Formal Verification of a Synchronous Data-flow Compiler: from Signal to C

Van-Chan Ngo | INRIA Rennes, France

Jean-Pierre Talpin | Advisor
INRIA Rennes, France

PhD Defense

Construct a translation validation-based verification framework to check the correctness of the synchronous data-flow compiler, Signal.

Agenda

Motivation
Related Work
Approach

Cock Semantics Preservation
Data Dependency Preservation
Value-Equivalence Preservation

Detected Bugs
Conclusion

Motivation

Compiles into (b ==0) & (c != 0)

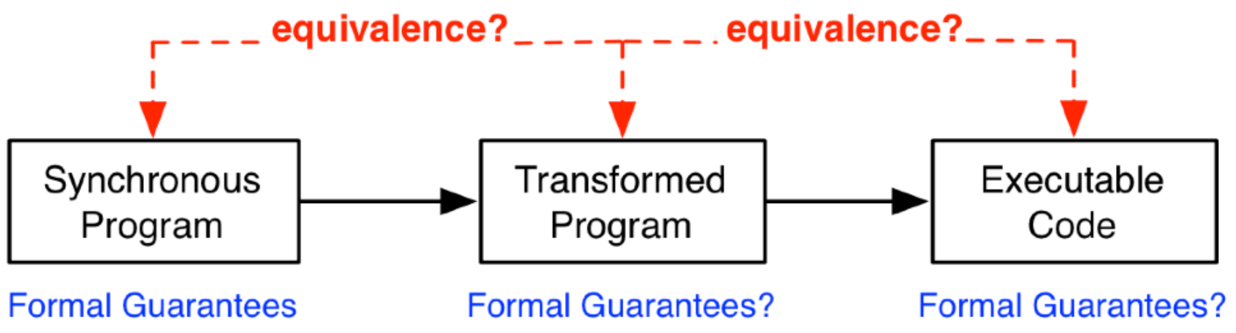
```
int lt (_Bool b, unsigned char c) {
    return b < c;
}

int main () {
    if (!lt(1, 'a'))
        abort();
}
```

Program always aborts

Compilers always might have bugs!

Development of critical software



- Safety requirements have to be implemented correctly
- Formal verification is applied at source level (static analysis, model checking, proof)
- The guarantees are obtained at source program might be broken due to the compiler bugs
- Raise awareness about the importance of compiler verification in critical software development

Related work on compiler verification

- SuperTest: test and validation suite
- DO-178: certification standards
- Astrée: a static analyzer
- Static analysis of Signal programs for efficient code generation (Gamatié et al.)
- Translation validation for optimizing compiler (Berkeley, US)
- CompCert: a certified C compiler (Inria, France)
- Verified LLVM compiler (Harvard, US)

Compiler verification

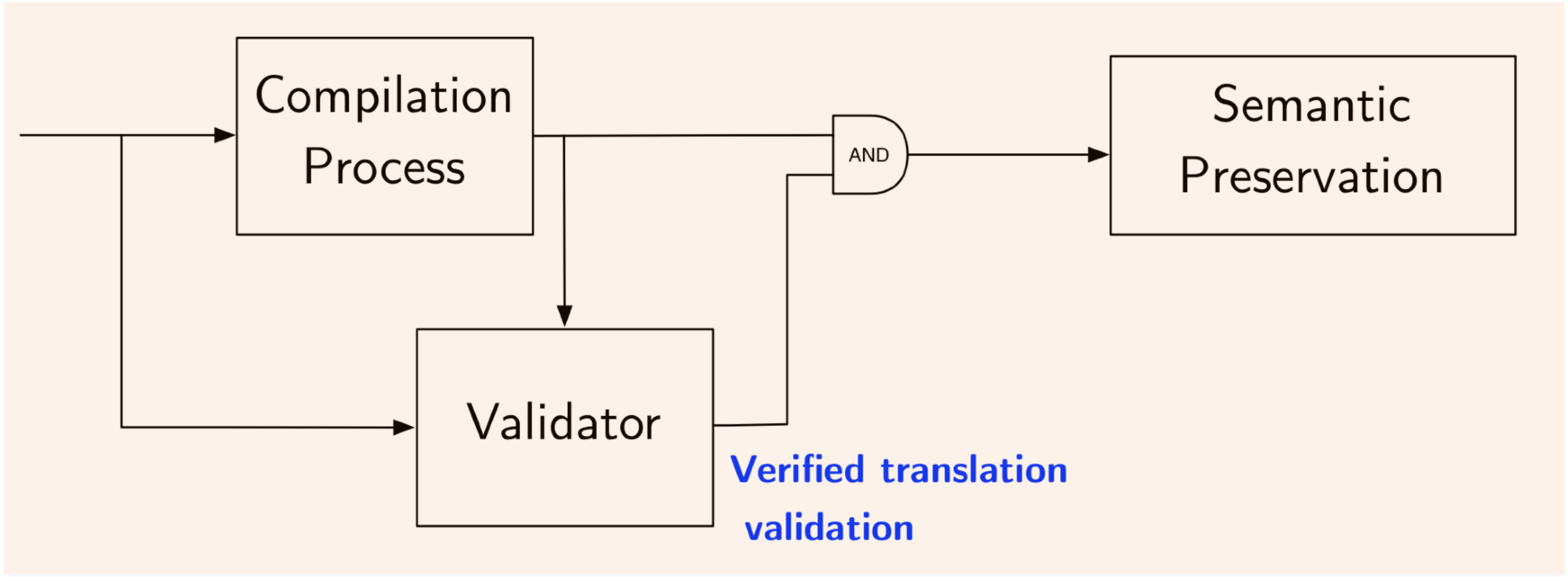
Testing-based approach

- Test and validation suite to verify compilers
- Test suite to qualify the compiler’s output

Formal method-based approach

- Formal verification of compilers
- Formal verification of compiler’s output
- Translation validation to check the correctness of the compilation

Translation validation



- Takes the source and compiled programs as input
- Checks that the source program semantics is preserved in the compiled program

Translation validation: Main components

Model builder

- Defines common semantics
- Captures the semantics of the source and compiled programs

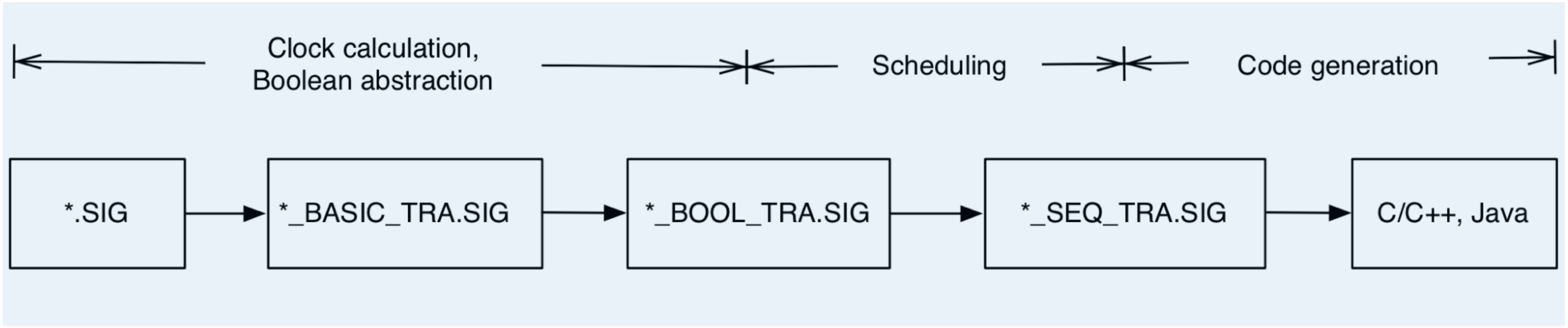
Analyzer

- Formalizes the notion of “correct implementation”
- Provides an automated proof method
- Generates a proof scripts or a counter-example

Translation validation: Features

- Avoiding redoing the proof with changes of compiler
- Independence of how the compiler works
- Less to prove (in general, the validator is much more simple than the compiler)
- Verification process is fully automated

Signal compiler



- Syntax and type checking
- Clock analysis
- Data dependency analysis
- Executable code generation

