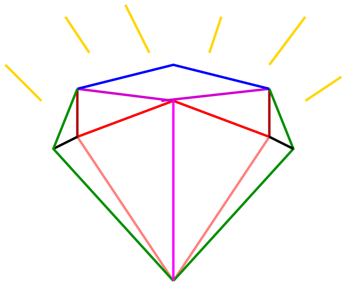


# Deduction is Forever

A. Pnueli

Weizmann Institute of Sciences
The John von Neumann Minerva Center for the Verification of Reactive Systems

Formal Methods'99, Toulouse
September 1999

Joint work with:

T. Arons and S. Ruah

# In the Friendly Competition

Between deductive verification and model checking, as methods for formal verification, Model Checking seems to be winning by points.

However, the race is not yet over.

Just as it was unavoidable, due to the growing complexity of circuits, that circuit manufacturers started to employ formal methods for verifying their designs, it is equally inevitable that more of the practicing verifiers will turn to deductive technologies, due to their significantly better scalability.

It is the responsibility of the community working on the development of deductive tools to make their tools more

- Automatic,
- Accessible (less forbidding), and
- User-friendly.

# Deduction vs. Exploration

and why should we care about the differences?

It is not that

- Only Deduction is symbolic.
- Only Exploration uses BDD's.
- Exploration is necessarily restricted to practically finite-state systems.

In our opinion, the main distinctions are that

- Deduction is based on induction while Exploration computes the set of reachable states.
- Deduction uses a more expressive language which includes quantifiers, leading to succinct specification of parameterized systems.
- Deduction requires user ingenuity and interaction.

# Preliminaries and Notations

A design to be verified is represented by a Synchronous Transition System $\mathcal{S} = \langle V, \Theta, \rho \rangle$ consisting of:

- $V$ — A set of system variables. A state $s$ is an interpretation of $V$. Let $\Sigma$ denote the set of all states.
- $\Theta$ — The initial condition. An assertion.
- $\rho(V, V')$ — A transition relation. An assertion referring to primed and unprimed copies of the system variables.

A computation of $\mathcal{S} = \langle V, \Theta, \rho \rangle$ is an infinite sequence of states:

$$\sigma: \quad s_0, s_1, \ldots,$$

satisfying $s_0 \models \Theta$ and $\rho(s_i[V], s_{i+1}[V]) = \mathrm{T}$, for every $i = 0, 1, \ldots$ .

# The Essence of Model Checking

Define the existential predecessor predicate transformer:

$$\rho \diamond \psi \quad = \quad \exists V' : \rho(V, V') \wedge \psi(V') \quad \sim_{CTL} \quad \mathbf{EX}\psi$$

$\rho \diamond \psi$ characterizes the states which have some $\rho$-successor satisfying $\psi$.

The immediate predecessor transformer can be iterated to yield the eventual predecessor transformer:

$$\rho^* \diamond \psi \quad \sim_{CTL} \quad \mathbf{EF}\psi \quad =$$
$$\psi \vee \rho \diamond \psi \vee \rho \diamond (\rho \diamond \psi) \vee \rho \diamond (\rho \diamond (\rho \diamond \psi)) \vee \cdots$$

$\rho^* \diamond \psi$ characterizes all states which have some $\rho^*$-successor satisfying $\psi$.

To model check $\mathcal{S} \models \square\, \psi$, check that

$$\Theta \ \wedge \ \rho^* \diamond \neg\psi \quad = \quad \mathrm{F}$$

# Deductive Verification

We may use the following rule to prove the invariance of assertion $p$:

Rule INV

$$\begin{array}{ll} \text{I1.} & \Theta \rightarrow \varphi \\ \text{I2.} & \varphi \wedge \rho \rightarrow \varphi' \\ \text{I3.} & \varphi \rightarrow p \\ \hline & \Box\, p \end{array}$$

An assertion $\varphi$ satisfying I2 and I3 is called inductive.

Note that if the system is finite state and $\varphi$ is given, then the three premises can be verified by computing their BDD's and checking that they equal $1$. In such cases, no theorem prover is necessary.

This is usually much cheaper than computing $\rho^* \diamond \neg p$. Thus, deductive verification can give us more for the higher price of coming up with the inductive assertion $\varphi$.

## A Proof Strategy:

Find an inductive assertion $\varphi$ strengthening $p$.

# Sometimes We Can Take $p = \varphi$

Consider the following program for ensuring mutual exclusion by the use of semaphores.

$$y: \quad \textbf{natural where } y = 1$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1 : & \textbf{Non-Critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y \end{array}\right] \end{array}\right] \quad \| \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} m_1 : & \textbf{Non-Critical} \\ m_2 : & \textbf{request } y \\ m_3 : & \textbf{Critical} \\ m_4 : & \textbf{release } y \end{array}\right] \end{array}\right]$$

The semaphore instructions **request** $y$ and **release** $y$ respectively stand for

$$\langle \textbf{await } y > 0 \,;\ y := 0 \rangle \quad \text{and} \quad y := 1.$$

