Verifying Data-Insensitive Circuits

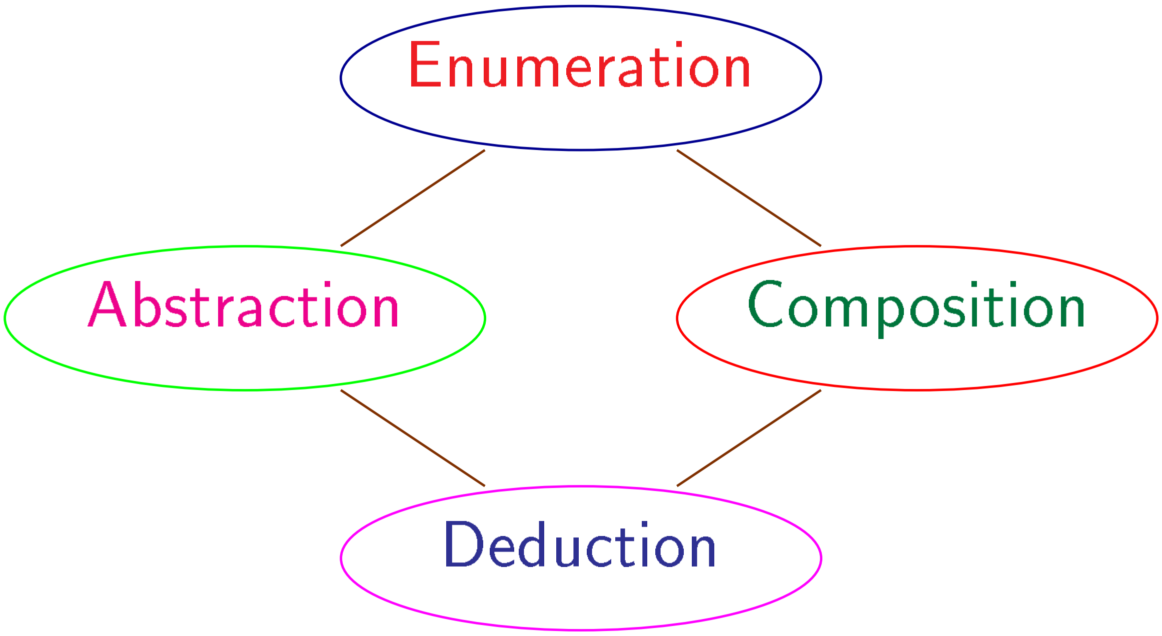
A. Pnueli
Weizmann Institute of Sciences

FMCAD'98, Palo Alto Nov. 6, 1998

Joint work with:

T. Arons, W. Damm, S. Ruah, O. Shtrichman

The Four Cornerstones of Formal Verification



- Enumeration (model checking): based on explicit or implicit exhaustive enumeration of all reachable states. Fully automatic but limited to finite-state (and not too big) systems.

- Deduction: based on theorem-proving methods. Applicable to infinite-state systems, but requires user interaction and creativity to provide auxiliary constructs.

- Abstraction and Composition are used to carve a big design into smaller and tractable pieces and to help glue together different methods.

Efforts for Bridging the Gap Between Deduction and Enumeration

- Making deduction more automatic:
  - Automatic construction of invariants and abstraction mappings.
  - Development of powerful decision procedures for automatically checking the verification conditions.

- Making (symbolic) model checking more expressive:
  - Using regular expressions, S1S, or S2S as assertional languages for symbolic model checking.
  - Extending the representation to cover symbolic expressions with uninterpreted functions.

Splitting the Territories

Until the wolf shall lie with the lamb in harmony, the common wisdom is:

- Pure control components, such as state machines, are more amenable to model checking techniques.
- Data- and arithmetical-intensive components are more amenable to deductive techniques.

What about data-insensitive components managing controlled manipulation of data but not caring too much about the nature or properties of the data?

This class of data-insensitive circuits contains many top-level architectural views of central components in the design of a modern micro-processor such as pipelines and out-of-order execution units.

Separation Between Control and Data

is important for good design but even more so for Formal Verification.

Control yields readily to finite-state model checking, while data is often the reason for state explosion.

It has been realized quite early that, for verification at the architectural level such as out-of-order designs (OOOD), it is sufficient to treat data as uninterpreted functions applied to an unspecified domain [BD94], [JDB95], [HIKB96].

In fact, this approach establishes a stronger property:

> The design is correct for every data domain and not only for the specific domain for which it was designed, such as floating point numbers.

What is actually established is

> Whenever the design applies an operation f to particular arguments then so can the reference machine (used as a specification), independently of the particular characteristics of f.

A more demanding requirement — easier to verify.

Uninterpreted Functions

are the default case for deductive verification. The problem there is how to interpret them in the appropriate way. They are effectively and efficiently handled by theorem provers such as PVS.

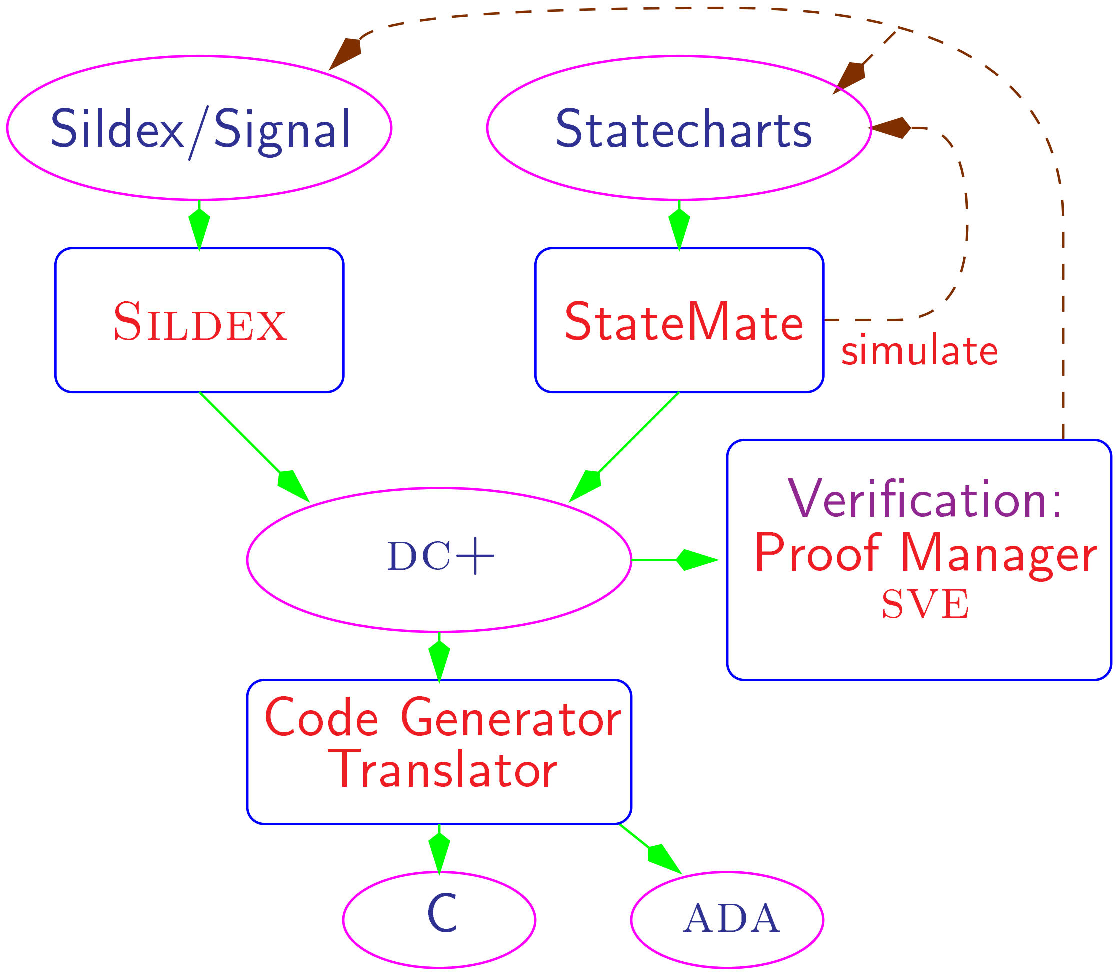
Have been recently rediscovered by the model checking community. The question is how to integrate them within automatic verification processes such as model checking.