Objective

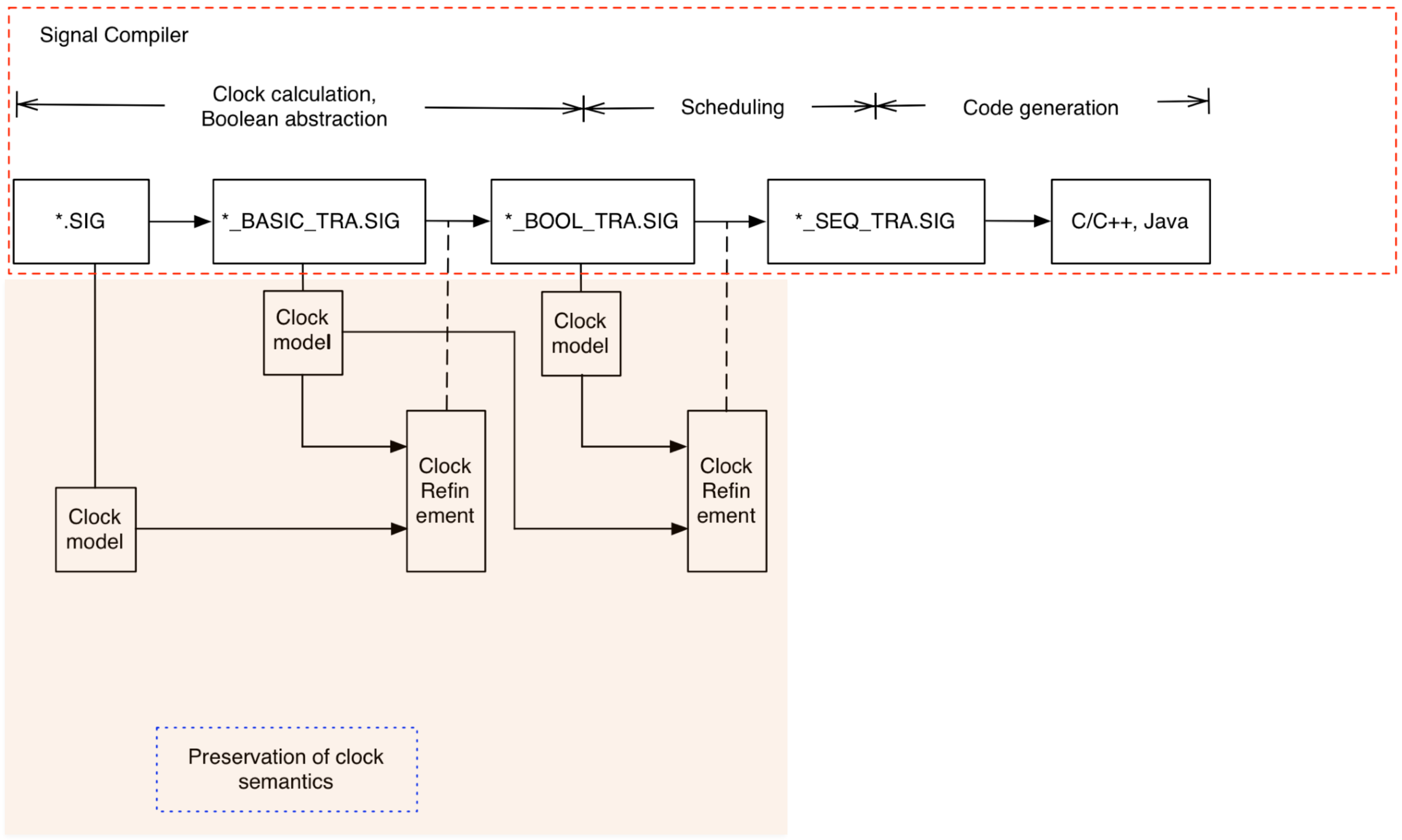
A method to formally verify the Signal compiler such that

- light weight
- scalable: deals with 500K lines of code of the implementation
- modularity
- accuracy: the proof is separated w.r.t the data structure (clock, data dependency, value-equivalence)

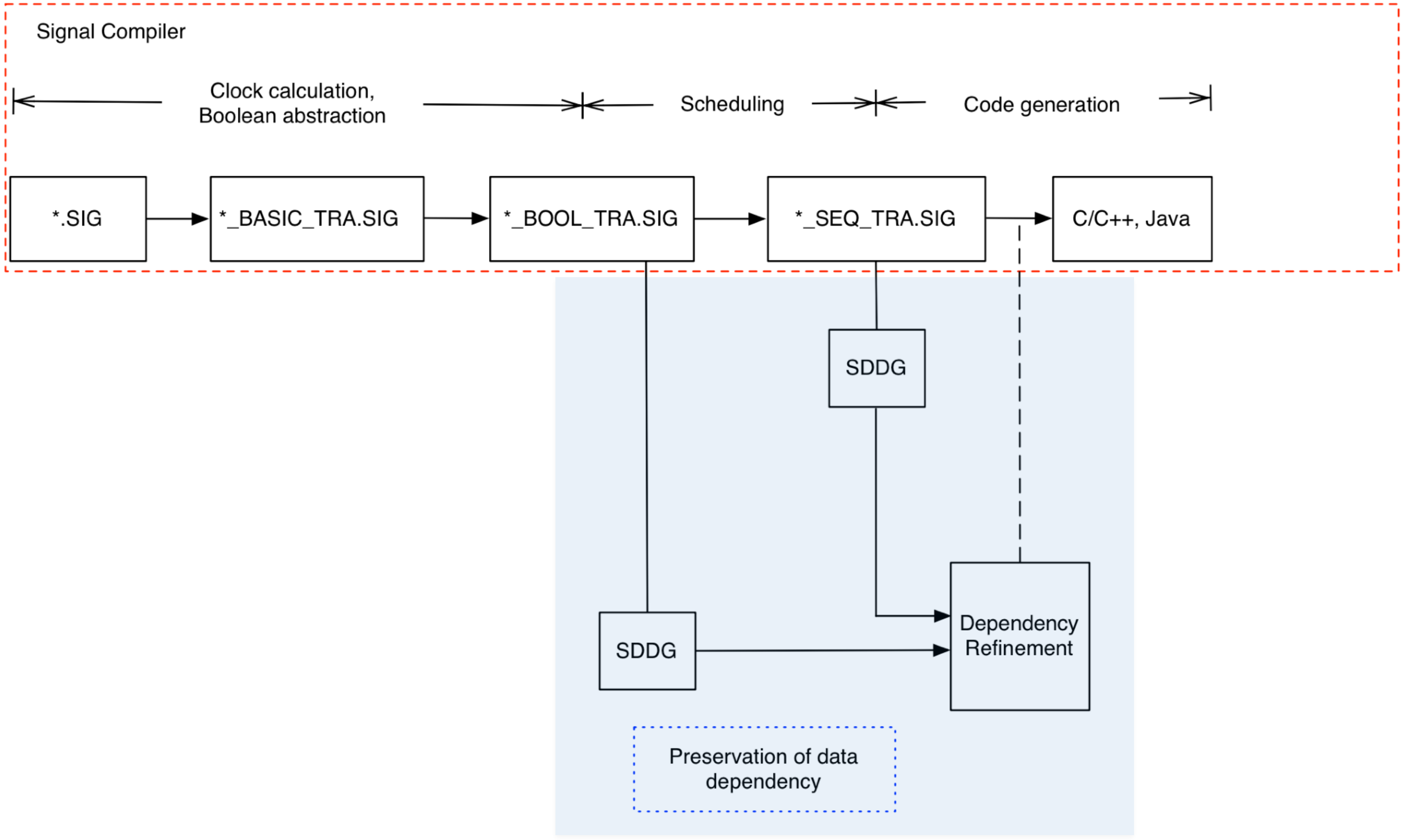
Approach

- Adopt translation validation approach
- Prove the correctness of each phase w.r.t the data structure carrying the semantics relevant to that phase
- Decompose the preservation of the semantics into the preservation of clock semantics, data dependency, and value-equivalence