Wishing to verify the invariance of the assertion $y = 0 \ \lor \ y = 1$, we can take

$$\varphi \quad = \quad p \quad = \quad (y = 0 \ \lor \ y = 1).$$

# Sometimes We Cannot

Wishing to verify the invariance of $\neg(at\_\ell_3 \wedge at\_m_3)$, guaranteeing mutual exclusion for the same program,

$$y: \quad \textbf{natural where } y = 1$$

$$\left[\begin{array}{l}\ell_0 : \textbf{loop forever do} \\ \left[\begin{array}{ll}\ell_1 : & \textbf{Non-Critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y\end{array}\right]\end{array}\right] \quad \| \quad \left[\begin{array}{l}m_0 : \textbf{loop forever do} \\ \left[\begin{array}{ll}m_1 : & \textbf{Non-Critical} \\ m_2 : & \textbf{request } y \\ m_3 : & \textbf{Critical} \\ m_4 : & \textbf{release } y\end{array}\right]\end{array}\right]$$

We must strengthen $p : \neg(at\_\ell_3 \wedge at\_m_3)$ into

$$\varphi: \quad \neg(at\_\ell_3 \wedge at\_m_3) \wedge ((at\_\ell_{3,4} \vee at\_m_{3,4}) \rightarrow y = 0)$$

# A Strengthening Heuristic

Define the universal predecessor predicate transformer:

$$\rho \,\square\, \psi \quad = \quad \forall V' : \rho(V, V') \rightarrow \psi(V') \quad \sim_{CTL} \quad \mathbf{AX}\psi$$

$\rho \,\square\, \psi$ characterizes states $s$, such that all the $\rho$-successors of $s$ satisfy $\psi$.

Note that

I3. $\varphi \wedge \rho \rightarrow \varphi'$ is valid iff $\varphi \rightarrow (\rho \,\square\, \varphi)$ is,

and that

If $\varphi$ is an invariant of $\mathcal{S}$ then so is $\rho \,\square\, \varphi$.

## A (Top-Down) Strengthening Heuristic:

If $\varphi$ violates I3: $\varphi \wedge \rho \rightarrow \varphi'$ then conjunct it with $\rho \,\square\, \varphi$.

# Iterated Strengthening

The immediate universal predecessor transformer $\rho \,\square\, \psi$ can be iterated to yield the eventual universal predecessor transformer:

$$\rho^* \square\, \psi \quad \sim_{CTL} \quad \mathbf{AF}\psi \quad =$$
$$\psi \wedge \rho \,\square\, \psi \wedge \rho \,\square\, (\rho \,\square\, \psi) \wedge \rho \,\square\, (\rho \,\square\, (\rho \,\square\, \psi)) \wedge \cdots$$

$\rho^* \square\, \psi$ characterizes the states all of whose $\rho^*$ successors satisfy $\psi$.

**The Iterated Strengthening Heuristic:**

Compute $\rho^* \square\, \psi$ until it terminates. When (and if) it does, this is the weakest inductive assertion strengthening $p$.

# Duality between Exploration and Deduction

The processes of backwards reachability analysis in model checking and that of iterated strengthening in deductive verification are dual.

$$\text{Verify} \qquad S \models \Box\, p$$

**Model Checking**

$\psi_0 := \neg p$
**for** $i := 0, 1, \ldots$ **do**
$\quad \psi_{i+1} := \psi_i \vee (\rho \diamond \psi_i)$
**until** $\psi_i = \psi_{i+1}$
**Check** $\Theta \wedge \psi_i = \mathit{false}$

**Deductive**

$\varphi_0 := p$
**for** $i := 0, 1, \ldots$ **do**
$\quad \varphi_{i+1} := \varphi_i \wedge (\rho \,\Box\, \varphi_i)$
**until** $\varphi_i = \varphi_{i+1}$
**Check** $\Theta \rightarrow \varphi_i$

