);
2. arrays, union types and pointers (& functions in C);
3. difficult to analyse with standard state-based semantics (non-termination behaviour changes);
4. minimal slices are not computable.

How should we slice OO languages?
Varieties of Slicing

**Static Slicing** compute the slice for all possible input values.
Varieties of Slicing

**Static Slicing** compute the slice for all possible input values. Can be computed using a Program Dependence Graph, or by compositional analysis.
Varieties of Slicing

**Static Slicing** compute the slice for all possible input values. Can be computed using a Program Dependence Graph, or by compositional analysis.

**Dynamic Slicing** compute the slice for a completely specified input.
Varieties of Slicing

**Static Slicing** compute the slice for all possible input values. Can be computed using a Program Dependence Graph, or by compositional analysis.

**Dynamic Slicing** compute the slice for a completely specified input.

**Amorphous Slicing** Allow more generic transformations.
Varieties of Slicing

**Static Slicing** compute the slice for all possible input values. Can be computed using a Program Dependence Graph, or by compositional analysis.

**Dynamic Slicing** compute the slice for a completely specified input.

**Amorphous Slicing** Allow more generic transformations.

**Variable Dependence** Extract relationships between input and output variables using a slicing algorithm.
Varieties of Slicing

**Static Slicing** compute the slice for all possible input values. Can be computed using a Program Dependence Graph, or by compositional analysis.

**Dynamic Slicing** compute the slice for a completely specified input.

**Amorphous Slicing** Allow more generic transformations.

**Variable Dependence** Extract relationships between input and output variables using a slicing algorithm.

**Conditioned Program Slicing** (impose *conditions* on input variables, or program points, and use that information to decrease the size of a subsequent backward slice.)
Conditioning

\[ \text{Conditioning} = \]
Conditioning = Symbolic Execution
Conditioning = Symbolic Execution + Theorem Proving
Conditioning

\[\text{Conditioning} = \text{Symbolic Execution} + \text{Theorem Proving}\]

Symbolic Execution
Conditioning = Symbolic Execution + Theorem Proving

Symbolic Execution Ideally finds all the possible paths through a program, and the corresponding symbolic states
Conditioning

Conditioning = Symbolic Execution + Theorem Proving

Symbolic Execution  Ideally finds all the possible paths through a program, and the corresponding symbolic states

Theorem Proving
Conditioning = Symbolic Execution + Theorem Proving

Symbolic Execution  Ideally finds all the possible paths through a program, and the corresponding symbolic states

Theorem Proving  Determines which of these paths are infeasible
Conditioning = Symbolic Execution + Theorem Proving

**Symbolic Execution** Ideally finds all the possible paths through a program, and the corresponding symbolic states.

**Theorem Proving** Determines which of these paths are infeasible, and hence which statements can be eliminated.
Conditioning = Symbolic Execution + Theorem Proving

Symbolic Execution  Ideally finds all the possible paths through a program, and the corresponding symbolic states.

Theorem Proving  Determines which of these paths are infeasible, and hence which statements can be eliminated. Other kinds of reasoning and simplification are also possible (such as the simplification of expressions).
Conditioning

\[ \text{Conditioning} = \text{Symbolic Execution} + \text{Theorem Proving} \]

**Symbolic Execution**  Ideally finds all the possible paths through a program, and the corresponding symbolic states.

**Theorem Proving**  Determines which of these paths are infeasible, and hence which statements can be eliminated. Other kinds of reasoning and simplification are also possible (such as the simplification of expressions).

**Conditioned Slicing**
Conditioning = Symbolic Execution + Theorem Proving

Symbolic Execution  Ideally finds all the possible paths through a program, and the corresponding symbolic states

Theorem Proving  Determines which of these paths are infeasible, and hence which statements can be eliminated. Other kinds of reasoning and simplification are also possible (such as the simplification of expressions)

Conditioned Slicing  program conditioning is combined with conventional backward slicing to give a conditioned-program slicer.
Slicing and Conditioning

Slicing
**Slicing** We are only interested in the values of a subset of the program variables
Slicing  We are only interested in the values of a subset of the program variables; which parts of the program can safely be removed?
Slicing and Conditioning

**Slicing**  We are only interested in the values of a subset of the program variables; which parts of the program can safely be removed?