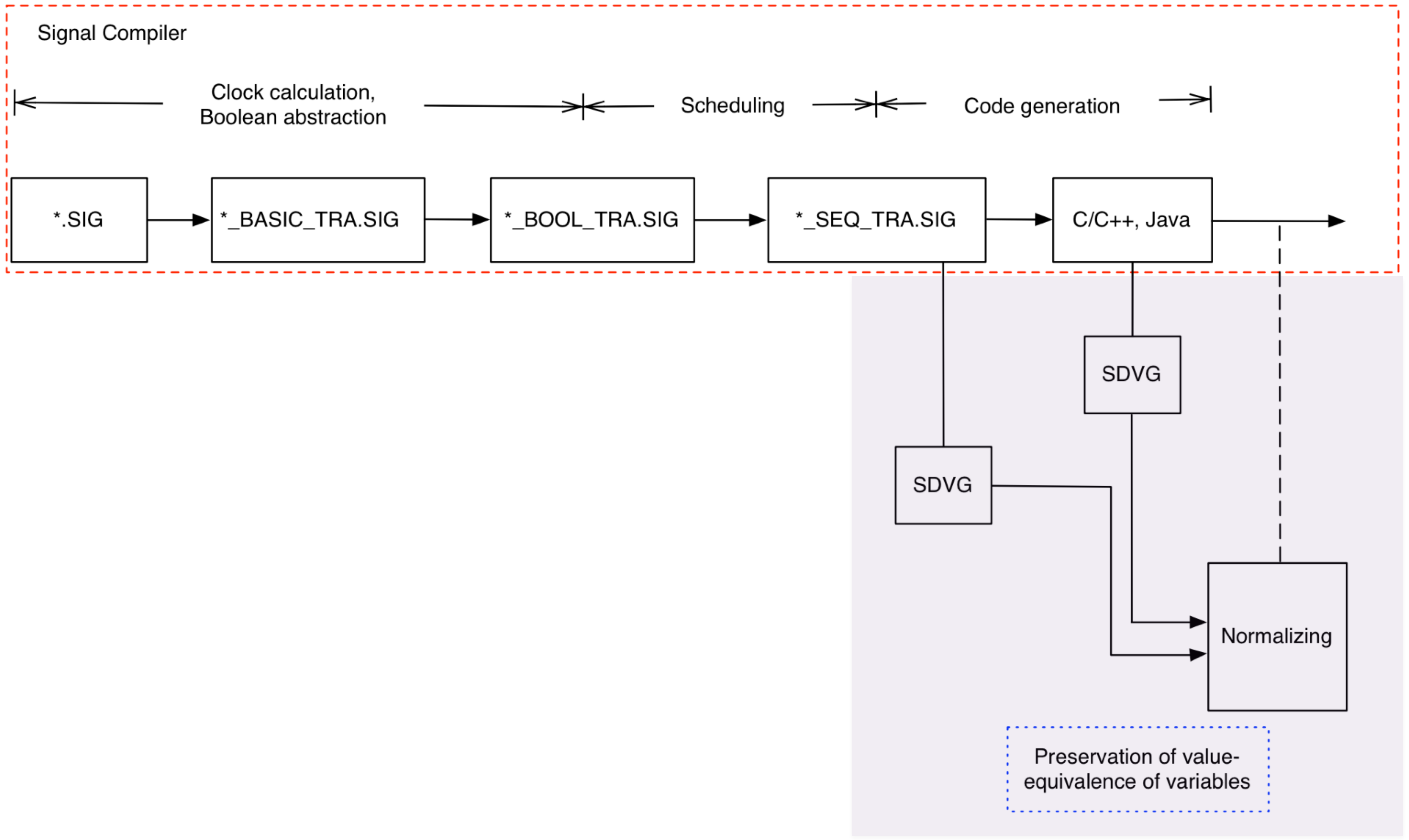
Formally verified Signal compiler



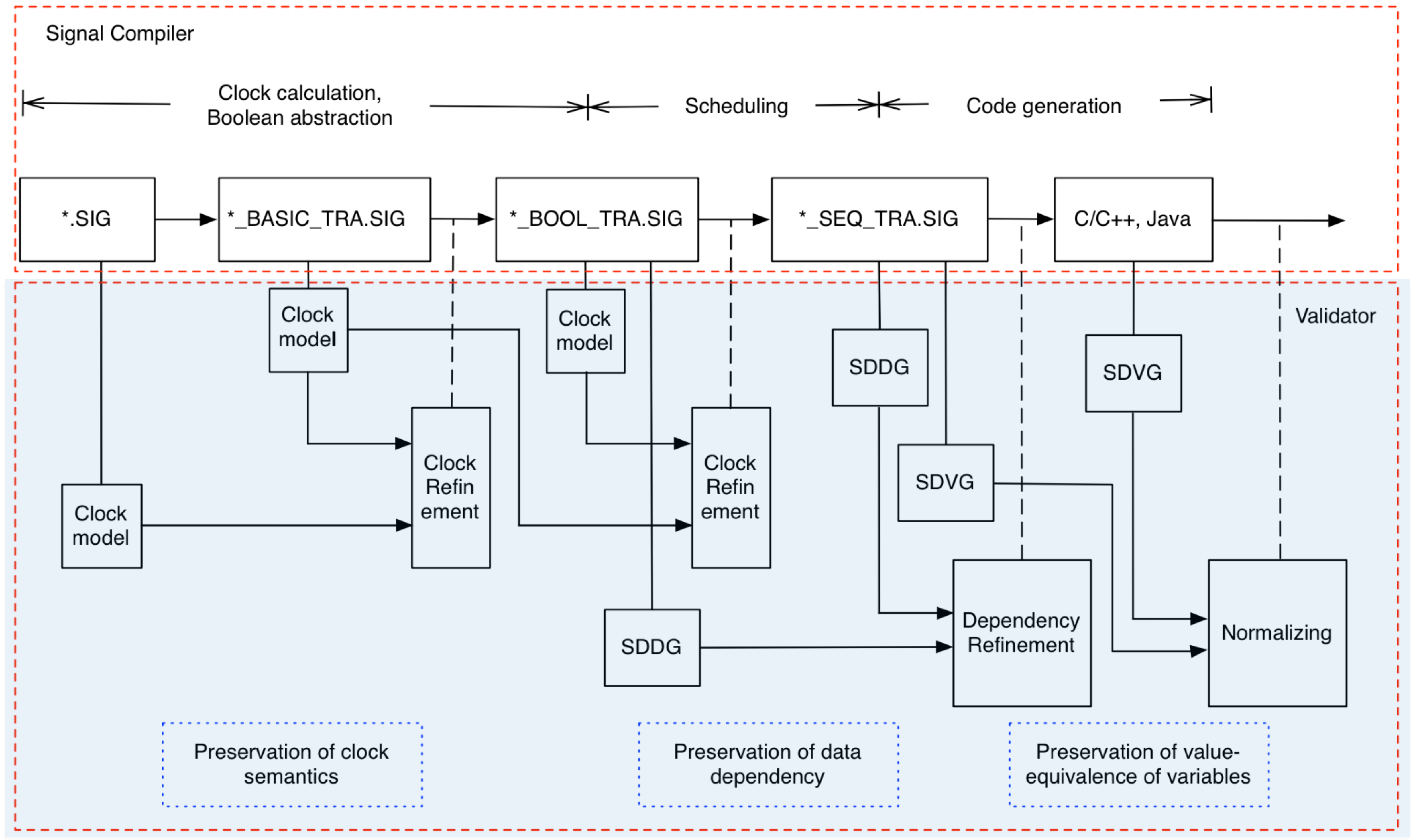
Formally verified Signal compiler



Formally verified Signal compiler



Formally verified Signal compiler



Signal language

- Signal x: sequences $x(t), t \in \mathbb{N}$ of typed values ($\perp$ is absence)
- Clock C_x of x: instants at which $x(t) \neq \perp$
- Process: set of equations representing relations between signals
- Parallelism: processes run concurrently
- Example: $y := x + 1, \forall t \in C_y, y(t) = x(t) + 1$
- Other languages: Esterel, Lustre, Scade, …

Primitive operators

- Stepwise functions: $y := f(x_1, ..., x_n)$

$$\forall t \in C_y, y(t) = f(x_1(t), ..., x_n(t)), C_y = C_{x_1} = ... = C_{x_n}$$

- Delay: $y := x\$1 \texttt{ init } a$

$$y(t_0) = a, \forall t \in C_x \wedge t > t_0, y(t) = x(t_-), C_y = C_x$$

- Merge: $y := x \texttt{ default } z$

$$y(t) = x(t) \text{ if } t \in C_x, y(t) = z(t) \text{ if } t \in C_z \setminus C_x,$$
$$C_y = C_x \cup C_z$$