Duality is established by the invariant

$$\psi_i = \neg\varphi_i$$

**Conclusion:** If one insists on making (finite-state) deduction fully automatic, it reduces to model-checking.

# Deduction Can Verify Bigger Finite-State Systems

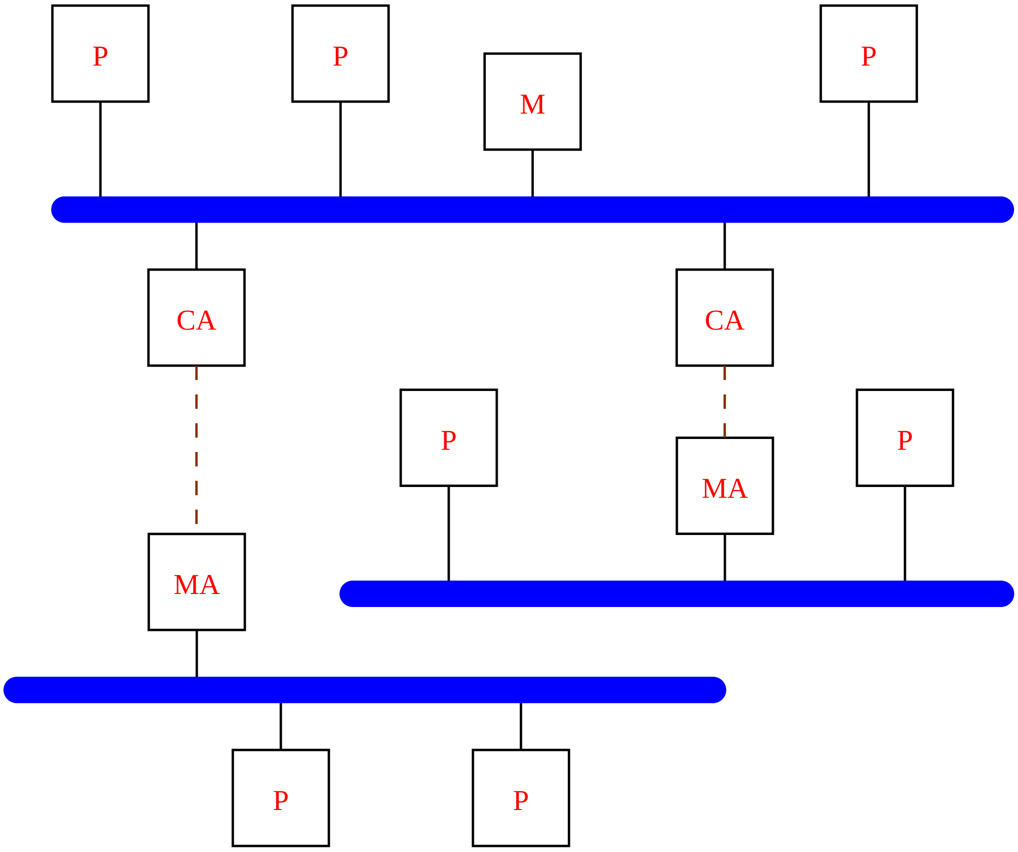
Using BDD's, it is cheaper to check

$$\varphi \ \wedge \ \rho \quad \rightarrow \quad \varphi',$$

than to check

$$\Theta \diamond \rho^* \ \rightarrow \ \varphi \qquad \text{or} \qquad \rho^* \diamond (\neg\varphi) \ \rightarrow \ \neg\Theta$$

# The FutureBus+ Case Study

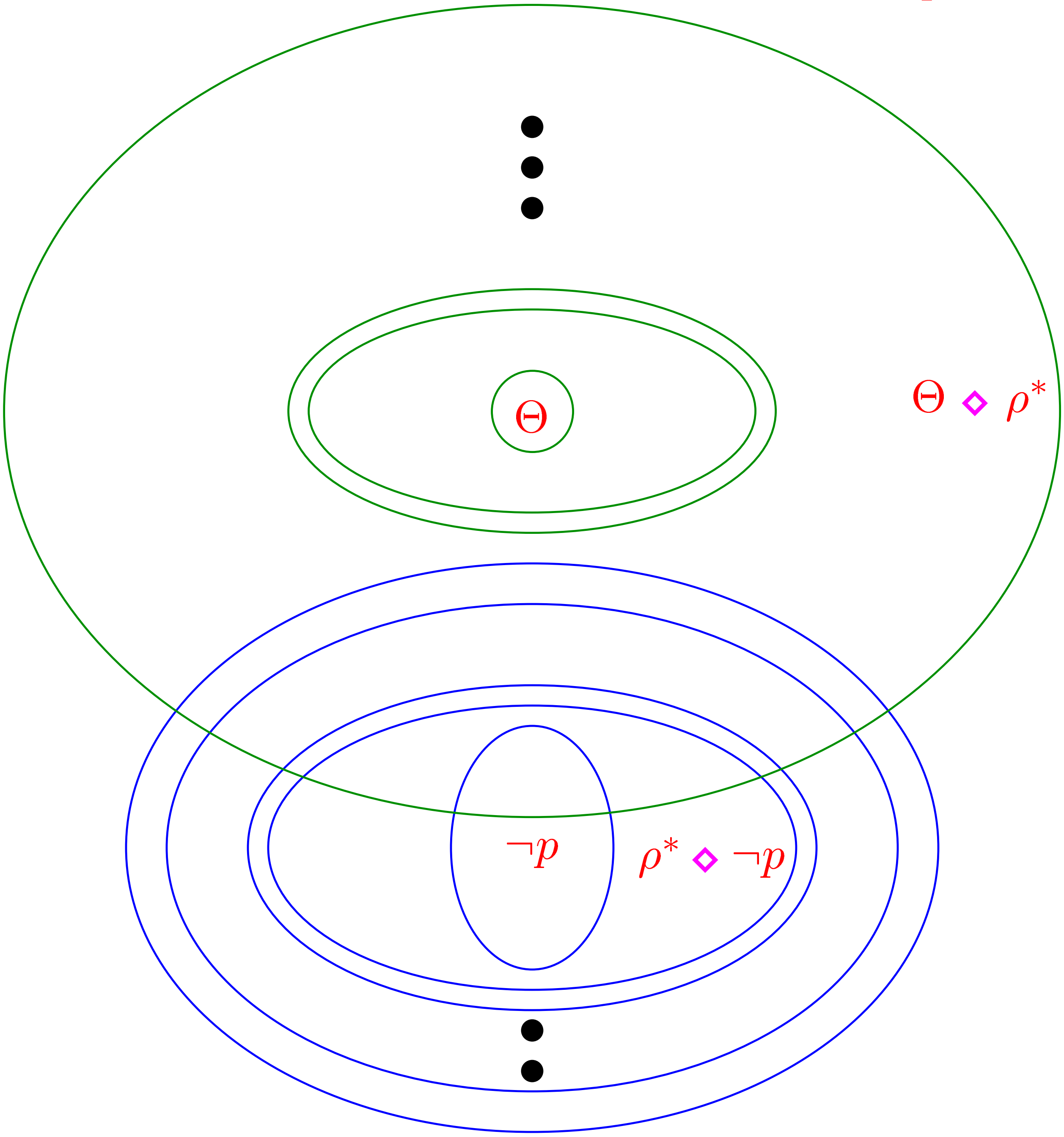



An array of buses interconnected by asynchronous bus bridges.