**Conditioning**
Slicing and Conditioning

**Slicing**  We are only interested in the values of a subset of the program variables; which parts of the program can safely be removed?

**Conditioning**  We are only interested in a subset of the possible input values
Slicing and Conditioning

**Slicing**  We are only interested in the values of a subset of the program variables; which parts of the program can safely be removed?

**Conditioning**  We are only interested in a subset of the possible input values; which parts of the program can safely be removed?
Slicing and Conditioning

**Slicing**  We are only interested in the values of a subset of the program variables; which parts of the program can safely be removed?

**Conditioning**  We are only interested in a subset of the possible input values; which parts of the program can safely be removed?

**Conditioned Slicing**
Slicing and Conditioning

**Slicing**  We are only interested in the values of a subset of the program variables; which parts of the program can safely be removed?

**Conditioning**  We are only interested in a subset of the possible input values; which parts of the program can safely be removed?

**Conditioned Slicing**  subsumes static and dynamic slicing.
Constraining the Context

- With conditioned slicing, we are interested in putting restrictions on possible input values.
With conditioned slicing, we are interested in putting restrictions on possible input values.

We could add conditions that quantify over the unique symbolic input values:
With conditioned slicing, we are interested in putting restrictions on possible input values.

We could add conditions that quantify over the unique symbolic input values:

```c
while (p) {
  ...scanf("%d", &a); ...
};
```
Constraining the Context

- With conditioned slicing, we are interested in putting restrictions on possible input values.
- We could add conditions that quantify over the unique symbolic input values:

\[ \forall n. (a_n > 0) \]

\[
\text{while } (p) \{ \\
\text{...scanf("%d", &a); ...} \\
\}
\]
With conditioned slicing, we are interested in putting restrictions on possible input values.

We could add conditions that quantify over the unique symbolic input values:

$$\forall n. (a_n > 0)$$

```c
while (p) {
    ... scanf("%d", &a); ...
};
```

In our implementation, we adopt the simpler approach of using statements of the form `assert(condition):`
Constraining the Context

- With conditioned slicing, we are interested in putting restrictions on possible input values.
- We could add conditions that quantify over the unique symbolic input values:
  \[
  \forall n. (a_n > 0)
  \]
  
  ```
  while (p) {
    ...scanf("%d", &a); ...
  }
  ```

- In our implementation, we adopt the simpler approach of using statements of the form `assert(condition)`:  
  ```
  while (p) {
    ...scanf("%d", &a);
    ...
  }
  ```
With conditioned slicing, we are interested in putting restrictions on possible input values.

We could add conditions that quantify over the unique symbolic input values:

$$\forall n.(a_n > 0)$$

```c
while (p) {
    ...scanf("%d", &a); ...
}
```

In our implementation, we adopt the simpler approach of using statements of the form `assert(condition):`

```c
while (p) {
    ...scanf("%d", &a); assert(a>0);
    ...
}
```
“Execution” of the program, but where all unknown and input values are represented by symbolic values.
“Execution” of the program, but where all unknown and input values are represented by symbolic values.
Symbolic Execution and Theorem Proving

“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
</table>
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$z_0$</td>
</tr>
</tbody>
</table>
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$z_0$</td>
</tr>
</tbody>
</table>
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := z;$</td>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$z_0$</td>
</tr>
</tbody>
</table>
Symbolic Execution and Theorem Proving

“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \leftarrow z;)</td>
<td>(x_0)</td>
<td>(y_0)</td>
<td>(z_0)</td>
</tr>
<tr>
<td>(y \leftarrow x + z;)</td>
<td>(z_0)</td>
<td>(y_0)</td>
<td>(z_0)</td>
</tr>
</tbody>
</table>

Chris Fox, University of Essex

A Conditioned Program Slicer
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := z;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>y := x + z;</td>
<td>z₀</td>
<td>z₀ + z₀</td>
<td>z₀</td>
</tr>
</tbody>
</table>
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := z;</td>
<td>x₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>y := x + z;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀ + z₀</td>
</tr>
</tbody>
</table>

**Theorem Proving**
Symbolic Execution and Theorem Proving

“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := z;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>y := x + z;</td>
<td>z₀</td>
<td>z₀ + z₀</td>
<td>z₀</td>
</tr>
</tbody>
</table>

