# From Source to Graphs — Compiling SMV and Verus

Sérgio Campos



## SMV and Verus

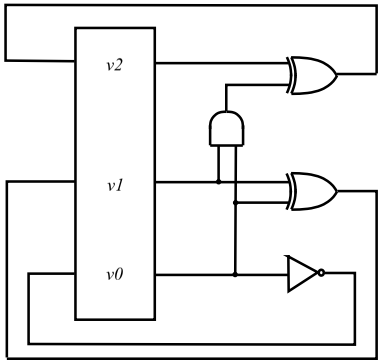
Simplified languages that represent two types:

| **SMV** | **Verus** |
|---|---|
| Declarative | Imperative |
| Hardware | Software |
| Parallel | Sequential |

```
MODULE main              main() {
VAR a: boolean;          boolean a;
INIT(a) := 0;
NEXT(a) := case          a = 0;
             b: !a;      if (b) a = !a;
           esac;         wait(1);
```



## SMV Example: Mod8counter




- $N_0 = (v_0' \Leftrightarrow \neg v_0)$
- $N_1 = (v_1' \Leftrightarrow v_0 \oplus v_1)$
- $N_2 = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$



## SMV Example: Mod8counter

```
MODULE main
var v0, v1, v2: boolean;

INIT(v0) := 0;
INIT(v1) := 0;
INIT(v2) := 0;

NEXT(v0) := !v0;
NEXT(v1) := v0 xor v1;
NEXT(v2) := (v0 && v1) xor v2;
```

## Generating the Transition Relation and Initial State

Transition relation:

- $N_0 = (v_0' \Leftrightarrow \neg v_0)$
- $N_1 = (v_1' \Leftrightarrow v_0 \oplus v_1)$
- $N_2 = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$

$$N = N_0 \wedge N_1 \wedge N_2$$

Initial state set:

$$Init = \neg v_0 \wedge \neg v_1 \wedge \neg v_2$$





## Transition X Invariant Relation

1. Long expression goes into transition relation:

```
ASSIGN
  NEXT(a) := case
               long_complex_expression;
             esac;
```

2. Long expression goes into invariant relation:

```
ASSIGN
  anext := long_complex_expression;

  NEXT(a) := anext;
```

$$EX(f) := \underbrace{Invar}_{1} \wedge (\exists \bar{v}' \left[ f(\bar{v}') \wedge \underbrace{N(\bar{v}, \bar{v}')}_{2} \right)$$

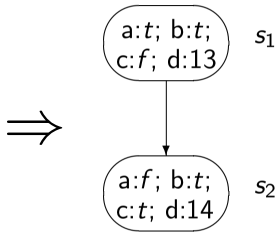
- Moving expressions between *Invar* and $N$ can affect performance significantly!





## Compiling Verus

```
...
wait(1);
a = false;
c = true;
d = d + 1;
wait(1);
...
```



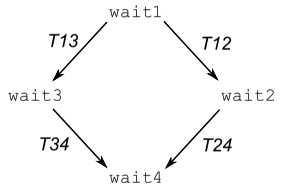

If $s_0$ is the current state when the first wait is executed, then $s_1$ will be the current state when the second wait is reached.





## Wait Graphs

```
001 wait1(1);
002 S1;
003 if (cond) {
004   S2;
005   wait2(1);
006   S3;
007 } else {
008   S4;
009   wait3(1);
010   S5;
011 };
012 S6;
013 wait4(1);
```

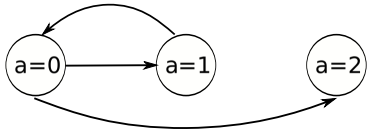



- Intermediate representation between source code and state-transition graph.
- Each node is a wait in the program.
- Transitions correspond to the control flow.
- Transitions are labelled by relations describing modifications to the state.

## Where do we come from ? Where do we go ?

Imagine the following program:

```
...
a=0;
wait(1);
a=1;
wait(1);
a=0;
wait(1);
a=2;
wait(1);
...
```




What about property: $\mathbf{AG}(a = 0 \rightarrow \mathbf{EX}\, a = 2)$ ?





## Wait Counters

Relation $T_{ij}$ in the wait graph contains no information about the value of $i$ and $j$.
A special variable is introduced in the model to maintain this information, the *wait counter*.

- An assignment $wc = j$ is introduced before $wait_j$.
  - This fixes the destination of the transition in $T_{ij}$.
- The semantics of wait determines the value of the current state wait counter to be $i$ after $wait_i$.
  - This fixes the origin of the transition in $T_{ij}$.

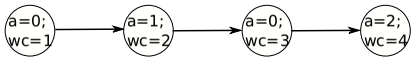
All relations $T_{ij}$ now contain information about its origin and its destination.