In [CGHJLMN93] it was formulated and specified, symbolically model checked for many configurations, and declared verified.

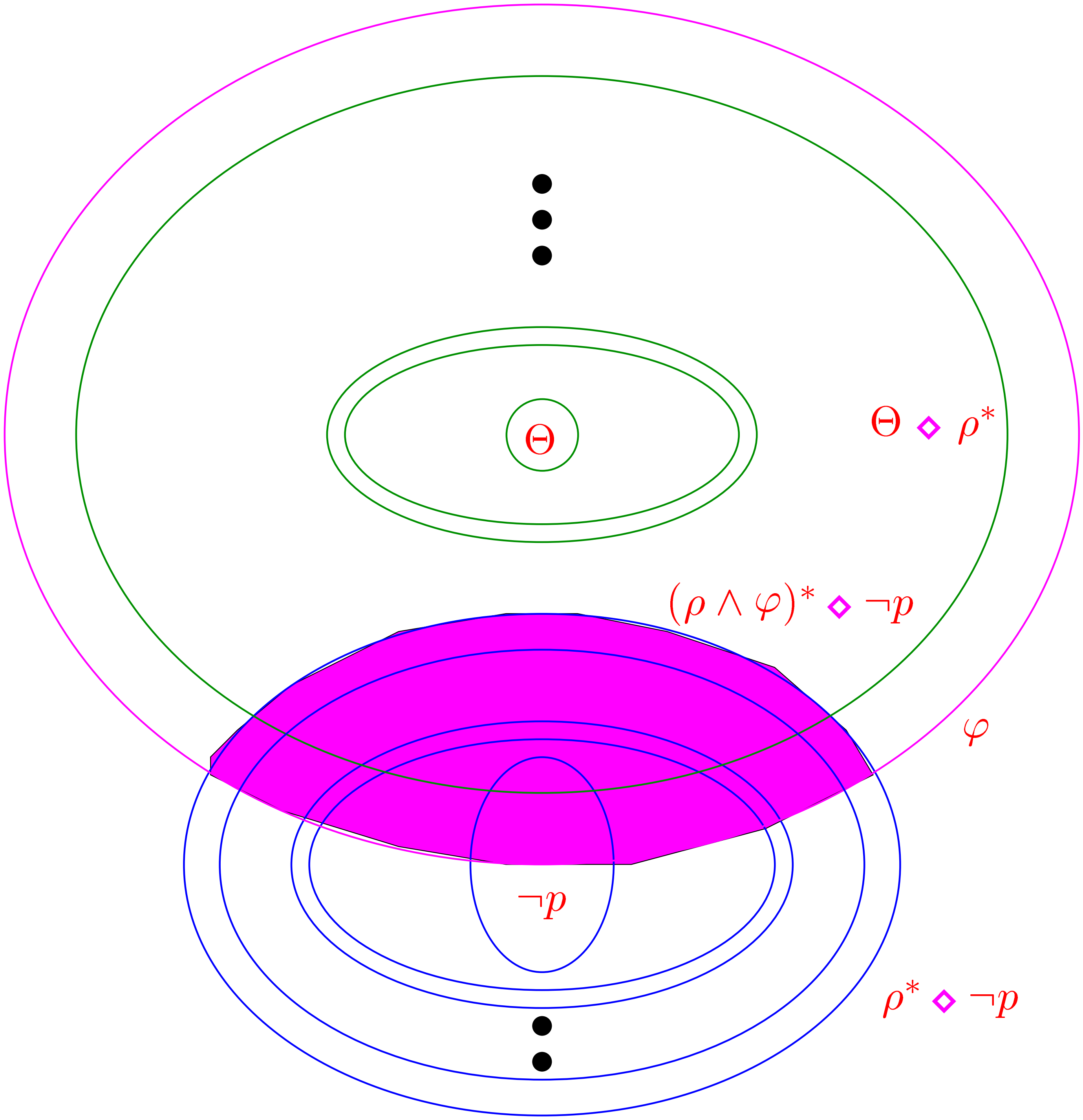
In [PS96] we applied deductive techniques to larger configurations and uncovered a remaining bug.

# Even a Little Deduction Goes a Long Way




Exploration in both directions (forward or backward) traverse a multitude of irrelevant states.

# With a Little Help from an Invariant




Backwards search more focused.

# In Search of a Wide-Base Usable Deductive Tool

The previous examples indicated the great raw potential possessed by the deductive technology which can often be impressively utilized by experts. We also pointed out that user ingenuity is an essential ingredient in its application.

The main challenge is to propose a working deductive tool and associated methodology which can be used by the "masses".