Theorem Proving

Does $y = 2 \times z$?
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;x := z;&quot;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>&quot;y := x + z;&quot;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀ + z₀</td>
</tr>
</tbody>
</table>

Theorem Proving

Does \( y = 2z \)? From the symbolic state, this is true if:
\[ z₀ + z₀ = 2z₀ \]
“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$z_0$</td>
</tr>
<tr>
<td>$x := z;$</td>
<td>$z_0$</td>
<td>$y_0$</td>
<td>$z_0$</td>
</tr>
<tr>
<td>$y := x + z;$</td>
<td>$z_0$</td>
<td>$z_0 + z_0$</td>
<td>$z_0$</td>
</tr>
</tbody>
</table>

**Theorem Proving**

- Does $y = 2z$? From the symbolic state, this is true if: $z_0 + z_0 = 2z_0$
- Is $x < y$?
Symbolic Execution and Theorem Proving

“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>x := z;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>y := x + z;</td>
<td>z₀</td>
<td>z₀ + z₀</td>
<td>z₀</td>
</tr>
</tbody>
</table>

Theorem Proving

- Does \(y = 2z\)? From the symbolic state, this is true if: \(z₀ + z₀ = 2z₀\)

- Is \(x < y\)? True if \(z₀ < 2z₀\).
Symbolic Execution and Theorem Proving

“Execution” of the program, but where all unknown and input values are represented by symbolic values.

<table>
<thead>
<tr>
<th>Program Source</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := z;</td>
<td>x₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
<tr>
<td>y := x + z;</td>
<td>z₀</td>
<td>y₀</td>
<td>z₀</td>
</tr>
</tbody>
</table>

Theorem Proving

- Does $y = 2z$? From the symbolic state, this is true if: $z₀ + z₀ = 2z₀$

- Is $x < y$? True if $z₀ < 2z₀$. What if $z$ is negative?
Combining Symbolic States

- The symbolic executor finds a set of pairs of *path conditions* and *symbolic states*. 
Combining Symbolic States

- The symbolic executor finds a set of pairs of *path conditions* and *symbolic states*.
- When encountering a condition `if (p) s else t`, each `(path ➝ state)` pair is replaced by the results of:
The symbolic executor finds a set of pairs of \textit{path conditions} and \textit{symbolic states}.

When encountering a condition \texttt{if (p) s else t}, each \texttt{(path \rightarrow state)} pair is replaced by the results of:
\begin{itemize}
  \item symbolically execution \texttt{s} in the context of \texttt{path \cup \{p\}};
\end{itemize}
The symbolic executor finds a set of pairs of path conditions and symbolic states.

When encountering a condition if \((p) s\) else \(t\), each \((path \rightarrow state)\) pair is replaced by the results of:

1. symbolically execution \(s\) in the context of \(path \cup \{p\}\);
2. symbolically execution \(t\) in the context of \(path \cup \{\neg p\}\);
The symbolic executor finds a set of pairs of path conditions and symbolic states.

When encountering a condition if (p) s else t, each (path $\rightarrow$ state) pair is replaced by the results of:

1. symbolically execution s in the context of path $\cup \{p\}$;
2. symbolically execution t in the context of path $\cup \{\neg p\}$;

With while loops present additional complexities.
The symbolic executor finds a set of pairs of *path conditions* and *symbolic states*.

When encountering a condition `if (p) s else t`, each `(path → state)` pair is replaced by the results of:

1. symbolically execution `s` in the context of `path ∪ {p}`;
2. symbolically execution `t` in the context of `path ∪ {¬p}`;

With *while* loops present additional complexities. We have chosen to implement a conservative approximation.
A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:
A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

- **Initial value**
Uninterpreted Constant Values

A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

- **Initial value** When $v$ is referenced prior to being assigned a value
Uninterpreted Constant Values

A variable $\nu$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value** When $\nu$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $\nu_0$. 
A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value** When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value**
A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value** When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value** When $v$ receives a value in an input (scanf) statement
Uninterpreted Constant Values

A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value** When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value** When $v$ receives a value in an input (`scanf`) statement, it is given a unique, uninterpreted constant value, $v_n$. 
Uninterpreted Constant Values

A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value**  When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value**  When $v$ receives a value in an input (scanf) statement, it is given a unique, uninterpreted constant value, $v_n$.