Primitive operators

- Sampling: $y := x \texttt{ when } b$

$\forall t \in C_x \cap C_b \wedge b(t) = true, y(t) = x(t), C_y = C_x \cap [b]$

- Composition: $P_1 | P_2$

Denotes the parallel composition of two processes

- Restriction: $P \texttt{ where } x$

Specifies that x as a local signal to P

Example

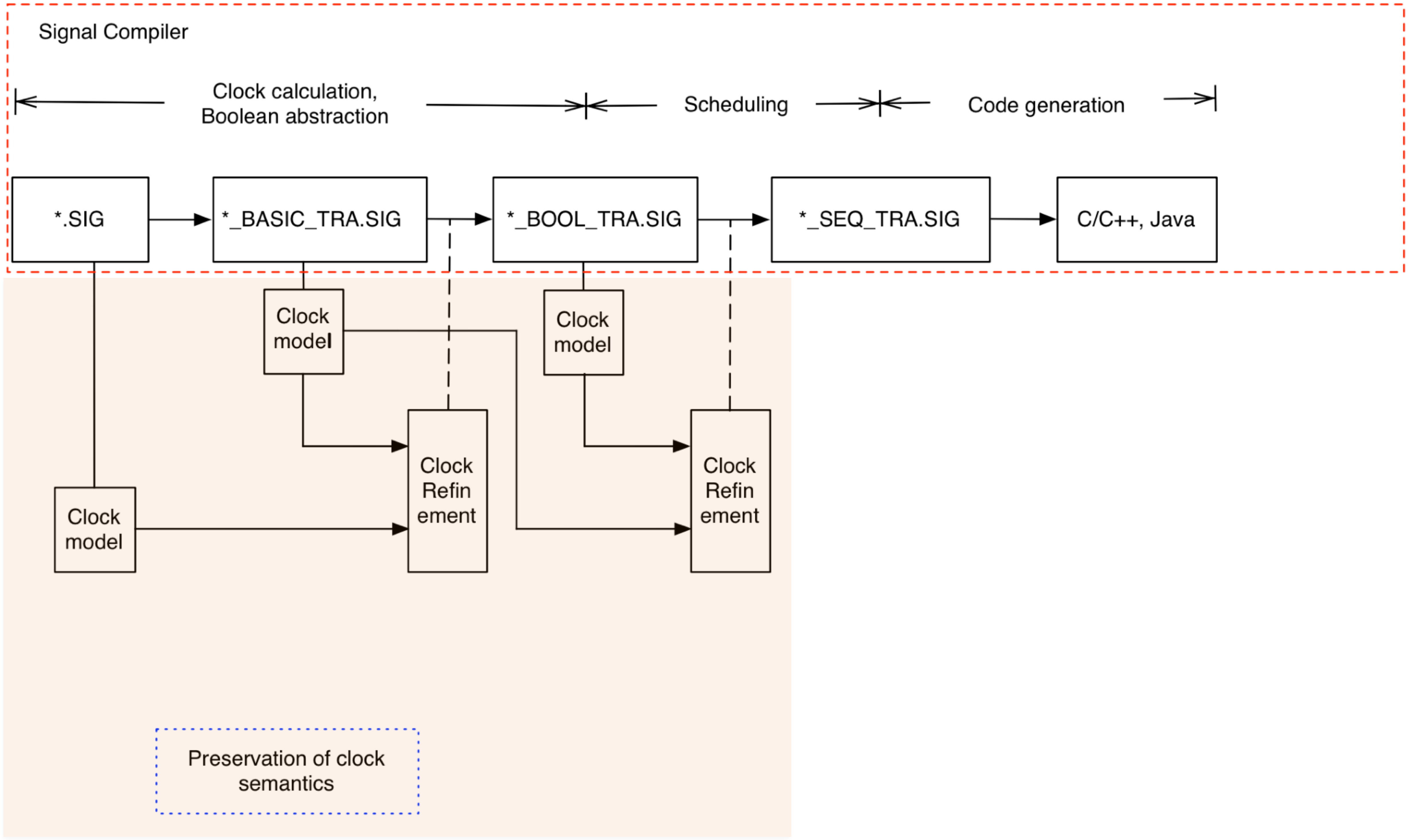
```
process DEC=
(? integer FB;! integer N) /* IO signals */
 (| FB ^= when (ZN<=1)
  | N := FB default (ZN-1) /* equations */
  | ZN := N$1 init 1        /* order does not matter */
  |)
where integer ZN end;       /* local signals */
```

- Emits a sequence of values $\texttt{FB}, \texttt{FB} - 1, \ldots, 1$

- Execution traces

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	...
FB	6	$\perp$	$\perp$	3	$\perp$	$\perp$	2	...
ZN	1	6	5	4	3	2	1	...
N	6	5	4	3	2	1	2	...

Preservation of clock semantics



Clock model

Encodes the clock

$$\phi(b := b_1 \texttt{ and } b_2) = (\widehat{b} \Leftrightarrow \widehat{b_1} \Leftrightarrow \widehat{b_2}) \wedge (\widehat{b} \Rightarrow (\widetilde{b} \Leftrightarrow \widetilde{b_1} \wedge \widetilde{b_2}))$$

Uninterpreted functions

Encodes the value

$$\phi(e := e_1 + e_2) = (\widehat{e} \Leftrightarrow \widehat{v^i_+} \Leftrightarrow \widehat{e_1} \Leftrightarrow \widehat{e_2}) \wedge (\widehat{e} \Rightarrow (\widetilde{e} = \widetilde{v^i_+}))$$

Clock model of P

$$\Phi(P) = \bigwedge_{i=1}^{n} \phi(eq_i)$$

Clock model of DEC

$FB \;\hat{}= \;\texttt{when}\ (ZN <= 1)$

$ZN := N\$1\ \texttt{init}\ 1$

$N := FB\ \texttt{default}\ (ZN - 1)$

$$
\begin{aligned}
&\ \ (\widehat{FB} \Leftrightarrow \widehat{ZN1} \wedge \widetilde{ZN1})\\
&\wedge (\widehat{ZN1} \Leftrightarrow \widehat{v^1_{<=}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN1} \Rightarrow (\widetilde{ZN1} = \widetilde{v^1_{<=}}))\\
&\wedge (\widehat{ZN} \Leftrightarrow \widehat{N}) \wedge (\widehat{ZN} \Rightarrow (\widetilde{ZN} = m.N \wedge m.N' = \widetilde{N}))\\
&\wedge (m.N_0 = 1)\\
&\wedge (\widehat{N} \Leftrightarrow \widehat{FB} \vee \widehat{ZN2}) \wedge (\widehat{N} \Rightarrow ((\widehat{FB} \wedge \widetilde{N} = \widetilde{FB})\\
&\qquad \vee(\neg\widehat{FB} \wedge \widetilde{N} = \widetilde{ZN2})))\\
&\wedge (\widehat{ZN2} \Leftrightarrow \widehat{v^1_{-}} \Leftrightarrow \widehat{ZN}) \wedge (\widehat{ZN2} \Rightarrow (\widetilde{ZN2} = \widetilde{v^1_{-}}))
\end{aligned}
$$

Clock refinement

- Clock event: A clock event is an interpretation over X. The set of clock events denoted by $\mathcal{E}c_X$

- Clock trace: A clock trace $T_c : \mathbb{N} \longrightarrow \mathcal{E}c_X$ is a chain of clock events. The concrete clock semantic of $\Phi(P)$ is a set of clock trace denoted by $\Gamma(\Phi(P))_{\setminus X}$

- Clock refinement: $\Phi(C) \sqsubseteq_{clk} \Phi(A)$ on X iff

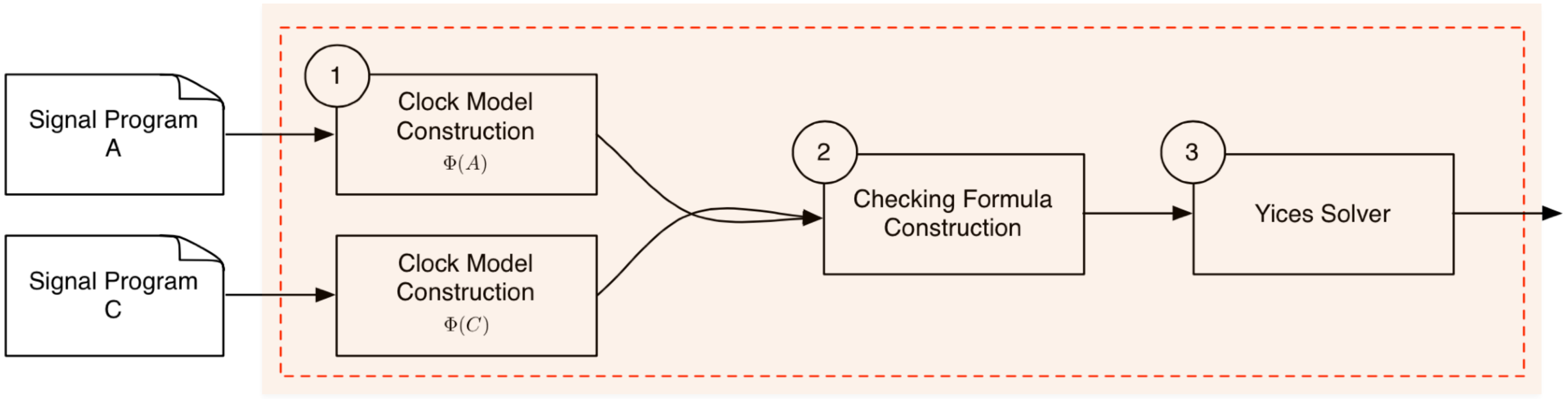
$$\forall X.T_c.(X.T_c \in \Gamma(\Phi(C))_{\setminus X} \Rightarrow X.T_c \in \Gamma(\Phi(A))_{\setminus X})$$

Proof method

- Define a variable mapping $\widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})$
- Given α, prove $\Phi(C) \sqsubseteq_{clk} \Phi(A)$ on X_{IO}

$$\frac{\begin{array}{l}\text{Premise}\\ \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}})\\ \forall \hat{I} \text{ over } \widehat{X_A} \cup \widehat{X_C}.(\hat{I} \models \Phi(C) \Rightarrow \hat{I} \models \Phi(A))\end{array}}{\begin{array}{l}\text{Conclusion}\\ \Phi(C) \sqsubseteq_{clk} \Phi(A) \text{ on } X_{IO}\end{array}}$$