Particularly useful in a verification problem involving a comparison (refinement, abstraction) between an implementation design and a reference model, where we wish to guarantee that every operation performed by the implementation mimics a similar operation executed by the reference model.

I. The Static (Combinational) Case: Translation Validation

The *Sacres* Project

Developing a methodology and a tool suit supporting the development of safety critical real time embedded systems.



A Basic Premise: All validation and verification activities take place at the levels of DC$^+$ and above. Once the design is believed correct, the target code (C or ADA) is generated automatically.

Translator Validation is Indispensable

According to standards and regulations, every component in the development of safety critical systems must be certified. In particular, every compiler or code generator must be validated.

Therefore, in order for the *Sacres* methodology to be acceptable for safety critical systems developments, it is essential that the $\text{DC}^{+}\rightarrow\text{C}$ translator be validated.

Translator vs. Translation Validation

Rather than verify the translator itself (using compiler verification techniques), we chose to verify the results of each run of the translator.

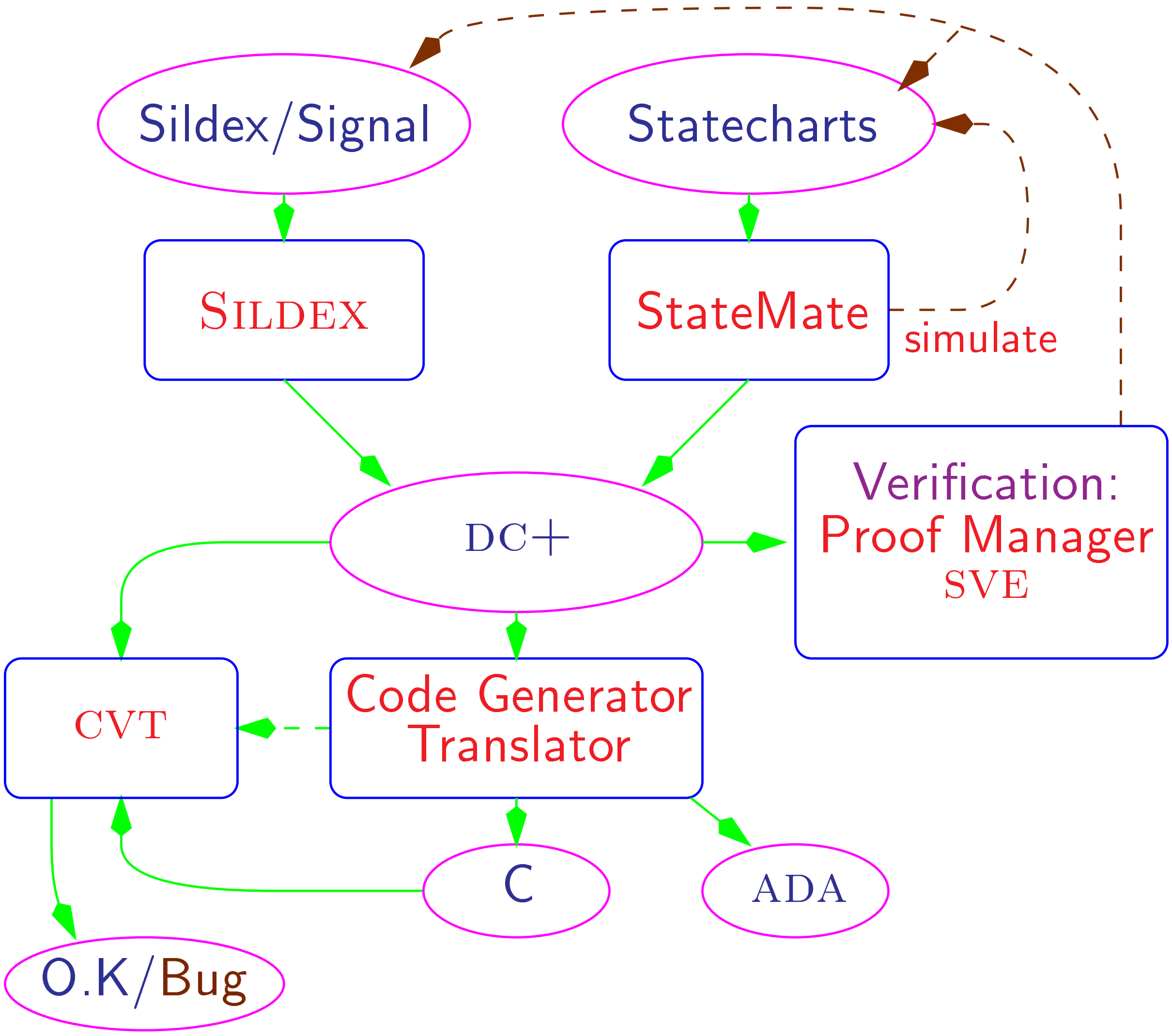
Constructed the CVT tool, which can be invoked after every run of the code generator, translating a DC^{+} specification D_1 into a C program C_1. Applied to (D_1, C_1), CVT verifies automatically that C_1 is a correct implementation of D_1 or identifies a bug in the translator.

The advantage of the translation validation approach are:

- Evolution of the translator is not frozen after verification.
- Significantly cheaper than translator verification.
- Completely oblivious to the assumptions intrinsic to the translator. Provides independent cross check.

The only drawback: constant run-time additional cost, can also be justified.

Placement of the CVT Tool



The Principle Underlying Proofs of Refinement, using Uninterpreted Functions

Every concrete application of a function f to arguments $t_1, \ldots, t_n$ corresponds to an abstract application of the same function to equal arguments.

As a running example, assume the the compiler translated $z := (x_1 + y_1) * (x_2 + y_2)$ into $u_1 := x_1 + y_1;\ u_2 := x_2 + y_2;\ z := u_1 * u_2$. The corresponding verification condition is

$$u_1 = x_1 + y_1 \ \land \ u_2 = x_2 + y_2 \ \land \ z = u_1 u_2 \quad \rightarrow \quad z = (x_1 + y_1)(x_2 + y_2)$$

The Decision Procedure: 1. Abstract into Uninterpreted Functions

Starting with the implication $\rho_C \rightarrow \rho_S$, whose validity we wish to verify,

1. Abstract into uninterpreted functions: replace each concrete arithmetical operation by a fresh function symbol.

Applied to

$$u_1 = x_1 + y_1 \;\wedge\; u_2 = x_2 + y_2 \;\wedge\; z = u_1 u_2 \quad \rightarrow \quad z = (x_1 + y_1)(x_2 + y_2),$$

where $+$ and $*$ are replaced by F and G, this yields

$$u_1 = F(x_1, y_1) \;\wedge\; u_2 = F(x_2, y_2) \;\wedge\; z = G(u_1, u_2) \quad \rightarrow \quad z = G(F(x_1, y_1), F(x_2, y_2))$$

Note that this transformation strengthens the statement to be proved. We validate the translation not only for the particular operations $+$ and $*$ but for all functions F and G.