**Within loops**
Uninterpreted Constant Values

A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

- **Initial value** When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

- **Input value** When $v$ receives a value in an input (scanf) statement, it is given a unique, uninterpreted constant value, $v_n$.

- **Within loops** When $v$ is assigned a value within a loop body.
Uninterpreted Constant Values

A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value**  When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value**  When $v$ receives a value in an input (scanf) statement, it is given a unique, uninterpreted constant value, $v_n$.

**Within loops**  When $v$ is assigned a value within a loop body, we associate the variable with an uninterpreted value $v_p$. 

Chris Fox, University of Essex  A Conditioned Program Slicer
Uninterpreted Constant Values

A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value** When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value** When $v$ receives a value in an input ($\text{scanf}$) statement, it is given a unique, uninterpreted constant value, $v_n$.

**Within loops** When $v$ is assigned a value within a loop body, we associate the variable with an uninterpreted value $v_p$, conceptually on the penultimate execution of the loop.
A variable $v$ is assigned a unique, uninterpreted constant value in the following circumstances:

**Initial value** When $v$ is referenced prior to being assigned a value, it is given a unique, uninterpreted constant value, $v_0$.

**Input value** When $v$ receives a value in an input `(scanf)` statement, it is given a unique, uninterpreted constant value, $v_n$.

**Within loops** When $v$ is assigned a value within a loop body, we associate the variable with an uninterpreted value $v_p$, conceptually on the penultimate execution of the loop, then we symbolically execute the loop body once to “approximate” the final symbolic values.
Symbolic States and Path Conditions

\[ x = y + 1; \]
Symbolic States and Path Conditions

\[
x = y + 1;
\]

(path condition $\implies$ symbolic state)

\[
\{ \top \implies (x = y_0 + 1) \}\]
Symbolic States and Path Conditions

\[ x = y + 1; \]

(path condition \( \models \) symbolic state)

\[ \{ \top \models (x = y_0 + 1) \}\]

\[
\begin{array}{l}
\text{if} \ (x < y) \\
\quad x = 5 \\
\text{else} \\
\quad x = 10;
\end{array}
\]
Conditional Statements

Symbolic States and Path Conditions

\[ x = y + 1; \]  
(path condition $\implies$ symbolic state) 
\( \{ \top \implies (x = y_0 + 1) \} \)

Condition True

\[
\begin{align*}
\text{if} & \ (x < y) \\
& x = 5 \\
\text{else} & \\
& x = 10;
\end{align*}
\]
Symbolic States and Path Conditions

\[ x = y + 1; \]

(path condition \( \Rightarrow \) symbolic state)

\[ \{ \top \Rightarrow (x = y_0 + 1) \} \]

Condition True

\[ y_0 + 1 < y_0 \]

\[
\begin{align*}
\text{if} (x < y) \\
& \quad x = 5 \\
\text{else} \\
& \quad x = 10;
\end{align*}
\]
Symbolic States and Path Conditions

\[
x = y + 1;
\]

(path condition $\rightarrow$ symbolic state)

\[
\{ \top \rightarrow (x = y_0 + 1) \}
\]

Condition True
\[
y_0 + 1 < y_0
\]
\[
x = 5
\]

\[
\text{if (x < y)}
\]
\[
\quad x = 5
\]
\[
\text{else}
\]
\[
\quad x = 10;
\]
Symbolic States and Path Conditions

\[ x = y + 1; \]

\[
\{ \top \implies (x = y_0 + 1) \}
\]

Condition True
\[
y_0 + 1 < y_0 \]
\[
x = 5
\]

Condition False
\[
\text{if } (x < y) \]
\[
x = 5
\]
\[
\text{else}
\]
\[
x = 10;
\]
Symbolic States and Path Conditions

\[ x = y + 1; \]

(path condition \(\Rightarrow\) symbolic state)

\[ \{ \top \Rightarrow (x = y_0 + 1) \} \]

Condition True
\[
y_0 + 1 < y_0
\]
\[
x = 5
\]