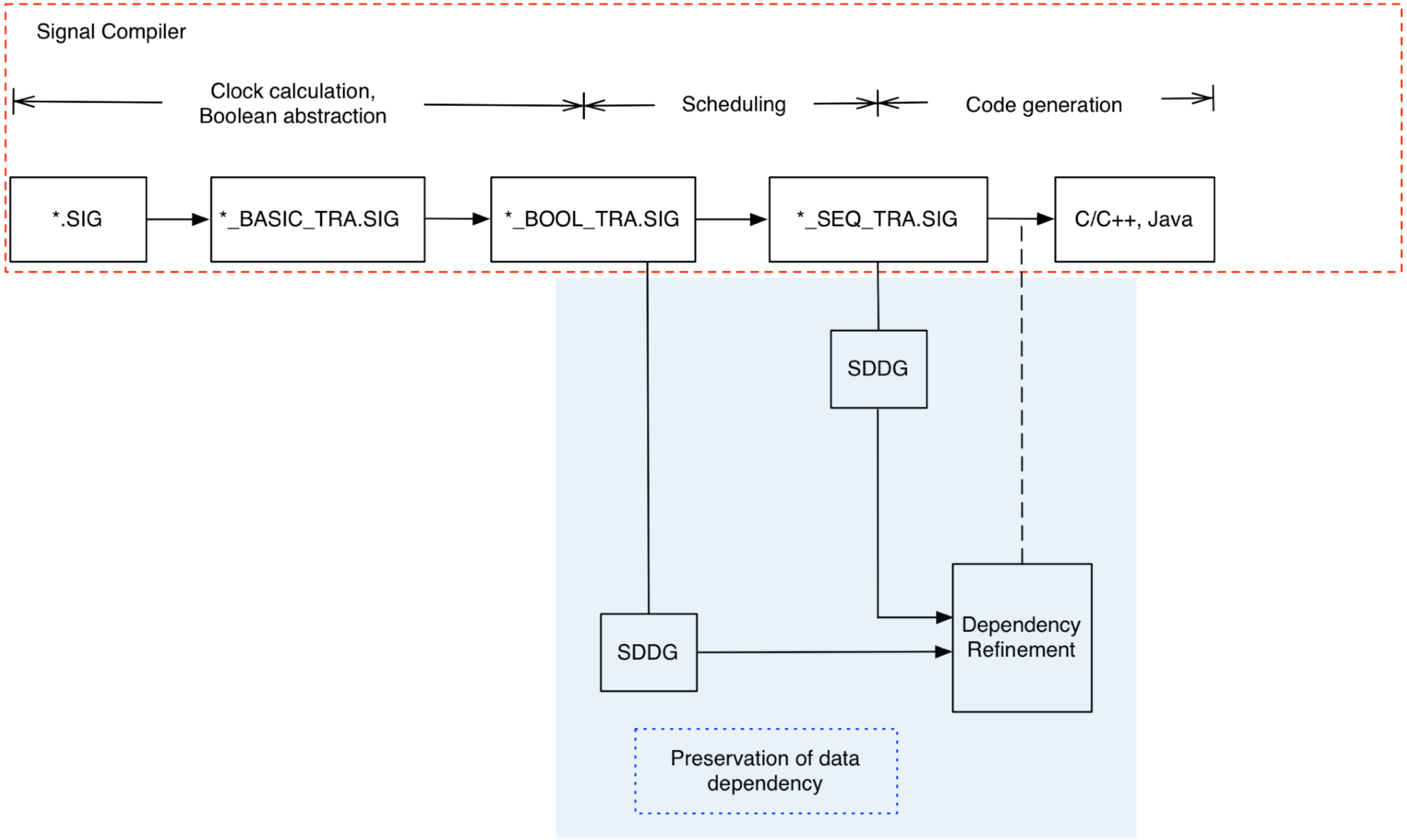
Implementation with SMT


Signal Program A
Signal Program C
1
Clock Model Construction $\Phi(A)$
Clock Model Construction $\Phi(C)$
2
Checking Formula Construction
3
Yices Solver

- Construct $\Phi(A)$ and $\Phi(C)$
- Establish $(\Phi(C) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(A))$
- Check the validity of

  $\models (\Phi(C) \wedge \widehat{X_A} \setminus \widehat{X_{IO}} = \alpha(\widehat{X_C} \setminus \widehat{X_{IO}}) \Rightarrow \Phi(A))$

Preservation of data dependency

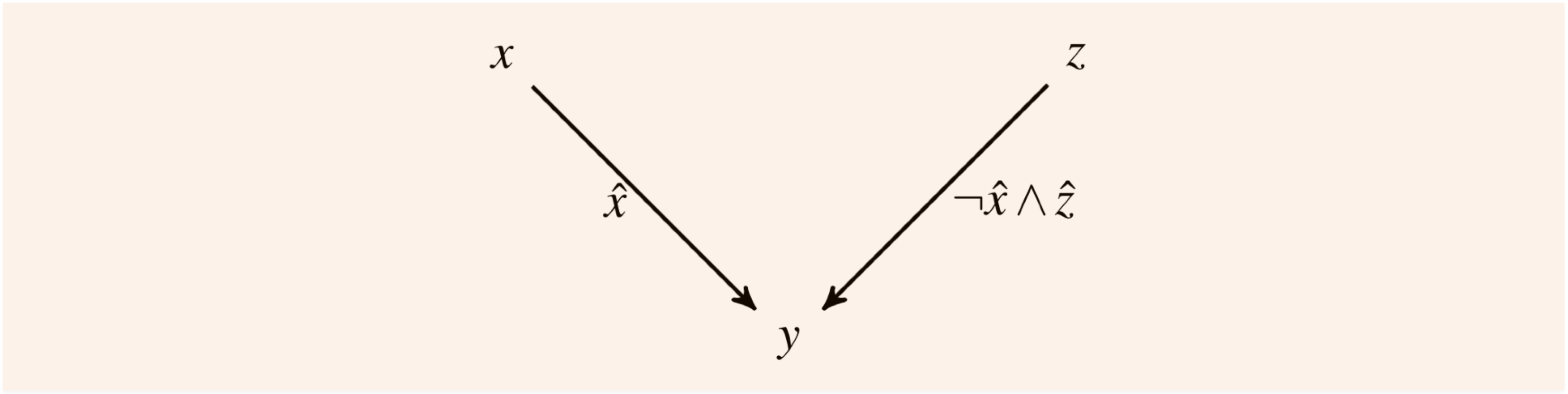


Synchronous data-flow dependency graph (SDDG)

- Data dependency is represented as a labeled directed graph
- Nodes are signals or clocks
- Edges express the dependencies among signals and clocks
- Clock constraints are first-order logic formulas to label the edges
- A dependency is effective iff its clock constraint has the value true

SDDG of primitive operators

$y := x$ `default` z


$$\begin{tikzcd} x \arrow[dr, "\hat{x}"'] & & z \arrow[dl, "\neg\hat{x} \wedge \hat{z}"] \\ & y & \end{tikzcd}$$

- y depends on x when $\hat{x} = \texttt{true}$
- Clock constraint of a dependency path

  $dp = (x_0, x_1, ..., x_n)$ is defined by $\bigwedge_{i=0}^{n-1} \widehat{c_i}$
- A cyclic path is a deadlock if and only if $M \models \bigwedge_{i=0}^{n-1} \widehat{c_i}$

SDDG of DEC

$$\begin{tikzcd}
 & & ZN \arrow[dr, "\widehat{ZN1}"] \arrow[ddrr, bend left, "\widehat{ZN2}"] & & \\
\widehat{FB} \arrow[ddr, "\widehat{FB}"'] & \widehat{ZN1} \arrow[l, "\widehat{FB}"'] \arrow[ur, "\widehat{ZN}"] \arrow[rr, "\widehat{ZN1}"] \arrow[drrr, bend right, "\widehat{ZN2}"] & & ZN1 \arrow[lll, bend left, "\widehat{FB}"] & \\
 & & & & ZN2 \arrow[ddll, "\neg\widehat{FB} \wedge \widehat{ZN2}"'] \\
 & FB \arrow[dr, "\widehat{FB}"] & & & \\
 & & N & &
\end{tikzcd}$$

```
FB ^= when (ZN <= 1)
ZN := N$1 init 1
N := FB default (ZN − 1)
```

Clock relations

$$\widehat{ZN1} = \widehat{ZN2} = \widehat{ZN} = \widehat{N}$$
$$\widehat{FB} = \widehat{ZN1} \wedge \widetilde{ZN1}$$
$$\widehat{N} = \widehat{FB} \vee \widehat{ZN2}$$