The Decision Procedure:
2. Eliminate All Function Symbols

Next, we eliminate all function symbols, using the Ackerman's transformation.

2. Replace each application of a function by a fresh variable and add conjuncts ensuring the functionality of these applications.

Applied to

$$u_1 = F(x_1, y_1) \;\wedge\; u_2 = F(x_2, y_2) \;\wedge\; z = G(u_1, u_2) \quad \to \quad z = G(F(x_1, y_1), F(x_2, y_2))$$

where $F(x_1, y_1)$, $F(x_2, y_2)$, $G(u_1, u_2)$, $G(f_1, f_2)$ are respectively replaced by variables f_1, f_2, g_1, and g_2, this yields

$$\left(\begin{array}{ll} & (x_1 = x_2 \wedge y_1 = y_2 \to f_1 = f_2) \\ \wedge & (u_1 = f_1 \wedge u_2 = f_2 \to g_1 = g_2) \\ \wedge & u_1 = f_1 \;\wedge\; u_2 = f_2 \;\wedge\; z = g_1 \end{array}\right) \quad \to \quad z = g_2$$

The Decision Procedure: 3. Assign an Adequate Domain and Submit to a BDD Checker

A domain assignment assigns a domain to each variable of a formula. It is called finite if all assigned domains are finite.

A domain assignment $\mathcal{D}$ is adequate for an equality formula φ if, for every consistent partition of the atomic formulas of φ into sets A and B, there exists a $\mathcal{D}$-valuation of the φ-variables which validates all formulas in A and falsifies all formulas in B.

If $\mathcal{D}$ is adequate for φ, then

φ is valid (satisfiable) iff φ is valid (satisfiable) over $\mathcal{D}$.

For example, the assignment which assigns the domain $\mathcal{D}_n : [1..n]$ to all variables, is adequate for any equality formula with n variables.

3. Determine an adequate finite domain assignment $\mathcal{D}$ and submit to a BDD checker.

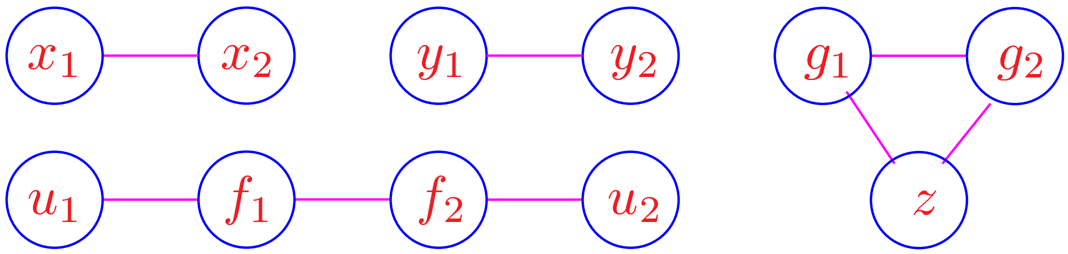
The Key to Our Success: Determination of Small Domains

Consider our running example formula:

$$\left(\begin{array}{ll} & (x_1 = x_2 \wedge y_1 = y_2 \rightarrow f_1 = f_2) \\ \wedge & (u_1 = f_1 \wedge u_2 = f_2 \rightarrow g_1 = g_2) \\ \wedge & u_1 = f_1 \ \wedge \ u_2 = f_2 \ \wedge \ z = g_1 \end{array}\right) \quad \rightarrow \quad z = g_2$$

Since it has 11 variables, we can always use $\mathcal{D}_{11} : [1..11]$, requiring 4 bits for each variable.

However, we can do much better. Drawing the comparison graph,



the following assignment is adequate (and optimal – requiring 1 bit per variable):

$$\begin{array}{llllll} x_1,\ x_2,\ y_1,\ y_2,\ u_1,\ f_1,\ f_2,\ u_2 : & & & & & \{0,1\} \\ g_1 : & \{0,1\} & g_2 : & \{0,2\} & z : & \{0,3\} \end{array}$$

Improving the domain assignment algorithm is still under intensive research.

II. The Dynamic (Sequential) Case: Out-of-Order Execution

Typical Representative: 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.

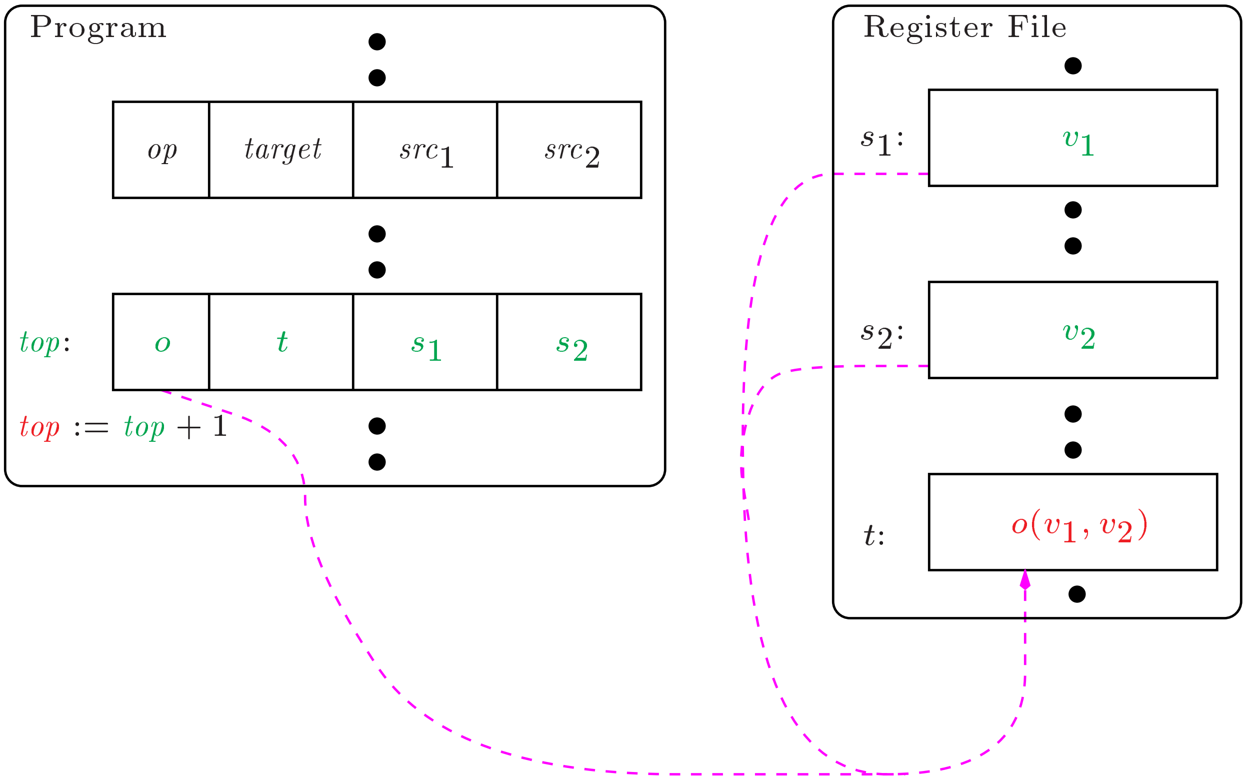
Modern processors execute many instructions in parallel, in an order often different from their program order – out-of-order execution.