## Wait Counters

```
...
a=0;
wc'=1; wait(1); wc=1;
       a=1;
wc'=2; wait(1); wc=2;
       a=0;
wc'=3; wait(1); wc=3;
       a=2;
wc'=4; wait(1); wc=4;
...
```




Transitions are:

$$\begin{aligned} N_1 &= wc = 1 \wedge a = 0 \wedge a' = 1 \wedge wc' = 2 \\ N_2 &= wc = 2 \wedge a = 1 \wedge a' = 0 \wedge wc' = 3 \\ N_3 &= wc = 3 \wedge a = 0 \wedge a' = 2 \wedge wc' = 4 \end{aligned}$$





## Verus Semantics

Given a program $P$:

$$R[\![P]\!] = \langle r, t \rangle$$

gives the transition relation as $t$. The Formula $r$ "collects" the various $T_{ij}$ between waits.

We will construct the semantics of a Verus program from the semantics of the commands:

- Wait;
- Assignments;
- Conditionals;
- Loops;

Obs: All Verus programs end with: `while (true) {};`

## Semantics of Wait

$$R[\![\texttt{wait}_i(1)]\!]\langle r, t\rangle = \langle((wc = i) \wedge \bigwedge_{\forall v \in Var} (v = v'), (t \vee r)\rangle$$

Example:

```
wait1;          ← (wc = 1) ∧ (a = a′)
a = a + 1;      ← (wc = 1) ∧ (a′ = a + 1)
wc = 2;         ← (wc = 1) ∧ (wc′ = 2) ∧ (a′ = a + 1)
wait2;          ← (wc = 2) ∧ (a = a′)
```

- All waits are numbered. `wait2` is the second wait in the program.
- The red formula is a complete transition




## Semantics of Assignments

Variables are referenced by the *next state* variables!

- This guarantees that the most recent value is used.

If we use $v$ instead of $v'$ we get an outdated value:

```
v = true;       ← v′
x = !v;         ← x′ = ¬v′
wait;
```

Even if no value is assigned it works, because of the wait:

```
v = false;      ← ¬v′
wait;           ← v = v′
x = !v;         ← (x′ = ¬v′ ∧ v = v′) ≡ (x′ = ¬v)
wait;
```




## Semantics of Assignments

$$R[\![\texttt{v} = expr]\!]\langle r, t\rangle = \langle(\exists y[v = Expr^{y}/_{v} \wedge r^{y}/_{v}]), t\rangle,$$

where $v = E[\![v]\!]$, $Expr = E[\![expr]\!]$ and $y$ is a new variable.

Example:

```
wait;
a = a + 1;
```

- After wait $r = (a = a')$
- $E[\![\texttt{v}]\!] = a'$ $\quad E[\![\texttt{a} + 1]\!] = a' + 1$

$$\exists y[\ \ a' = (a' + 1)^{y}/_{a'} \wedge (a = a')^{y}/_{a'}]$$
$$\exists y[\ \ a' = (y + 1) \wedge (a = y)]$$
$$a' = a + 1$$




## Conditionals

$$R[\![\texttt{if } cond\ S_1 \texttt{ else } S_2]\!]\langle r, t\rangle = \langle r' \vee r'', t' \vee t''\rangle$$

Where

$$\begin{aligned}\langle r', t'\rangle &= R[\![S_1]\!]\langle(r \wedge cond), t\rangle\\ \langle r'', t''\rangle &= R[\![S_2]\!]\langle(r \wedge \neg cond), t\rangle\end{aligned}$$

Example:

```
                        ← a = a′
if (a < 8) {            ← a = a′ ∧ a′ < 8
   a = a + 1;           ← a′ = a + 1 ∧ a < 8
} else {
                        ← a = a′ ∧ a′ ≥ 8
   a = 0;               ← a′ = 0 ∧ a ≥ 8
}                       ← ((a′ = a + 1) ∧ (a < 8))∨
                          ((a′ = 0) ∧ (a ≥ 8))
```

## While Loop

Two versions:

- Understandable:

$$R[\![\texttt{while}\ cond\ S_1]\!] = R[\![\texttt{if}\ cond\{S_1; \texttt{while}\ cond\ S_1\}]\!]$$

- Correct:

$$R[\![\texttt{while}\ cond\ S_1]\!] = \mathrm{fix}(\lambda f\, \lambda\langle r, t\rangle.\langle r' \vee (r \wedge \neg cond), t \vee t'\rangle$$

where $\langle r', t'\rangle = f(R[\![S_1]\!]\langle r \wedge cond, t\rangle$.





## Conclusion

From source code we can:

- Generate state graphs;
- Symbolically;
- Imperative and declarative languages





## Conclusion

The whole process:

1. Model the system being verified
   - Generate code in a tool language
2. Specify properties
   - Temporal logic
3. Compile program into state graphs
4. Traverse graph
5. Identify errors and print counterexamples.

After all of that is done, *repeat!*