Dependency refinement

SDDG(C) is a dependency refinement of SDDG(A) if:

- For every path dp_1 in SDDG(A), there exits a path dp_2 in SDDG(C) such that dp_2 is a reinforcement of dp_1
- For every path in dp_1 SDDG(A), for any path dp_2 in SDDG(C), dp_2 is deadlock-consistent with dp_1

Reinforcement

A path dp₂ is a reinforcement of a path dp₁ iff the following formula is valid

Clock constraint of dp₂

$$\models \underbrace{\bigwedge_{i=0}^{n-1} \widehat{c_i}}_{\text{Clock constraint of dp}_1} \Rightarrow \bigwedge_{j=0}^{m-1} \widehat{c'_j}$$

At any instant, if dp₁ is effective, then dp₂ is effective too

Deadlock-consistency

A path dp₂ is a deadlock-consistent with a path dp₁ if for any path dp'₁ such that

- dp₁ and dp'₁ form a cycle path,
- then for every path dp'₂ that forms a cycle path the following formula is valid

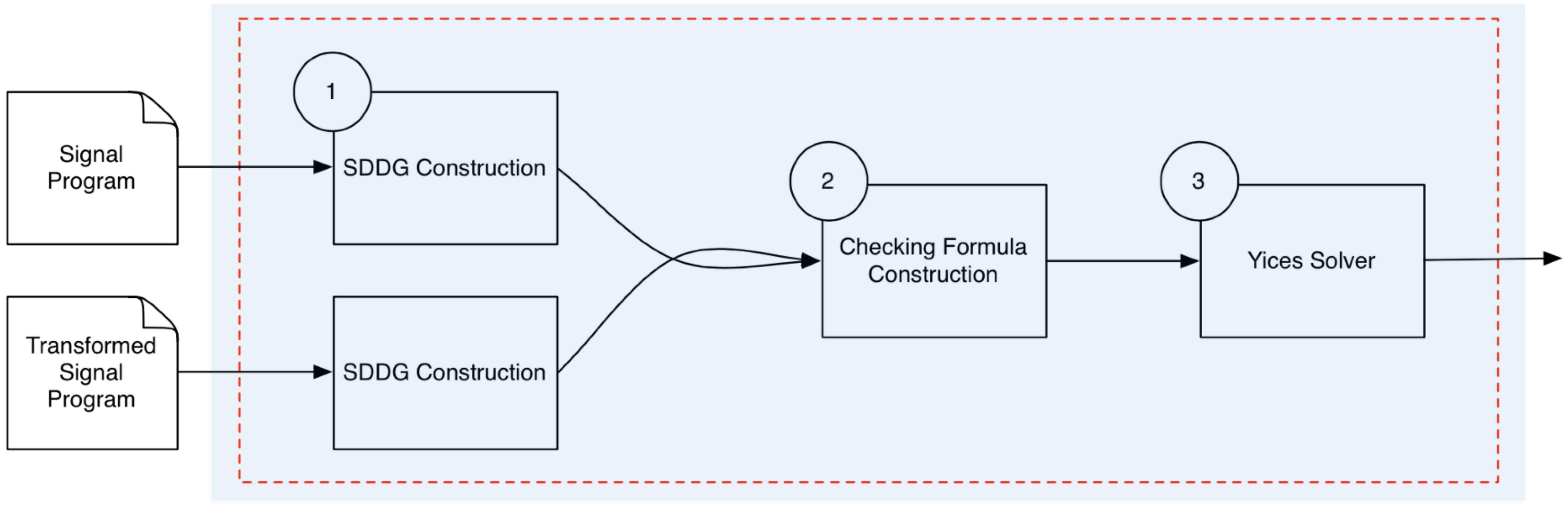
$$\models \Big(\bigwedge_{i=0}^{n-1} \widehat{c_i} \wedge \bigwedge_{j=0}^{p-1} \widehat{c_j^{inv}}\Big) \Leftrightarrow \mathtt{false} \Rightarrow \Big(\bigwedge_{k=0}^{m-1} \widehat{c'_k} \wedge \bigwedge_{l=0}^{q-1} \widehat{c_l^{inv'}}\Big) \Leftrightarrow \mathtt{false}$$

Clock constraint of (dp₁,dp'₁)

Clock constraint of (dp₂,dp'₂)

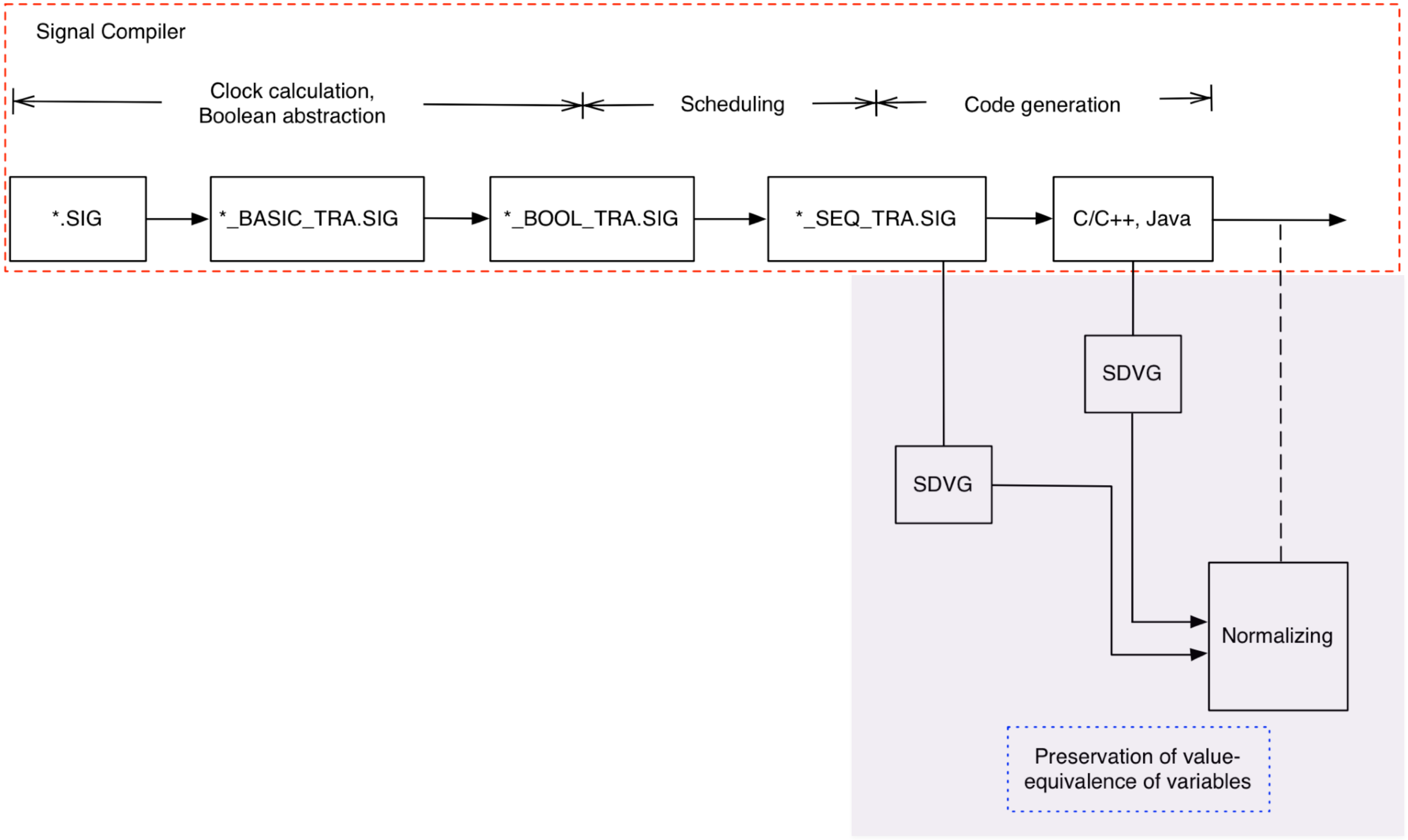
It indicates that if (dp₁, dp'₁) doesn't stand for a deadlock, then (dp₂, dp'₂) doesn't either

Implementation with SMT



- Construct SDDG(A) and SDDG(C)
- Establish the formulas for checking the reinforcement and deadlock-consistency
- Check the validity of the checking formulas