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

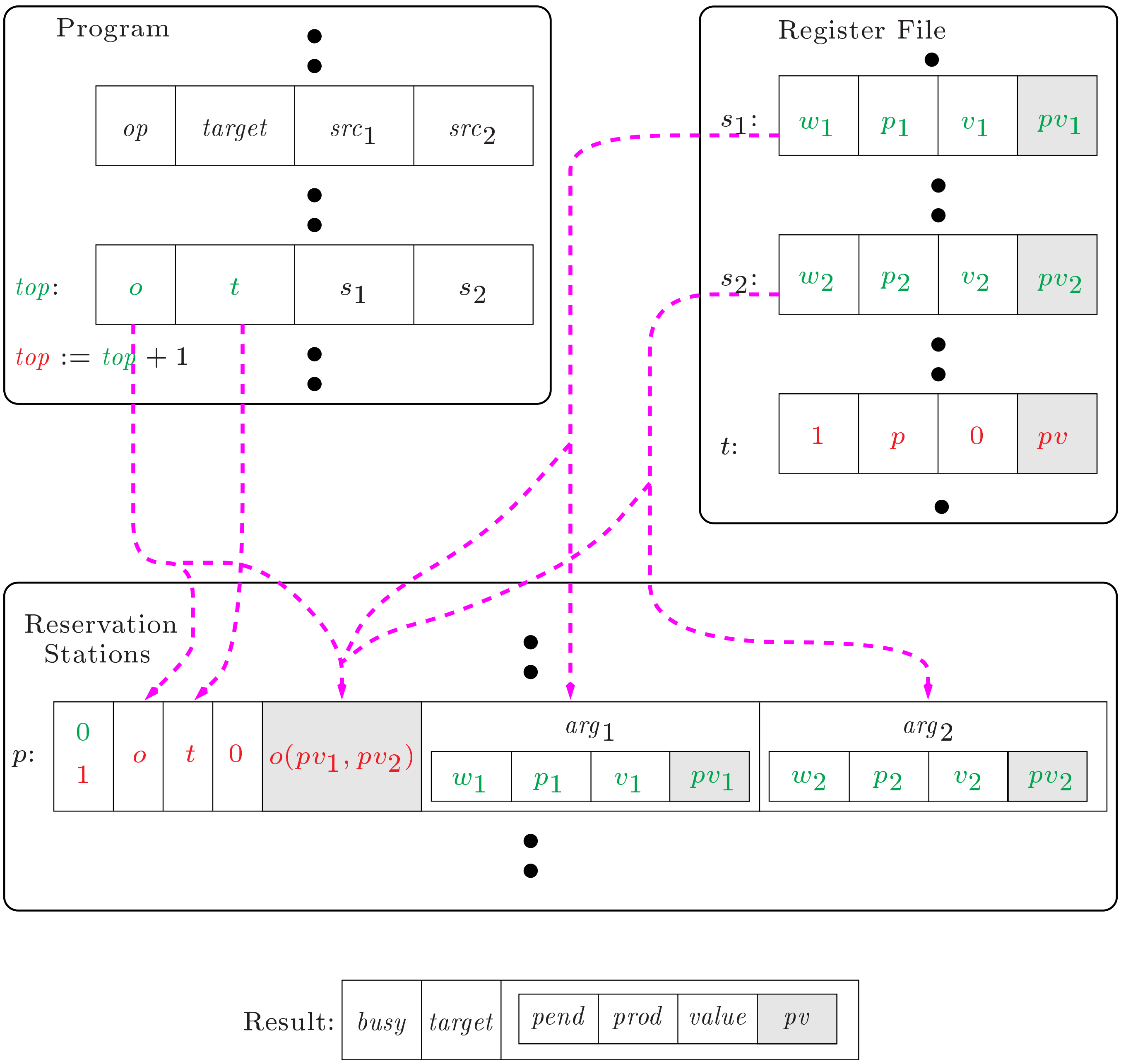
To handle this problem we introduce a system SEQ, representing the sequential reference model and a system DES, representing the design to be verified.

Typically, SEQ executes every instruction as it arrives, while DES 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.

Example Reference Model: System seq



Example Design to be Verified: System Tomasulo



An Abstract Model: Execution Automata

A finite-state automaton augmented with a finite set of data registers and a finite set of operation registers.

Programs consist of a list of instructions using uninterpreted functions as operations. E.g.