Condition False
\[
y_0 + 1 \not< y_0
\]

if (\(x < y\))
\[
x = 5
\]
else
\[
x = 10;
\]
Symbolic States and Path Conditions

\[
x = y + 1; \\
(\text{path condition} \implies \text{symbolic state}) \\
\{\top \implies (x = y_0 + 1)\}
\]

Condition True
\[
y_0 + 1 < y_0 \\
x = 5
\]

Condition False
\[
y_0 + 1 \not< y_0 \\
x = 10
\]

Chris Fox, University of Essex
A Conditioned Program Slicer
Symbolic States and Path Conditions

\[ x = y + 1; \]
\[(\text{path condition } \implies \text{symbolic state})\]
\[ \{\top \implies (x = y_0 + 1)\}\]

Condition True
\[ y_0 + 1 < y_0 \]
\[ x = 5 \]

Condition False
\[ y_0 + 1 \not< y_0 \]
\[ x = 10; \]

Final Symbolic States:
\[ \{(y_0 + 1 < y_0) \implies x = 5, \ (y_0 + 1 \not< y_0) \implies x = 10\} \]
Conditioned “if”

\[ x = y + 1; \]

(path condition \( \implies \) symbolic state)
\[ \{ \top \implies (x = y_0 + 1) \} \]

Condition True
\[ y_0 + 1 < y_0 \]
\[ x = 5 \]

Condition False
\[ y_0 + 1 \nless y_0 \]
\[ x = 10 \]

Final Symbolic States:
\[ \{ (y_0 + 1 < y_0) \implies x = 5, \quad (y_0 + 1 \nless y_0) \implies x = 10 \} \]
Conditioned "if"

\[
x = y + 1;
\]

(path condition \(\Rightarrow\) symbolic state)

\[
\{ \top \Rightarrow (x = y_0 + 1) \}
\]

Condition True

\[
y_0 + 1 < y_0
\]

\[
x = 5
\]

Condition False

\[
y_0 + 1 \not\approx y_0
\]

\[
x = 10
\]

Final Symbolic States:

\[
\{(y_0 + 1 < y_0) \Rightarrow x = 5, \ (y_0 + 1 \not\approx y_0) \Rightarrow x = 10\}
\]
Conditioned “if”

```
x = y + 1;
```

(path condition $\implies$ symbolic state)

Condition True

\[
y_0 + 1 < y_0
\]

\[
x = 5
\]

Condition False

\[
y_0 + 1 \not< y_0
\]

\[
x = 10
\]

Final Symbolic States:

\[
\{(y_0 + 1 \not< y_0) \implies x = 10\}
\]
While Loops

\[ x = y + 1; \]

\[ \{ \top \implies (x = y_0 + 1) \} \]
While Loops

\[
x = y + 1;
\]

\[\{ \top \Rightarrow (x = y_0 + 1) \}\]

\[
\text{while } (x > y) \\
x = x - 1;
\]

Final States:
While Loops

Initially False:

```
x = y + 1;
{\top \implies (x = y_0 + 1)}

while (x > y)
    x = x - 1;
```

Final States:

1. condition initially false $\implies$ state unchanged
While Loops

\[
x = y + 1;
\]
\[\{ \top \implies (x = y_0 + 1) \}\]

\[
\text{while } (x > y) \\
\quad x = x - 1;
\]

Initially False:
\[y_0 + 1 \not\approx y_0\]

Final States:

1. \[(y_0 + 1 \not\approx y_0) \implies \text{state unchanged}\]
While Loops

\[
x = y + 1;
\]
\[
\{ T \implies (x = y_0 + 1) \}
\]

Initial Condition:
\[
y_0 + 1 \not\geq y_0
\]
State:
\[
x = y_0 + 1
\]

Final States:
1. \((y_0 + 1 \not\geq y_0) \implies (x = y_0 + 1)\)
While Loops

Initially True:

\[ y_0 + 1 > y_0 \]

\[
\begin{align*}
x &= y + 1; \\
\{ \top \} &\quad \Rightarrow (x = y_0 + 1)
\end{align*}
\]

Initially False:

\[ y_0 + 1 \not> y_0 \]

\[
\begin{align*}
\text{while } (x > y) \\
\quad x &= x - 1;
\end{align*}
\]

State: \[ x = y_0 + 1 \]

Final States:

1. \((y_0 + 1 \not> y_0) \quad \Rightarrow (x = y_0 + 1)\)
2. initially true, penultimately true, finally false \( \Rightarrow \) new state
While Loops

\[
x = y + 1; \\
\{ \top \implies (x = y_0 + 1) \}
\]

Initially True:
\[
y_0 + 1 > y_0 \\
\text{State: } x = x_p - 1
\]

Initially False:
\[
y_0 + 1 \not> y_0 \\
\text{State: } x = y_0 + 1
\]

Final States:
1. \((y_0 + 1 \not> y_0) \implies (x = y_0 + 1)\)
2. \((y_0 + 1 > y_0)\), penultimately true, finally false \implies \text{ new state}
While Loops

Initially True:
\[ y_0 + 1 > y_0 \]
State: \( x = x_p - 1 \)

\[ x = y + 1; \]
{ \( \top \Rightarrow (x = y_0 + 1) \) }

Initially False:
\[ y_0 + 1 \not> y_0 \]
State: \( x = y_0 + 1 \)

\[ \text{while } (x > y) \]
\[ x = x - 1; \]

Final States:
1. \( (y_0 + 1 \not> y_0) \Rightarrow (x = y_0 + 1) \)
2. \( (y_0 + 1 > y_0), (x_p > y_0), \text{finally false} \Rightarrow \text{new state} \)
While Loops

Initially True:

\[ y_0 + 1 > y_0 \]
State: \( x = x_p - 1 \)

Initially False:

\[ y_0 + 1 \not> y_0 \]
State: \( x = y_0 + 1 \)

Final States:

1. \((y_0 + 1 \not> y_0) \implies (x = y_0 + 1)\)
2. \((y_0 + 1 > y_0), (x_p > y_0), (x_p - 1 \not> y_0) \implies \text{new state}\)
While Loops

Initially True:

\( y_0 + 1 > y_0 \)

State: \( x = x_p - 1 \)

\[
\{ \top \implies (x = y_0 + 1) \}
\]

Initially False:

\( y_0 + 1 \not> y_0 \)

State: \( x = y_0 + 1 \)

while \((x > y)\)

\[
\begin{align*}
x &= x - 1; \\
\end{align*}
\]

Final States:

1. \((y_0 + 1 \not> y_0) \implies (x = y_0 + 1)\)
2. \((y_0 + 1 > y_0), (x_p > y_0), (x_p - 1 \not> y_0) \implies (x = x_p - 1)\)
While Loops

\[
x = y + 1;
\]
\[
\begin{align*}
\{ \top \} & \implies (x = y_0 + 1) \\
\end{align*}
\]

Initially True:
- \( y_0 + 1 > y_0 \)
- State: \( x = x_p - 1 \)

Initially False:
- \( y_0 + 1 \not> y_0 \)
- State: \( x = y_0 + 1 \)

Initial States:
- \( y_0 + 1 > y_0 \)
- State: \( x = x_p - 1 \)
- \( y_0 + 1 \not> y_0 \)
- State: \( x = y_0 + 1 \)

Final States:
1. \((y_0 + 1 \not> y_0) \implies (x = y_0 + 1)\)
2. \((y_0 + 1 > y_0), (x_p > y_0), (x_p - 1 \not> y_0) \implies (x = x_p - 1)\)
3. If we can show that neither path condition is true...
While Loops

\[ x = y + 1; \]
\[ \{ \top \implies (x = y_0 + 1) \} \]

Initially True:
\[ y_0 + 1 > y_0 \]
State: \[ x = x_p - 1 \]

Initially False:
\[ y_0 + 1 \not> y_0 \]
State: \[ x = y_0 + 1 \]

Final States:
1. \[ (y_0 + 1 \not> y_0) \implies (x = y_0 + 1) \]
2. \[ (y_0 + 1 > y_0), (x_p > y_0), (x_p - 1 \not> y_0) \implies (x = x_p - 1) \]
3. If we can show that neither path condition is true, then we know that the loop does not terminate
**Conditioned “while”**

Initially True:

\[ y_0 + 1 > y_0 \]

State: \( x = x_p + 1 \)

\[
\begin{align*}
\text{x} &= y + 1; \\
\{ \top \} &\implies (x = y_0 + 1) \\
\text{while } (x > y) &\\
\text{x} &= x - 1;
\end{align*}
\]