Preservation of value-equivalence



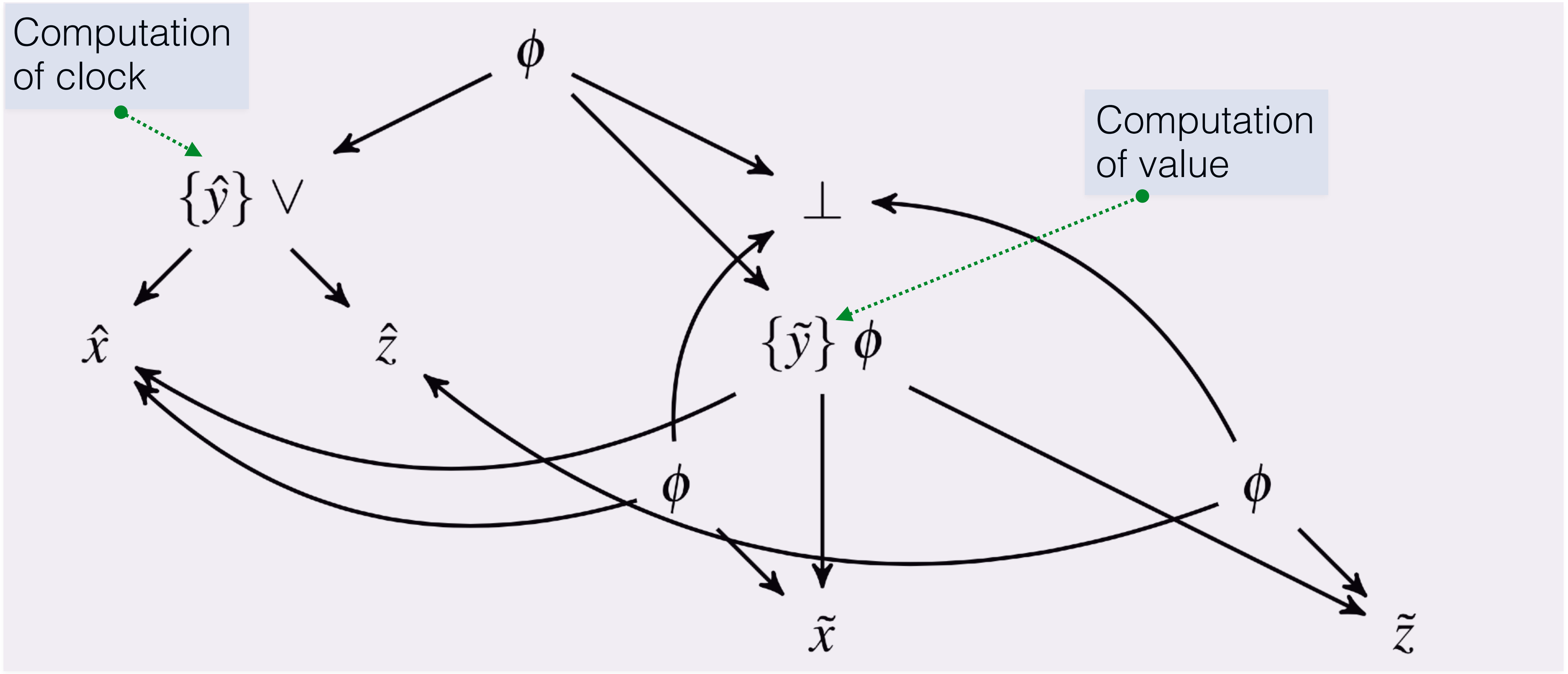
Synchronous data-flow value-graph (SDVG)

- Signal and clock computation is represented as a labeled directed graph
- Nodes are clocks, signals, variables, operators, or gated -node function
- Edges describe the computation relation between the nodes
- The computation of both Signal program and generated C code is represented by a shared graph

SDVG of Signal

- For each signal x, its computation is represented as $x = \phi(\hat{x}, \tilde{x}, \perp)$
- The computation of y in $y := x \texttt{ default } z$ is graphically represented by

$$y = \phi(\hat{y}, \phi(\hat{x}, \tilde{x}, \tilde{z}), \perp); \hat{y} \Leftrightarrow (\hat{x} \vee \hat{z})$$



SDVG of generated C

- The computation of the corresponding variable, denoted by x^c, is implemented as follows

```
if (C_x) {
    computation;
}
```

- This computation can be represented by the graph

$$x^c = \phi(C\_x, \widetilde{x^c}, \perp)$$

SDVG translation validation: Normalizing

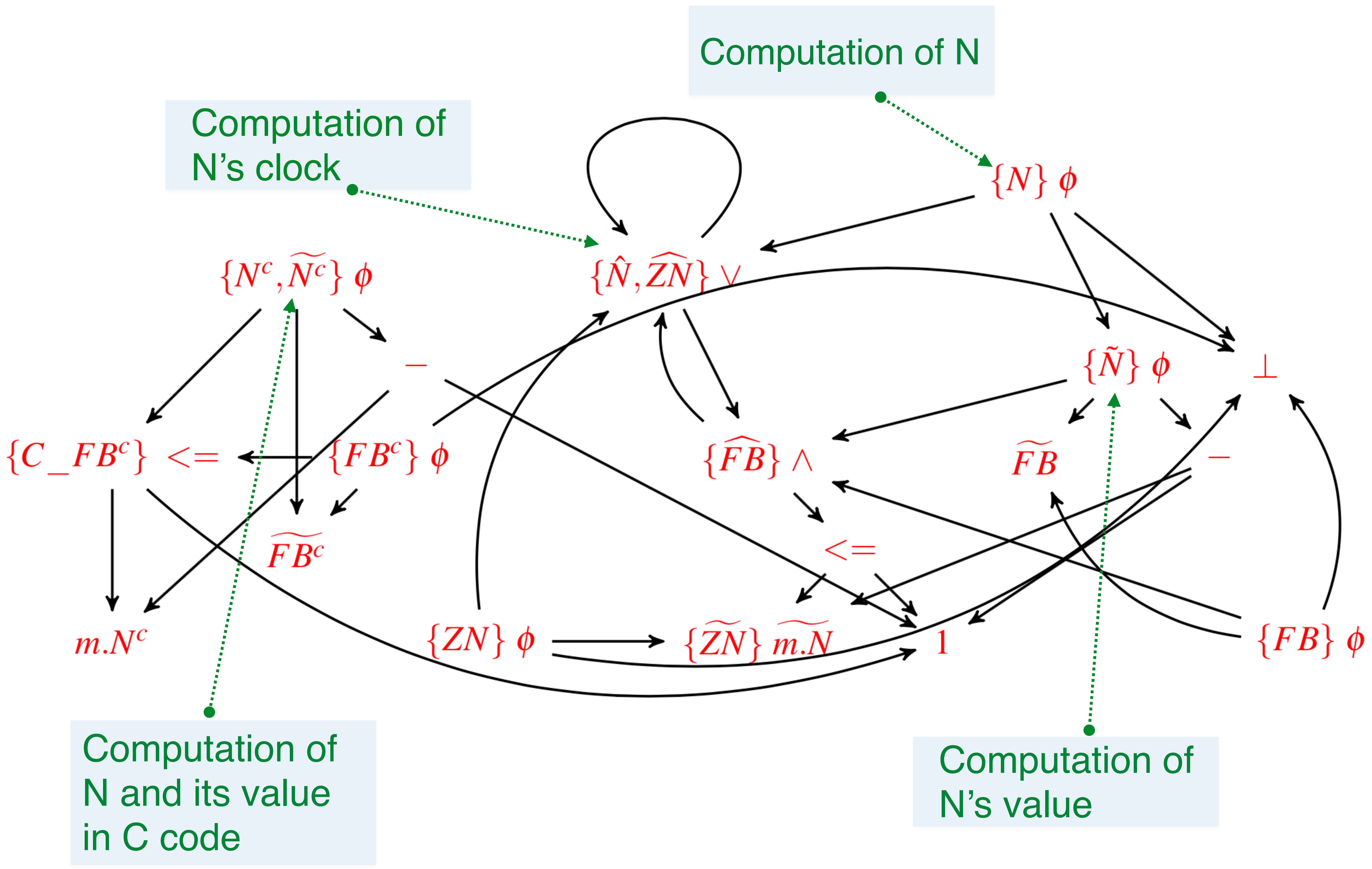
Objective

- Prove that for every output signal x and its corresponding variable x^c, they have the same value

Principle

- Define a set of rewrite rules
- Apply the rewrite rules to each graph node individually
- When there is no more rules can be applied to resulting graph, maximized the shared nodes
- Terminate when there exists no more sharing or rewrite rules can be applied

SDVG of DEC



Computation of N
Computation of N's clock
Computation of N and its value in C code
Computation of N's value
$\{N\}\ \phi$
$\{\hat{N}, \widehat{ZN}\}\ \vee$
$\{N^c, \widetilde{N^c}\}\ \phi$
$-$
$\{\tilde{N}\}\ \phi$
$\perp$
$\{C\_FB^c\}\ <=$
$\{FB^c\}\ \phi$
$\{\widehat{FB}\}\ \wedge$
$\widetilde{FB}$
$-$
$\widetilde{FB^c}$
$<=$
$m.N^c$
$\{ZN\}\ \phi$
$\{\widetilde{ZN}\}\ \widetilde{m.N}$
1
$\{FB\}\ \phi$