# Necessary Conditions for the Ultimate Tool

In our opinion, such an ultimate tool should satisfy the following necessary conditions:

- Focus on a specific application area, e.g. verification of micro-processor micro-architecture.
- All verification conditions should be algorithmically resolvable. No need for interactive theorem proving.
- Invalid verification conditions must yield usable and focused counter-examples, clearly pointing the way to a missing invariant.
- The need for ingenuity and cleverness should be factored out into:
  - Familiarity with the verified design.
  - Bottom-up invariants, which can be algorithmically constructed [BBM95], [BLS96].
  - Generic proof methods developed by experts.

# Our Experience with Application-Domain Focused Deductive Verification

The previous observations are based on our experience in a continuous case study since 1997, verifying various versions of multi-scalar (out-of-order micro-architecture designs. The versions we have considered are:

- Tomasulo algorithm – one of the earliest out-of-order textbook designs.
- Tomasulo+in-order retirement – the system maintains a coherent register file image.
- The above + speculation – reaching a branch point, execution chooses one of the branches and continue. In case the branch prediction proved wrong, some of the computation must be undone.
- The above + exceptions – allowing internal (e.g. overflow) and external exceptions which necessitates restoring to a stable and coherent machine state.

While verifying (deductively and by other means) these series of case studies, we attempted to formulate a steady methodology and examine the degree of re-usability we could achieve.

# Verification of Out-of-Order Designs

Standard conceptual model for the execution of machine-language programs is sequential – An instruction executes only when the previous instruction is completed.

**Modern processors** execute many instructions in parallel, in an order often different from their program order – out-of-order execution.

**The Problem:** Verify that an out-of-order design is a correct implementation of the sequential model.

To handle this problem we introduce a system SEQ, representing the sequential reference model and a system OOOD, representing the out-of-order design to be verified.

Typically, SEQ executes every instruction as it arrives, while OOOD accumulates in its belly a set of instructions, analyzes their dependency, and executes them, as much in parallel as possible, according to the availability of functional units.

# The Proof Method

In principle, the problem is that of refinement, having to establish that $\text{OOOD} \sqsubseteq \text{SEQ}$

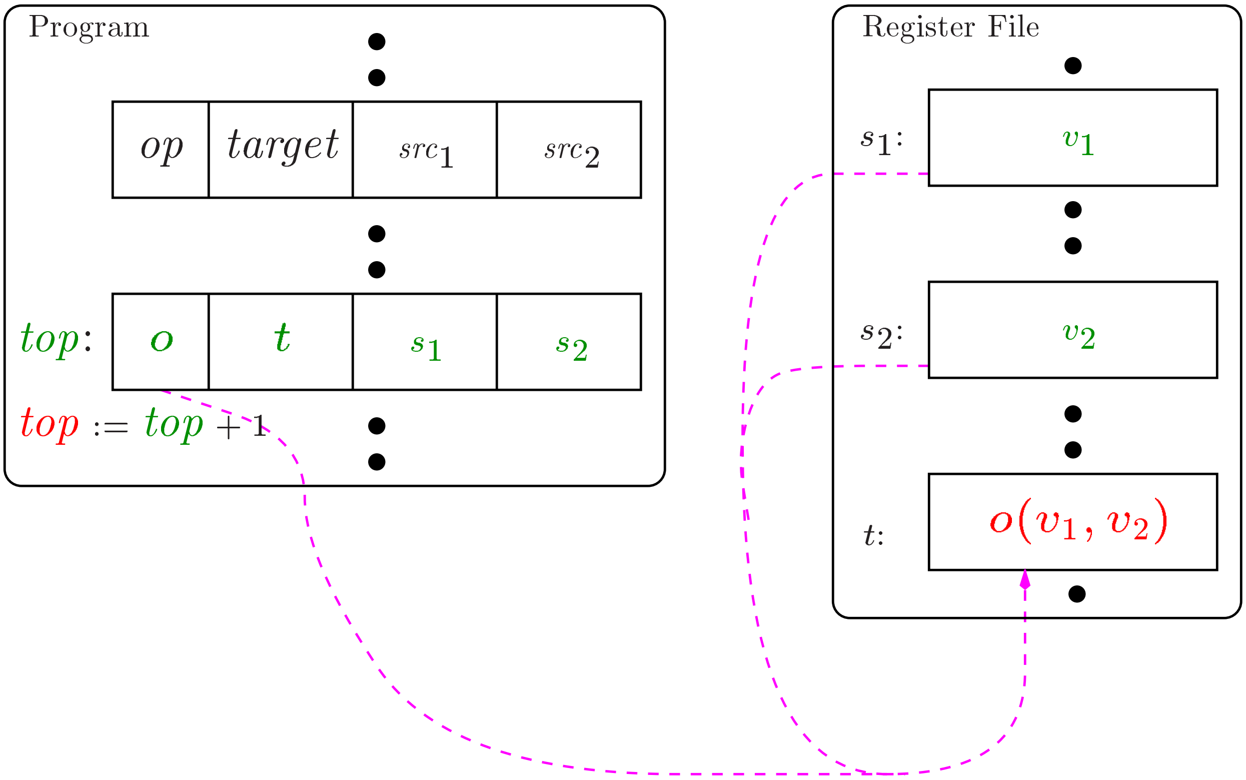
That is, every observation of OOOD corresponds to some observation of SEQ. This is usually verified by identifying an abstraction mapping, establishing the correspondence between OOOD-states and SEQ-states, and then showing that, under an appropriate OOOD-invariant, every OOOD-step corresponds to a SEQ-step.