Initially False:

\[ y_0 + 1 \not> y_0 \]

State: \( x = y_0 + 1 \)

Final States:

\[
\begin{align*}
\{ (y_0 + 1 \not> y_0) \} &\implies (x = y_0 + 1), \\
(y_0 + 1 > y_0), (x_p > y_0), (x_p - 1 \not> y_0) &\implies (x = x_p - 1)
\end{align*}
\]
Conditioned "while"

Initially True:
\[ y_0 + 1 > y_0 \]
State: \[ x = x_p + 1 \]

Initially False:
\[ y_0 + 1 \neq y_0 \]
State: \[ x = y_0 + 1 \]

Final State:
\[ (y_0 + 1 > y_0), (x_p > y_0), (x_p - 1 \neq y_0) \implies (x = x_p - 1) \]
Comments on “while”

- Although (in this case) we have not simplified the loop, we have gained some information that can be used when conditioning statements which follow the loop:
Although (in this case) we have not simplified the loop, we have gained some information that can be used when conditioning statements which follow the loop:

- We know that the loop will be executed at least once.
Comments on “while”

- Although (in this case) we have not simplified the loop, we have gained some information that can be used when conditioning statements which follow the loop:
  - We know that the loop will be executed at least once.
  - We know that the loop terminates.
Comments on “while”

- Although (in this case) we have not simplified the loop, we have gained some information that can be used when conditioning statements which follow the loop:
  - We know that the loop will be executed at least once.
  - We know that the loop terminates.
  - If we were to add the statement `p=5` within the loop body, and the loop was then followed by a conditional `if (p=5) s`, then the system can determine that the statement `s` would be executed.
Comments on “while”

- Although (in this case) we have not simplified the loop, we have gained some information that can be used when conditioning statements which follow the loop:
  - We know that the loop will be executed at least once.
  - We know that the loop terminates.
  - If we were to add the statement p=5 within the loop body, and the loop was then followed by a conditional if (p=5) s, then the system can determine that the statement s would be executed.

*Although a programmer might not put a statement of the form p=5 within the loop body, it might have arisen as a result of conditioning the loop body.*
In the example given, the system can determine that the final value of $x$ is less than or equal to the initial value of $y$, and that $x + 1$ (i.e. the penultimate value of $x$) is greater than the initial value of $y$. 
Comments on “while”

- In the example given, the system can determine that the final value of $x$ is less than or equal to the initial value of $y$, and that $x + 1$ (i.e. the penultimate value of $x$) is greater than the initial value of $y$.

- This helps us to simplify any condition involving $x$ and $y$ that follows the loop.
In the example given, the system can determine that the final value of \( x \) is less than or equal to the initial value of \( y \), and that \( x + 1 \) (i.e. the penultimate value of \( x \)) is greater than the initial value of \( y \).

This helps us to simplify any condition involving \( x \) and \( y \) that follows the loop.

In the example given, if \( x \) and \( y \) are integers, then we know that \( x = y \) when the loop terminates.
In the example given, the system can determine that the final value of $x$ is less than or equal to the initial value of $y$, and that $x + 1$ (i.e. the penultimate value of $x$) is greater than the initial value of $y$.

This helps us to simplify any condition involving $x$ and $y$ that follows the loop.

In the example given, if $x$ and $y$ are integers, then we know that $x = y$ when the loop terminates.

This analysis of loops appears to be more powerful than in any other published work in symbolic execution.
Summary
Summary

- Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.
Summary

- Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.
- Our `assert` statement (now not unusual) simplifies the system and creates additional expressiveness.
Summary

- Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.
- Our assert statement (now not unusual) simplifies the system and creates additional expressiveness.
- The conditioner can be modified so that asserts can be used to establish preconditions and check postconditions.
Summary

- Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.
- Our assert statement (now not unusual) simplifies the system and creates additional expressiveness.
- The conditioner can be modified so that asserts can be used to establish preconditions and check postconditions.
- We have generalised conditioning to the “backward” case.
Summary

- Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.
- Our `assert` statement (now not unusual) simplifies the system and creates additional expressiveness.
- The conditioner can be modified so that asserts can be used to establish preconditions and check postconditions.
- We have generalised conditioning to the “backward” case.
  - Removes code that does not contribute to the specified outcome.
Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.