$$r_1 := f_1(r_1, r_2),\ r_2 := f_2(r_2, r_1),\ \ldots,$$

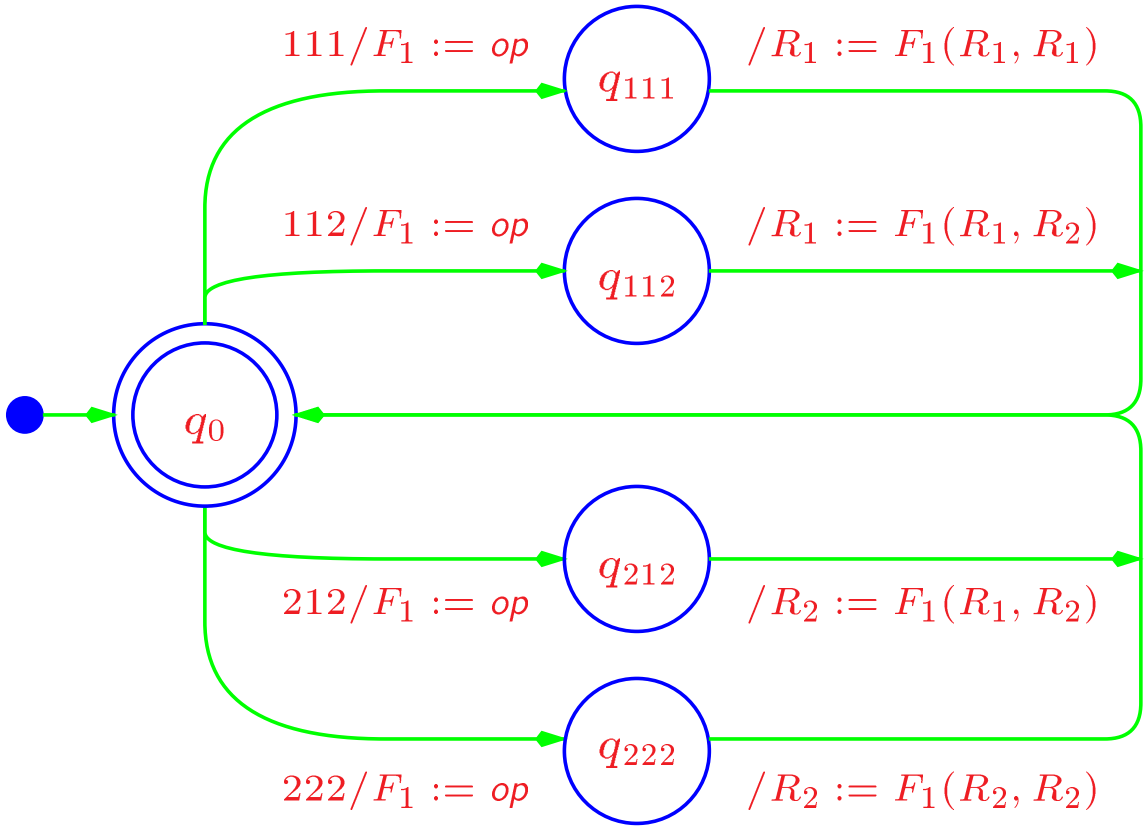
encoded as

$$\langle f_1, 112\rangle,\ \langle f_2, 221\rangle\ \ldots,$$

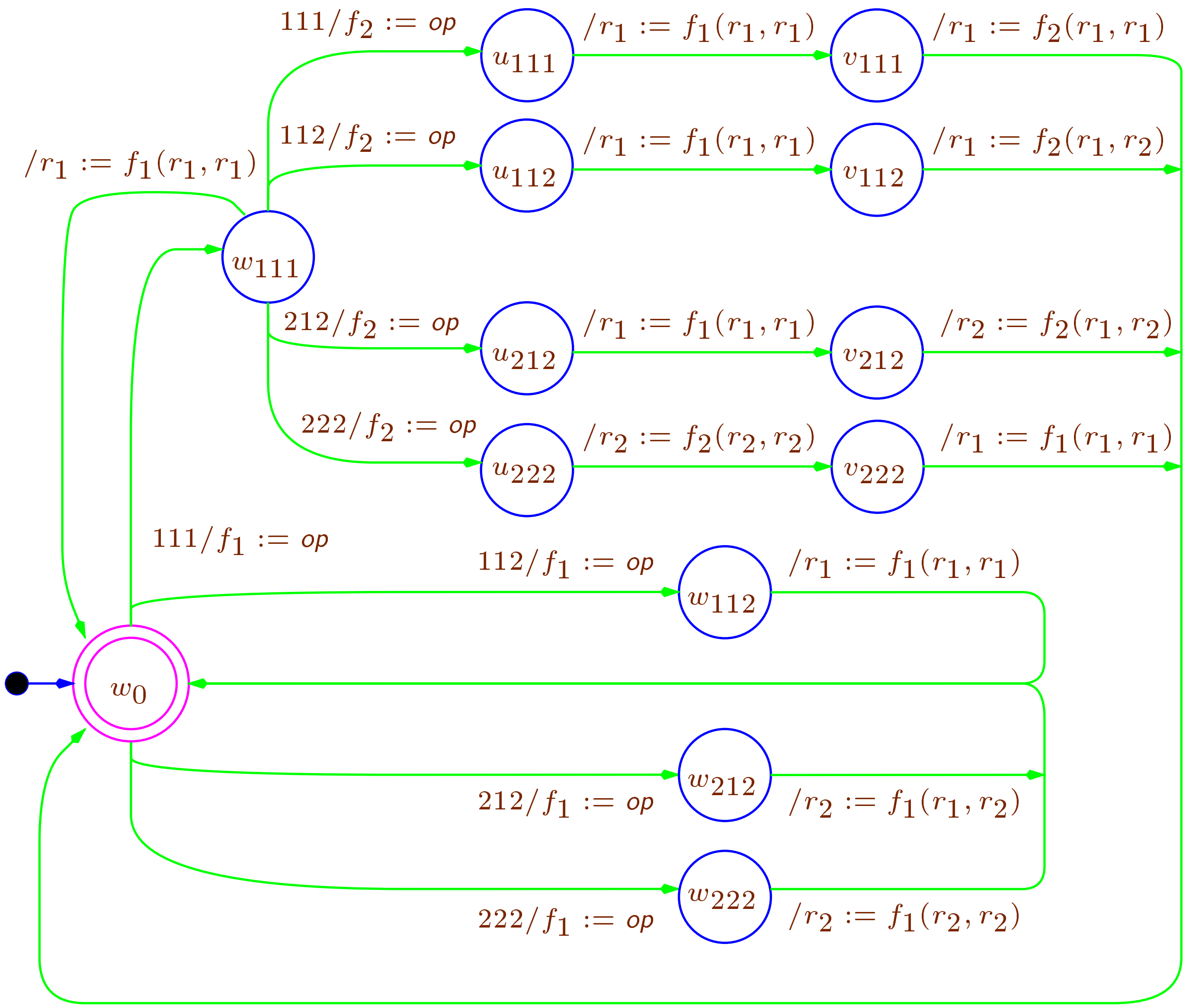
At any step, the execution automaton may

- Read an instruction, placing the operation in an operation register and move to another location (control state) according to the register-designation field of the instruction.
- Apply an operation from an operation register to two data registers (possibly the same) and store the result in a data register.

Execution Automaton SEQ



Execution Automaton PERM



Sample Joint Computation

$$\begin{array}{ll}
\rightarrow\langle +, \quad 112\rangle & \\
\langle *, \quad 111\rangle & \mathrm{SEQ}: \quad \langle q_0; \quad R_1\!:1, R_2\!:2; \;\; F_1\!:-\rangle \\
\langle +, \quad 222\rangle & \mathrm{PERM}: \langle w_0; \;\; r_1:1,\, r_2:2; \;\; f_1:-,\, f_2\!:-\rangle
\end{array}$$

$$\begin{array}{ll}
\langle +, \quad 112\rangle & \\
\rightarrow\langle *, \quad 111\rangle & \mathrm{SEQ}: \quad \langle q_{112}; \quad R_1\!:1, R_2\!:2; \;\; F_1\!:+\rangle \\
\langle +, \quad 222\rangle & \mathrm{PERM}: \langle w_{112}; \;\; r_1:1,\, r_2:2; \;\; f_1:+,\, f_2\!:-\rangle
\end{array}$$

$$\begin{array}{ll}
\langle +, \quad 112\rangle & \\
\rightarrow\langle *, \quad 111\rangle & \mathrm{SEQ}: \quad \langle q_0; \quad R_1\!:3, R_2\!:2; \;\; F_1\!:+\rangle \\
\langle +, \quad 222\rangle & \mathrm{PERM}: \langle w_0; \;\; r_1:3,\, r_2:2; \;\; f_1:+,\, f_2\!:-\rangle
\end{array}$$

$$\begin{array}{ll}
\langle +, \quad 112\rangle & \\
\langle *, \quad 111\rangle & \mathrm{SEQ}: \quad \langle q_{111}; \quad R_1\!:3, R_2\!:2; \;\; F_1\!:*\rangle \\
\rightarrow\langle +, \quad 222\rangle & \mathrm{PERM}: \langle w_{111}; \;\; r_1:3,\, r_2:2; \;\; f_1:*,\, f_2\!:-\rangle
\end{array}$$

$$\begin{array}{ll}
\langle +, \quad 112\rangle & \\
\langle *, \quad 111\rangle & \mathrm{SEQ}: \quad \langle q_0; \quad R_1\!:9, R_2\!:2; \;\; F_1\!:*\rangle \\
\rightarrow\langle +, \quad 222\rangle & \mathrm{PERM}: \langle w_{111}; \;\; r_1:3,\, r_2:2; \;\; f_1:*,\, f_2\!:-\rangle
\end{array}$$

$$\begin{array}{ll}
\langle +, \quad 112\rangle & \\
\langle *, \quad 111\rangle & \mathrm{SEQ}: \quad \langle q_{222}; \quad R_1\!:9, R_2\!:2; \;\; F_1\!:+\rangle \\
\langle +, \quad 222\rangle & \mathrm{PERM}: \langle u_{222}; \;\; r_1:3,\, r_2:2; \;\; f_1:*,\, f_2\!:+\rangle \\
\rightarrow &
\end{array}$$