Normalize SDVG of DEC

There might have more than one normalization scenario

$$\wedge(x, \texttt{true}) \rightarrow x \tag{1}$$

$$\phi(\texttt{true}, x_1, x_2) \rightarrow x_1 \tag{2}$$

$$x^c \mapsto \phi(\texttt{true}, \widetilde{x^c}, \perp) \rightarrow x \mapsto \phi(\texttt{true}, \tilde{x}, \perp) \tag{3}$$

$$m.x^c \mapsto G_1 + \widetilde{m.x} \mapsto G_2 \rightarrow m.x^c, \widetilde{m.x} \mapsto G_1 \tag{4}$$

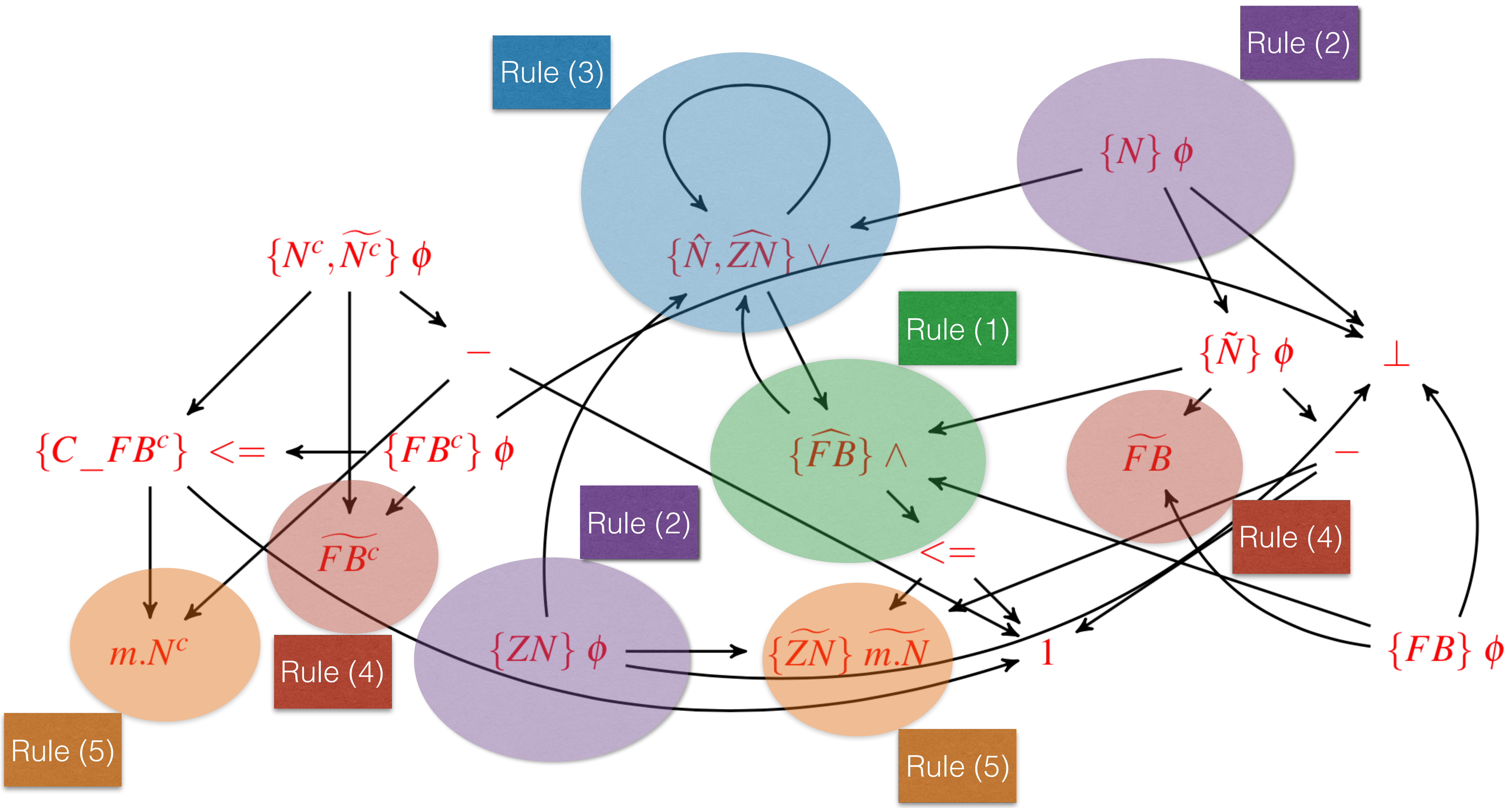
$$\widetilde{x^c} \mapsto G_1 + \tilde{x} \mapsto G_2 \rightarrow \widetilde{x^c}, \tilde{x} \mapsto G_1 \tag{5}$$

$$\begin{tikzcd}
\text{Rule (3)} \arrow[d] & \text{Rule (5)} \\
\text{Rule (1)} \arrow[d] & \text{Rule (4)} \arrow[u] \\
\text{Rule (2)} \arrow[r] & \text{Rule (2)} \arrow[u]
\end{tikzcd}$$

The set of applicable rewrite rules

A potential scenario

Normalize SDVG of DEC



Normalize SDVG of DEC: Final graph

Values of N and N^c represented by the same subgraph

$$\begin{tikzcd}
& \{N^c, \widetilde{N^c}, N, \tilde{N}\}\ \phi \arrow[dl] \arrow[d] \arrow[dr] & & \{FB^c, FB\}\ \phi \arrow[dlll] \arrow[dll] \arrow[d] \\
\{C\_FB^c, \widehat{FB}\}\ <= \arrow[d] \arrow[dr] & \{\widetilde{FB^c}\}\ \widetilde{FB} & - \arrow[dll] \arrow[dl] & \perp \\
\{\widetilde{m.N}, ZN, \widetilde{ZN}\}\ m.N^c & 1 & &
\end{tikzcd}$$

Detected bugs: Multiple constraints on a clock

```
// P.SIG
| x ^= when (y <= 9)
| x ^= when (y >= 1)
// P_BASIC_TRA.SIG
...
| CLK_x := when (y <=
    9)
| CLK := when (y >= 1)
| CLK_x ^= CLK
| CLK ^= XZX_24
...
// P_BOOL_TRA.SIG
...
| when Tick ^= C_z ^=
    C_CLK
| when C_z ^= x ^= z
| C_z := y <= 9
| C_CLK := y >= 1
...
```

Cause: The synchronization between CLK and XZX_24

- In P_BASIC_TRA, x might be absent when XZX_24 is absent, which is not the case in P and P_BOOL_TRA
- XZX_24 is introduced without declaration

Detection:

$$\Phi(\mathtt{P\_BOOL\_TRA}) \not\sqsubseteq_{clk} \Phi(\mathtt{P\_BASIC\_TRA})$$

Detected bugs: XOR operator

```
// P.SIG
| b3 := (true xor true)
      and b1
// P_BASIC_TRA.SIG
...
| CLK_b1 := ^b1
| CLK_b1 ^= b1 ^= b3
| b3 := b1
...
```

Cause: wrong implementation of XOR operator

- In P_BASIC_TRA, true xor true is true

Detection:

$$\Phi(\texttt{P\_BASIC\_TRA}) \not\sqsubseteq_{clk} \Phi(\texttt{P})$$

Detected bugs: Merge with constant signal

```
// P.SIG
| y := 1 default x
// C code
if (C_y)
{
   y = 1; else y = x;
   w_P_y(y);
}
```

Cause: wrong implementation of merge operator with constant signal

- In the generated C code, a syntax error y = 1; else y = x;

Detection: when constructing the SDVG graph

Conclusion

A method to formally verify the Signal compiler

- Adopts the translation validation
- Is light-weight, scalable, modular
- Separates the proof into three smaller and independent sub-proofs: clock semantic, data dependency, and value-equivalence preservations

Future work

- Fully implementation of the validator: benchmarks and integration into Polychrony toolset
- Extended the proof to use with the other code generation schemes (e.g., modular and distributed code generations)
- Use an SMT solver to reason on the rewrite rules in SDVG transformations

Publication

- *Translation validation for clock transformations in a synchronous compiler* - FASE - ETAPS 2015
- *Precise deadlock detection for polychronous data-flow specifications* - ESLsyn - DAC 2014
- *Formal verification of synchronous data-flow program transformations towards certified compilers* - FCS 2013
- *Formal verification of compiler transformations on polychronous equations* - IFM 2012
- *Formal verification of automatically generated C-code from polychronous data-flow equations* - HLDVT 2012

Thank you!