Our assert statement (now not unusual) simplifies the system and creates additional expressiveness.

The conditioner can be modified so that asserts can be used to establish preconditions and check postconditions.

We have generalised conditioning to the “backward” case.
  - Removes code that does not contribute to the specified outcome.
  - Potentially useful in combination with forward conditioning;
Summary

- Developed the first implementation of a fully automatic conditioned slicer (ConSIT) using SVC (and Isabelle) for theorem proving.
- Our assert statement (now not unusual) simplifies the system and creates additional expressiveness.
- The conditioner can be modified so that asserts can be used to establish preconditions and check postconditions.
- We have generalised conditioning to the “backward” case.
  - Removes code that does not contribute to the specified outcome.
  - Potentially useful in combination with forward conditioning; *forward conditioning* on the pre-conditions and *backward conditioning* on the negation of the post-conditions can isolate those code fragments that might contribute to out-of-specification behaviour.
Related and Other Work
ConSUS [David Daoudi] a WSL (Martin Ward) version of a conditioned slicer using WSL’s built in *simplify* and also *CVC*.
ConSUS [David Daoudi] a WSL (Martin Ward) version of a conditioned slicer using WSL’s built in simplify and also CVC.

VADA: variable dependence analyser for “ANSI C” [with Harman and DaimlerChrysler].
Related and Other Work

- ConSUS [David Daoudi] a WSL (Martin Ward) version of a conditioned slicer using WSL’s built in \textit{simplify} and also \textit{CVC}.
- VADA: variable dependence analyser for “ANSI C” [with Harman and DaimlerChrysler].
  - Uses slicing techniques to determine which variables determine the truth-value of a specified expression.
Related and Other Work

- ConSUS [David Daoudi] a WSL (Martin Ward) version of a conditioned slicer using WSL’s built in `simplify` and also CVC.
- VADA: variable dependence analyser for “ANSI C” [with Harman and DaimlerChrysler].
  - Uses slicing techniques to determine which variables determine the truth-value of a specified expression.
  - This helps to constrain the search space in Daimler’s evolutionary test-generation system.
ConSUS [David Daoudi] a WSL (Martin Ward) version of a conditioned slicer using WSL’s built in `simplify` and also `CVC`.

VADA: variable dependence analyser for “ANSI C” [with Harman and DaimlerChrysler].
- Uses slicing techniques to determine which variables determine the truth-value of a specified expression.
- This helps to constrain the search space in Daimler’s evolutionary test-generation system.

Restructing transformations for testing: translate code with (multi-level) `break` statements into “pure” structured code, whilst preserving feasible paths [with Hierons and Harman].
ConSUS [David Daoudi] a WSL (Martin Ward) version of a conditioned slicer using WSL’s built in \textit{simplify} and also \textit{CVC}.

VADA: variable dependence analyser for “ANSI C” [with Harman and DaimlerChrysler].
  - Uses slicing techniques to determine which variables determine the truth-value of a specified expression.
  - This helps to constrain the search space in Daimler’s evolutionary test-generation system.

Restructing transformations for testing: translate code with (multi-level) break statements into “pure” structured code, whilst preserving feasible paths [with Hierons and Harman].
  1. A test-set for the original programming will have the same class of coverage (Statement, Branch, MCDC etc) for the structured version of the program.
  2. Increases the applicability of tools and techniques for testing and analysis.
Current Activities

- We are currently working on an analysis of conditioning [Arthorn Luangsodsai] and slicing [Lahcen Ouarbya] as semantic preserving transformations.
Current Activities

- We are currently working on an analysis of conditioning [Arthorn Luangsodsai] and slicing [Lahcen Ouarbya] as semantic preserving transformations.
- We are exploring the use of conditioning for specification-based testing [Hierons].
Current Activities

- We are currently working on an analysis of conditioning [Arthorn Luangsodsai] and slicing [Lahcen Ouarbya] as semantic preserving transformations.
- We are exploring the use of conditioning for specification-based testing [Hierons].
- There is a relationship between conditioned slicing and refinement (Chung, Lee, Yoon and Kwon) which merits further exploration [Voelkner].
The End