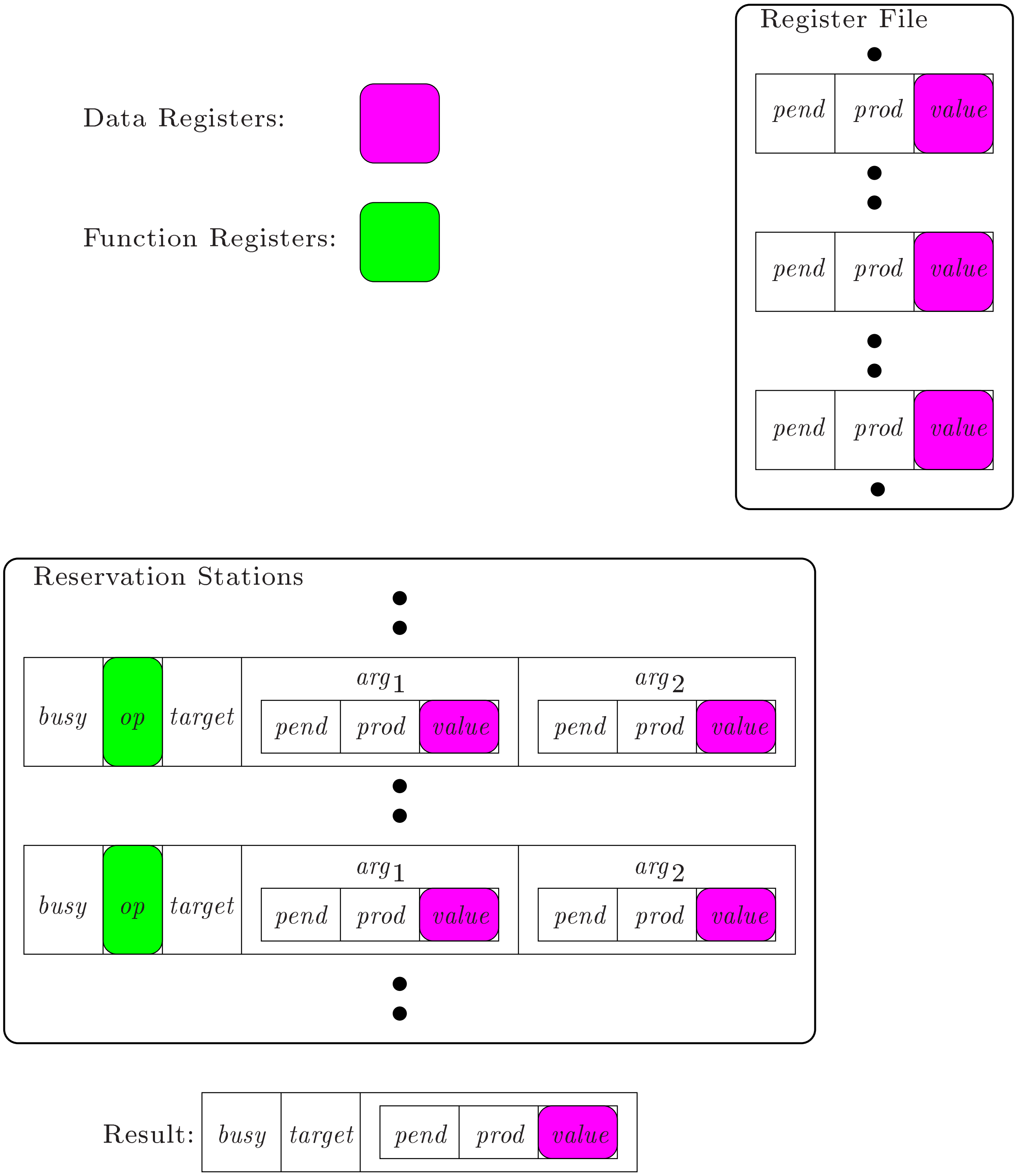
$$
\begin{array}{lll}
\langle +, & 112\rangle & \\
\langle *, & 111\rangle & \text{SEQ}: \quad \langle q_0; \quad R_1{:}\,9, R_2{:}\,4; \;\; F_1{:}\,+\rangle \\
\langle +, & 222\rangle & \text{PERM}: \langle v_{222}; \;\; r_1 : 3, \; r_2 : 4; \;\; f_1 : *, \; f_2{:}\,+\rangle
\end{array}
$$

$\longrightarrow$

$$
\begin{array}{lll}
\langle +, & 112\rangle & \\
\langle *, & 111\rangle & \text{SEQ}: \quad \langle q_0; \quad R_1{:}\,9, R_2{:}\,4; \;\; F_1{:}\,+\rangle \\
\langle +, & 222\rangle & \text{PERM}: \langle w_0; \;\; r_1 : 9, \; r_2 : 4; \;\; f_1 : *, \; f_2{:}\,+\rangle
\end{array}
$$

$\longrightarrow$

Tomasulo as an Execution Automaton



Abstract Execution Automata

Abstract execution automata (AEA) read abstract programs, and yield symbolic expressions as the final values in their registers.

For example, running automaton SEQ over the abstract program

$$\langle f_1, 112\rangle,\ \langle f_2, 111\rangle,\ \langle f_3, 222\rangle,$$

yields the following final data state:

$$\begin{array}{ll} R_1 : & f_2(f_1(c_1, c_2),\ f_1(c_1, c_2)) \\ R_1 : & f_3(c_2, c_2) \end{array}$$

Verification Reduced to Equivalence

The verification problem (over uninterpreted operations) is reduced to proving equivalence between the AEA DES representing the (out-of-order) design and the AEA SPEC representing the reference machine.

Linkage with Program Schemata: The Bad News

The AEA model closely resembles the program schemata model [LPP70] proposed for studying structural (data-independent) program transformations such as elimination of go-to's.

The main conclusion from this similarity is:

The equivalence problem for AEA's is undecidable.

Undecidability is established by simulating a Turing Machine, using uninterpreted functions.

a_0	a_1	a_2	a_3	a_4	a_5	$\bullet\bullet\bullet$
c	$f(c)$	$f(f(c))$	$f^3(c)$	$f^4(c)$	$f^5(c)$	$\bullet\bullet\bullet$
$p_0(c)$	$p_1(f(c))$	$p_2(f^2(c)$	$p_3(f^3(c))$	$p_4(f^4(c))$	$p_5(f^5(c))$	$\bullet\bullet\bullet$

The ability to simulate a TM depends on the consistency of repeated questions applied to the same arguments. Implies an unbounded memory.