Currently employed proof methods are:

- Refinement with the abstraction mapping proposed in [BD94]. The SEQ state corresponding to an OOOD state is obtained by flushing all pending instructions.
- Refinement with an abstraction mapping based on predicted values [AP99].
- A new approach based on the method of convenient computations [KG99]. Given a computation of OOOD, one shows that it can always be reduced to a computation of SEQ, by permuting pairs of adjacent independent operations.

Each of these methods deserves a tool of its own, reflecting the highest level of user ingenuity.

# Example Reference Model: System seq

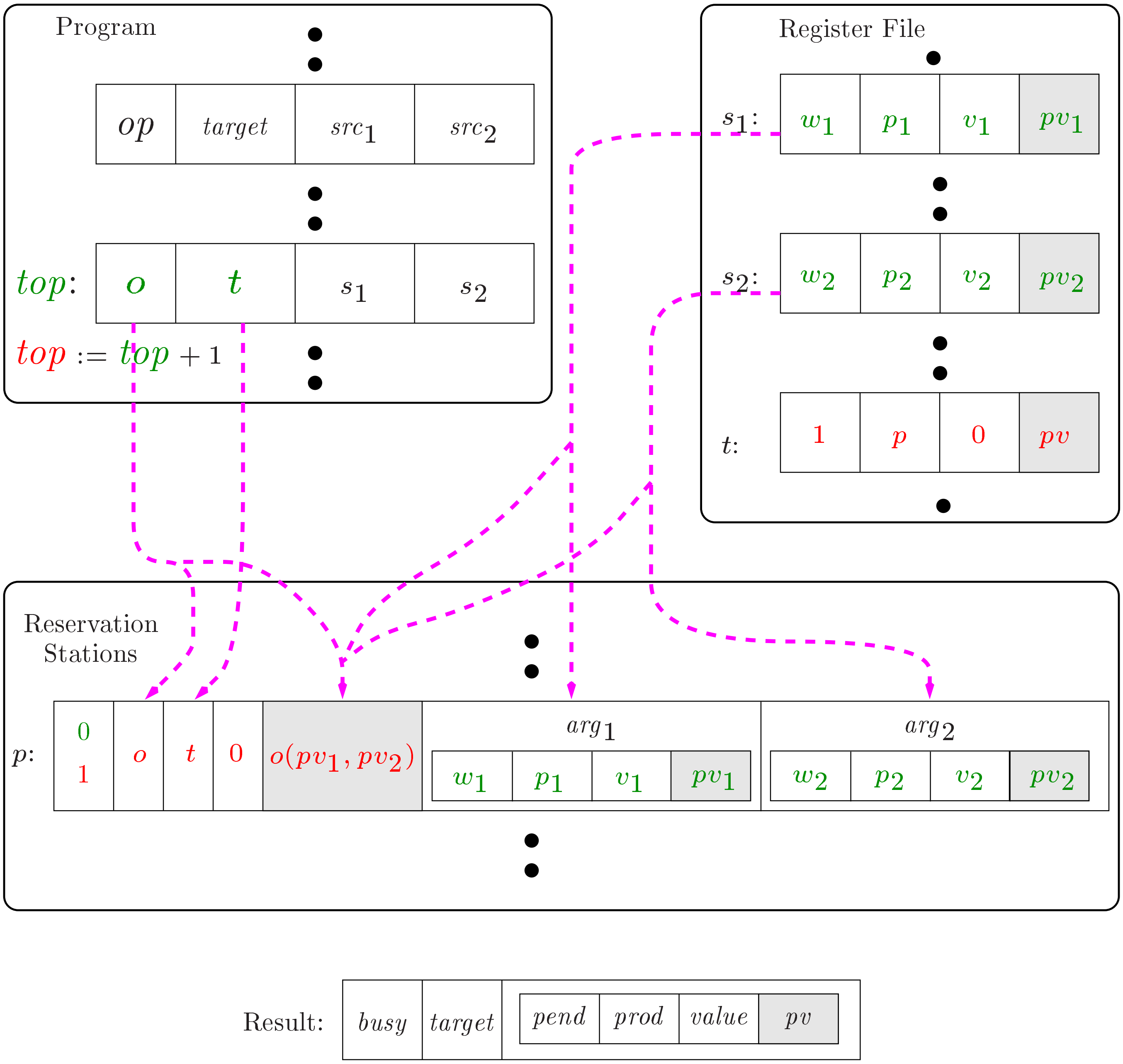


# Example Design to be Verified: System Tomasulo

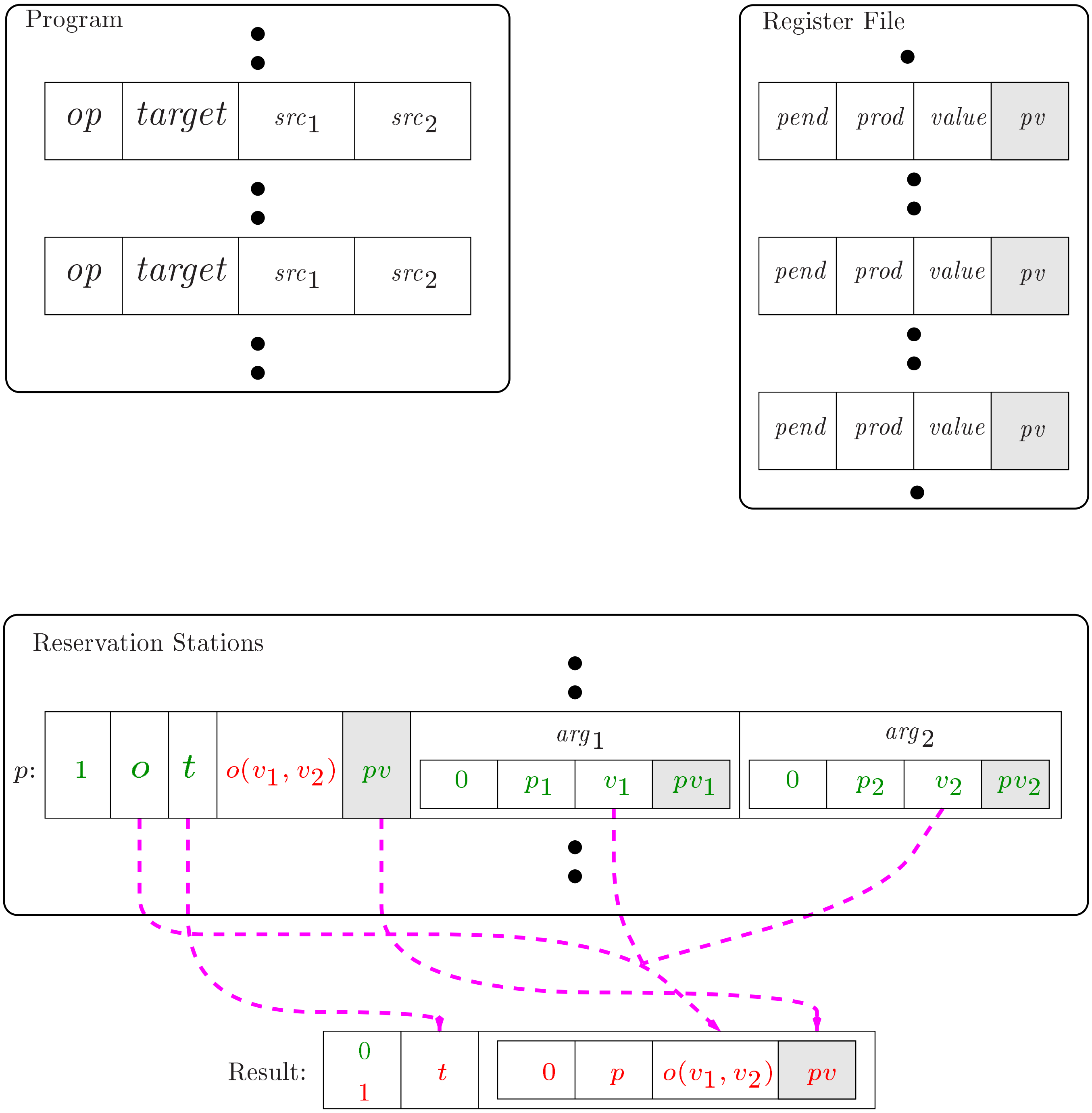
**System** TOMASULO **is**

$$\text{DISPATCH} \quad \| \quad \text{COMPLETE} \quad \| \quad \text{WRITE\_BACK}$$