Ensuring Bounded Memory: Forgetful AEA's

An AEA with a bound on the number of times each uninterpreted function is applied, is called a forgetful abstract execution automaton (FAEA). A very natural assumption for AEA's.

$$\begin{array}{ll} \rightarrow & \langle h_1, \; 21\rangle \\ & \langle h_2, \; 31\rangle \\ & \langle h_3, \; 23\rangle \end{array} \quad \left\langle \begin{array}{llll} R_1\!: & * & F_1\!: & - \\ R_2\!: & * & F_2\!: & - \\ R_3\!: & * & F_3\!: & - \end{array} \right\rangle$$

$$\begin{array}{ll} & \langle h_1, \; 21\rangle \\ \rightarrow & \langle h_2, \; 31\rangle \\ & \langle h_3, \; 23\rangle \end{array} \quad \left\langle \begin{array}{llll} R_1\!: & * & F_1\!: & h_1 \\ R_2\!: & F_1(R_1) & F_2\!: & - \\ R_3\!: & * & F_3\!: & - \end{array} \right\rangle$$

$$\begin{array}{ll} & \langle h_1, \; 21\rangle \\ & \langle h_2, \; 31\rangle \\ \rightarrow & \langle h_3, \; 23\rangle \end{array} \quad \left\langle \begin{array}{llll} R_1\!: & * & F_1\!: & h_1 \\ R_2\!: & F_1(R_1) & F_2\!: & h_2 \\ R_3\!: & F_2(R_1) & F_3\!: & - \end{array} \right\rangle$$

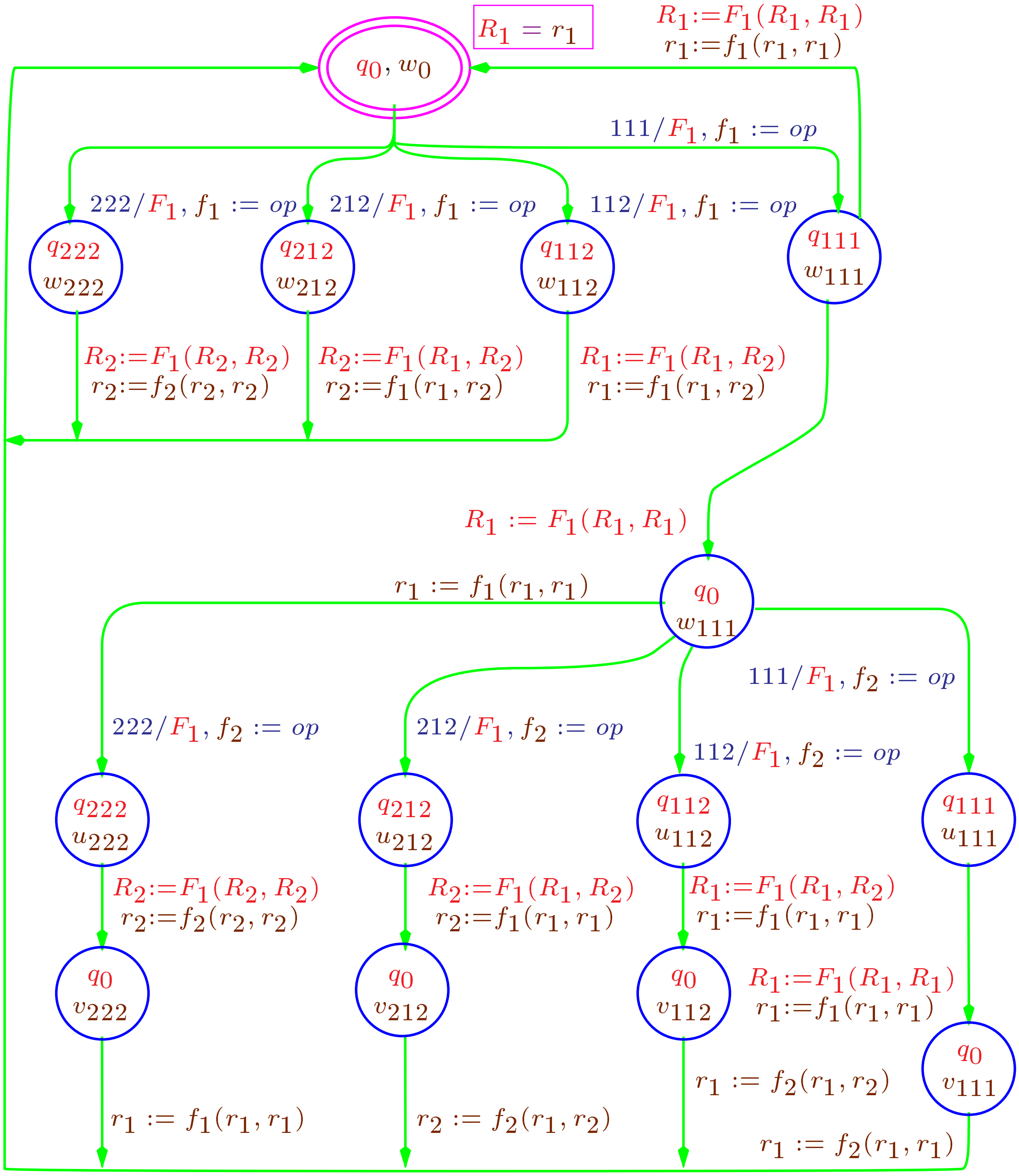
$$\begin{array}{ll} & \langle h_1, \; 21\rangle \\ & \langle h_2, \; 31\rangle \\ & \langle h_3, \; 23\rangle \\ \rightarrow & \end{array} \quad \left\langle \begin{array}{llll} R_1\!: & * & F_1\!: & h_1 \\ R_2\!: & F_2(R_3) & F_2\!: & h_3 \\ R_3\!: & * & F_3\!: & - \end{array} \right\rangle$$

Equivalence is Decidable for FAEA's

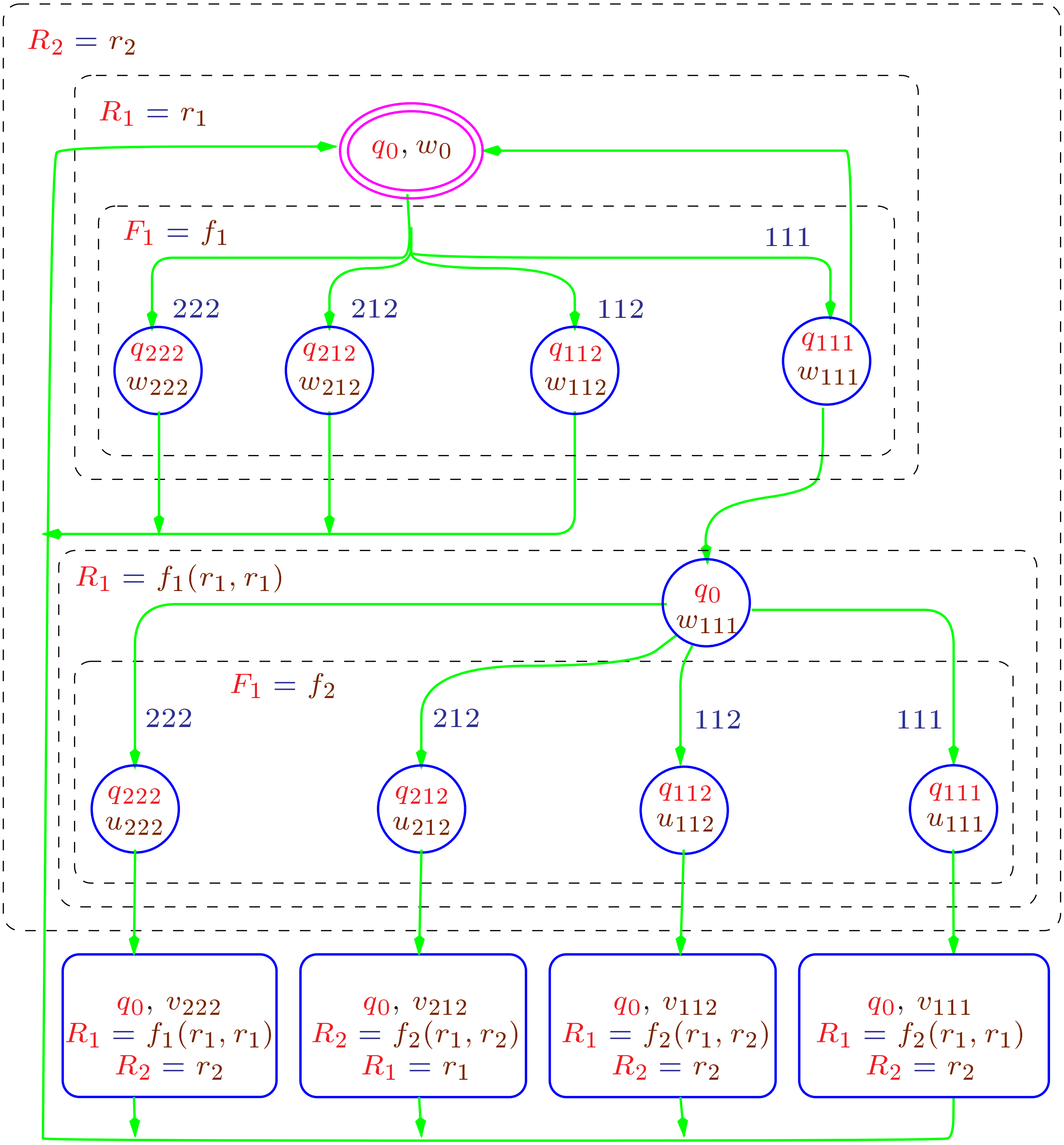
Applying a method of (forgetful) symbolic execution, we see that the size of the symbolic state representation is bounded, concluding:

Claim 1. *The Equivalence problem of forgetful abstract execution automata is decidable.*

Comparison Automaton of SEQ and PERM



Annotation


$R_2 = r_2$
$R_1 = r_1$
q_0, w_0
$F_1 = f_1$
111
222
212
112
q_{222} w_{222}
q_{212} w_{212}
q_{112} w_{112}
q_{111} w_{111}
$R_1 = f_1(r_1, r_1)$
q_0 w_{111}
$F_1 = f_2$
222
212
112
111
q_{222} u_{222}
q_{212} u_{212}
q_{112} u_{112}
q_{111} u_{111}
q_0, v_{222} $R_1 = f_1(r_1, r_1)$ $R_2 = r_2$
q_0, v_{212} $R_2 = f_2(r_1, r_2)$ $R_1 = r_1$
q_0, v_{112} $R_1 = f_2(r_1, r_2)$ $R_2 = r_2$
q_0, v_{111} $R_1 = f_2(r_1, r_1)$ $R_2 = r_2$

The Model Theoretic Approach

Identify a single concrete data domain which can faithfully represent all the diversity that can be demonstrated by any data domain.

We recommend

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

where

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

Justification

Assume that the design and the specification (reference machine) can both be modeled by an execution automaton, which executes every program instruction at most once. Let $\mathcal{D}$ and $\mathcal{S}$ denote these automata.

We say that an input program π distinguishes between automata $\mathcal{D}$ and $\mathcal{S}$ if the two automata produce different results after completely executing π.

Claim 2. *Automata $\mathcal{D}$ and $\mathcal{S}$ can be distinguished by an input program π iff $\mathcal{D}$ and $\mathcal{S}$ can be distinguished by a $\mathcal{B}_4$-program.*

Based on the lemma that if t_1 and t_2 are two unequal symbolic terms such that no term contains two occurrences of the same function (operation) symbol, then there exists β, a $\mathcal{B}_4$-interpretation of the function symbols, such that $\beta(t_1) \neq \beta(t_2)$.

Corollary 1. *The Equivalence problem of execution automata is decidable.*

Lemma

If $t_1 \neq t_2$, then $\exists B \in \mathcal{B}_4$, such that

$$B(t_1) \neq B(t_2)$$

Proof:
By induction on the size of the smaller t_i.

Case: $t_1 = f(u_1, v_1) \neq g(u_2, v_2) = t_2$

Take $B(f) = \beta_0$ and $B(g) = \beta_3$, leading to:

$$B(t_1) \;=\; \beta_0(\ldots) \;=\; 0 \;\neq\; 1 \;=\; \beta_3(\ldots) \;=\; B(t_2).$$

Case: $t_1 = f(u_1, v_1) \neq f(u_2, v_2) = t_2$

Then $u_1 \neq u_2$ or $v_1 \neq v_2$. Assume $u_1 \neq u_2$. By induction, find B such that $B(u_1) \neq B(u_2)$. Take $B^* = B \cup [B(f) = \beta_1]$, leading to:

$$\begin{array}{rclcl} B^*(t_1) & = & \beta_1(B(u_1), \ldots) & = & B(u_1) \;\neq \\ B^*(t_2) & = & \beta_1(B(u_2), \ldots) & = & B(u_2). \end{array}$$

The $\mathcal{B}_4$-Interpretation of EA's

Restriction of the data/operations domain of an EA $\mathcal{A}$ to $\mathcal{B}_4$, we obtain an automaton $\mathcal{B}_4(\mathcal{A})$ whose input programs are words over the alphabet

$$\langle i;\ t, s_1, s_2\rangle,$$

where $i \in \{0, \ldots, 3\}$ and $t, s_1, s_2 \in \{1, \ldots, m\}$, and whose states have the form

$$\langle \ell;\ d_1, \ldots, d_m;\ op_1, \ldots, op_n\rangle,$$

where $d_1, \ldots, d_m \in \{0, 1\}$ and $op_1, \ldots, op_n \in \{0, \ldots, 3\}$.

We also define $\mathcal{B}_4(\mathcal{A})^0$ to be the automaton $\mathcal{B}_4(\mathcal{A})$ whose accepting states are all the states s such that $s.\ell$ is terminal and $s.r_1 = 0$. This is the automaton accepting all programs which lead to a final result of $r_1 = 0$. In a similar way, we define $\mathcal{B}_4(\mathcal{A})^1$ to be the automaton $\mathcal{B}_4(\mathcal{A})$ whose accepting states are all the states s such that $s.\ell$ is terminal and $s.r_1 = 1$.

Claim 3. *Automata $\mathcal{D}$ and $\mathcal{S}$ are equivalent iff*

$$\mathcal{B}_4(\mathcal{D})^0 \sim \mathcal{B}_4(\mathcal{S})^0 \qquad \textit{and} \qquad \mathcal{B}_4(\mathcal{D})^1 \sim \mathcal{B}_4(\mathcal{S})^1$$

A New Era in Hardware Architecture

An exciting new development is the introduction of the EPIC (Explicitly controlled Parallel Instruction Computing) family of processor architectures, in which the tasks of analyzing the dependencies and scheduling the instructions are shifted back from hardware to software. Architecture specific optimizing compilers are expected to perform the analysis and instruction scheduling and control the hardware by means of very long instructions.

This family of architectures is also known by the generic name IA-64 and the company specific (Intel+HP) code name Merced.

Academic versions of architecture specific optimizing compilers are freely available from the Trimaran project, jointly developed by NYU, HP, and the Compiler Group at the University of Illinois.

An exciting challenge is to combine the expertise developed in translation validation and the use of uninterpreted functions for out-of-order verification and develop a method for validating the translation of EPIC compilers such as the Trimaran compiler.