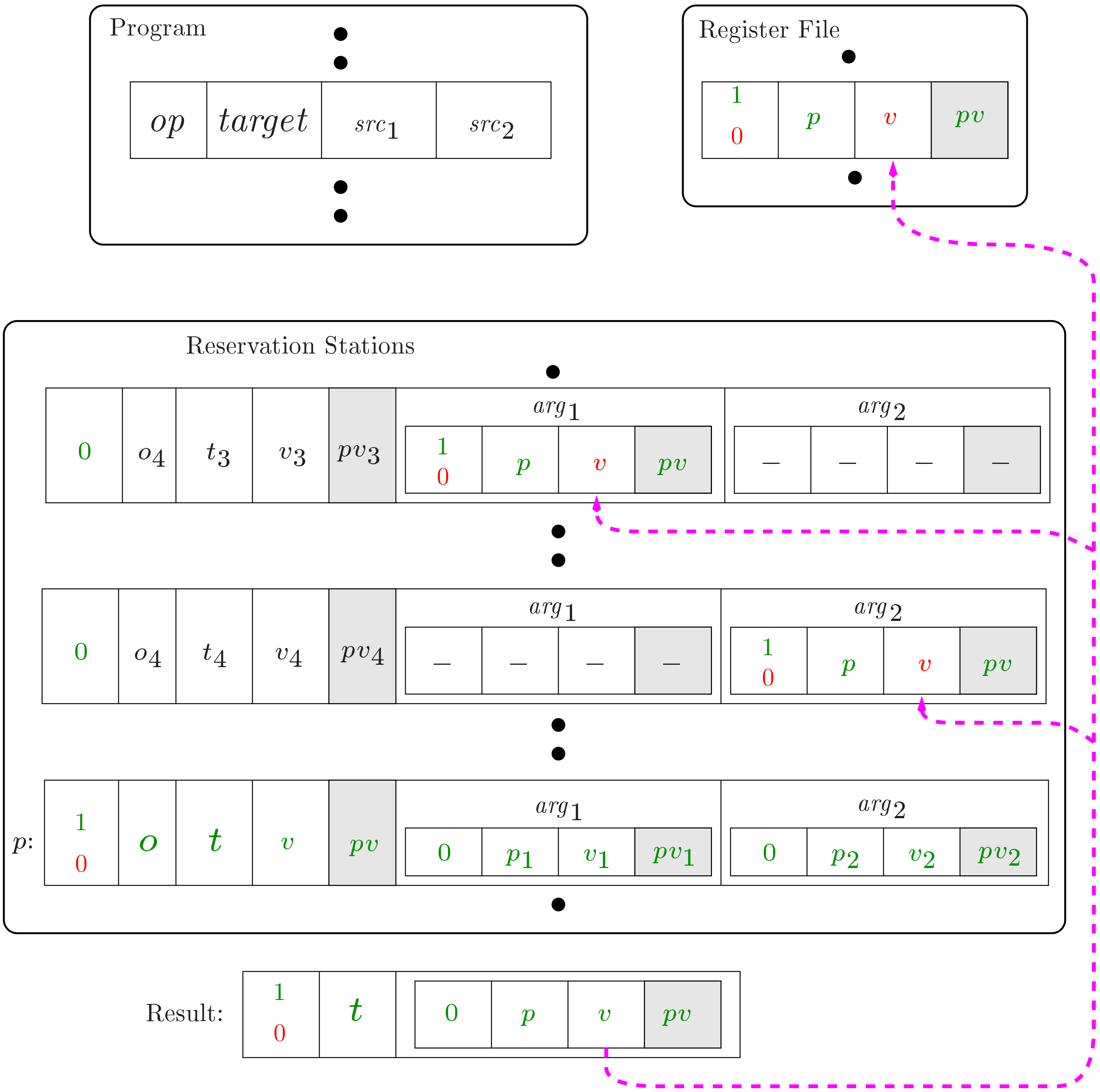
# Module TOMASULO.dispatch



# Module TOMASULO.complete



# Module TOMASULO.write_back



# Predicted values

- Every value field in the system is paired with an auxiliary predicted value field.
- Predicted values are calculated using the predicted values of the operands when instructions are dispatched.
- When a value becomes available it equals the predicted value field.

# The Abstraction Mapping

The abstraction mapping is given by

$$\mathcal{R}: \quad \left(\begin{array}{rcll} top_A & = & top_C & \wedge \\ \forall r : [0..R] : \ reg[r] & = & RF[r].pv & \end{array}\right)$$

The mapping states that, at all points in the joint execution of TOM and SEQ, the concrete and the abstract program pointers agree, and that the value of $reg[r]$ equals the predicted value of $RF[r]$.

# The Proof Process

Good old incremental strengthening.

To verify $\Box\, p$, perform the following:

**if** $\neg(\Theta \rightarrow p)$ **then exit** $--$ $p$ is not an invariant
$\varphi := p$
**while** $\neg(\varphi \wedge \rho \rightarrow \varphi')$ **do**
  **if** the state violating $\varphi \wedge \rho \rightarrow \varphi'$ is reachable
  **then exit** $--$ $p$ is not an invariant
  $\varphi := \varphi \wedge \psi$  where $\psi$ is an assertion excluding the counter-example

**Exit** $--$ $p$ is an invariant

This identifies an additional necessary condition:

A linkage with a model checker, providing a quick check whether a violating pre-state is reachable.

# Example: Application to Tomasulo

The main invariant in the verification of Tomasulo is:

$$\forall r : [0..R] : \ \neg RF.pend \quad \rightarrow \quad (RF.value = RF.pv),$$

stating that when a register pending bit is off, then the actual and the predicted value should agree.

Trying to verify this, we find that it is violated by the write-back transition, with a counter-example pre-state in which

$$Res.pend = 0, \quad Res.value = 0, \quad Res.pv = 1$$

To correct this, we strengthen the above invariant by

$$\forall r : [0..R] : \ \neg Res.pend \quad \rightarrow \quad (Res.value = Res.pv)$$

# Algorithmic Resolution of the Verification Conditions

is based on the following observations:

- In the verification of micro-architecture designs (excluding those implementing the arithmetical circuits), we can replace all operations by uninterpreted functions.

- Furthermore, the data operations can be represented by the following single concrete data domain

$$\mathcal{B}_4: \quad \langle \{0,1\};\ \beta_0, \beta_1, \beta_2, \beta_3 : \{0,1\} \times \{0,1\} \mapsto \{0,1\} \rangle$$

$$\text{where}$$

$$\begin{array}{rclcrcl} \beta_0(x,y) & = & 0 & \qquad & \beta_1(x,y) & = & x \\ \beta_2(x,y) & = & y & & \beta_3(x,y) & = & 1 \end{array}$$

- Most arising verification conditions are of the AE form, i.e. $\forall x\, \exists\, y \varphi$, which lead to an effective decision procedure by a small finite instantiation.

# In Summary

Deductive verification is preordained to eventually take its rightful place in the formal verification of circuit designs, alongside model checking and other methods.

We can help it get there sooner by customizing our deductive tools and making them more accessible and widely usable.