# Herbrand Automata for Hardware Verification*

W. Damm[1], A. Pnueli[2], and S. Ruah[3]

[1] OFFIS, Oldenburg, Germany
[2] Weizmann Institute of Science, Rehovot, Israel. Contact author at amir@wisdom.weizmann.ac.il
[3] Weizmann Institute of Science, Rehovot, Israel

**Abstract.** The paper presents the new computational model of *Herbrand engines* which combines finite-state control with uninterpreted data and function registers, thus yielding a finite representation of infinite-state machines. Herbrand engines are used to provide a high-level model of out-of-order execution in the design of micro-processors. The problem of verifying that a highly parallel design for out-of-order execution correctly implements the Instruction Set Architecture is reduced to establishing the equivalence of two Herbrand engines. We show that, for a reasonably restricted class of such engines, the equivalence problem is decidable, and present two algorithms for solving this problem.
Ultimately, the appropriate statement of correctness is that the out-of-order execution produces the same final state (and all relevant intermediate actions, such as writes to memory) as a purely sequential machine running the same program.

## 1 Introduction

Modern processor architectures such as the PowerPC or the DEC Alpha employ aggressive implementation techniques to sustain peak-throughput of instructions. Multiple functional units inside the data-path allow for concurrent execution of multiple instructions and allow to hide latencies stemming from data-dependencies as well as varying pipeline delays. The design of controllers maintaining consistency to sequential program execution on the face of a mixture of out-of-order execution of instructions, speculative execution of instructions, interrupts, and load/store buffers is both challenging and error-prone (c.f. e.g. [7]).

As the complexity of designs have grown, so has the need for advanced validation techniques. The need for formal verification tools to support industrial design processes is now recognized [10] and is apparent by the introduction of commercial products. However, while circuit comparison and - to a lesser extent - property verification based on symbolic model- checking [4] have found their way into industrial applications (c.f. e.g. [5, 1]) , coping with the complexity of industrial designs remains a key challenge, requiring complementary proof-methods

* This research was supported in part by a gift from Intel, a grant from the Minerva foundation, and an *Infrastructure* grant from the Israeli Ministry of Science and the Arts.

to be combined in verification environments. In particular, proof-methods based on decision procedures for first-order logic [3], [9] have gained high attention due to their ability to naturally cope with abstractions from data-computations when analyzing complex control circuitry while allowing full automation of the proof (in contrast to approaches relying on interactive theorem proving such as [6,13,15]).

In this paper we introduce the new computational model of *Execution Automata* and the model of *Herbrand Engines* based on it. This model combines finite-state control with uninterpreted data and function registers, thus yielding a finite representation of infinite-state machines. Herbrand engines are used to provide a high-level model of the proposed implementation as well as the reference model which performs the same program in a strictly sequential mode. The problem of verifying that a proposed out-of-order design correctly implements the reference model (which serves as a specification) is reduced to checking the equivalence of two Herbrand Engines.

To make this problem (which is undecidable in the general case) tractable, we restrict our attention to automata which apply each operation requested by an instruction only a bounded number of times. For such *Automata of Bounded Application* (ABA's), we show that the equivalence problem is decidable. In Section 3, we present a decision algorithm for a general ABA, which is based on annotation of all reachable control states by invariants that hold on all visits to this location. In Section 4, we present a more efficient decision algorithm for the special case of 1-bounded automata, which is based on the comparison of the two automata over programs with operations taken from a limited set of four boolean operations. This more efficient algorithm can be performed using BDD machinery [2] and the techniques of symbolic model checking [4]. The class of 1-bounded automata is sufficient to model the out-of-order-execution algorithms we aim to verify, as will be later described.

Both algorithms we present have been implemented. The annotation-based algorithm of Section 3 has been implemented on the tool OOO-CHECK developed at the Weizmann Institute, and has been used to check small examples such as the one presented here. The sampled-operations algorithm of Section 4 has been implemented over the TLV system [14] and was used for verifying a small configuration of the Tomasulo algorithm [7] supporting out-of-order execution of floating point instructions, for small configurations.

While the initial motivation for the development of the Herbrand engines formalism is that of hardware verification, we strongly believe that the model has a much wider applicability and can be utilized in other contexts for the verification and refinement between reactive systems.

## 2 Execution Automata and Engines

Our approach for verifying the correctness of a proposed design for out-of-order execution is based on the consideration of two *execution automata* (defined later). One execution automaton, to which we refer as SEQ, represents the strictly sequential execution of the presented program, and serves as the *specification*.

The second, more concrete, execution automaton, to which we refer as DES, represents the proposed design for out-of-order execution and is considered as the implementation. Our objective is to show that DES refines SEQ.

We first present the definitions of Execution Automata and their equivalence. We assume a fixed *signature* $\Xi = \{c, h_0, h_1, \ldots\}$ which consists of a set of function symbols $c, h_0, h_1 \ldots$, where $c$ has arity 0 (is a constant), and all other function symbols have arity 2. We refer to $\mathcal{O}ps = \{h_0, h_1, \ldots\}$ as the set of *operations*.

An *Execution Automaton* (EA) $A = \langle \Sigma, Q, \mathcal{R}, \mathcal{F}, L, E, q_0, \mathcal{A} \rangle$ consists of:

- $\Sigma$ – A finite *input alphabet*.
- $Q$ – A set of *locations* representing the control part of the automaton state.
- $\mathcal{R} = \{R_1, \ldots, R_b\}$ – A set of *data registers*. These registers store the intermediate results in the execution of a program.
- $\mathcal{F} = \{F_1, \ldots, F_p\}$ – A set of *function registers*. These registers store the operations corresponding to pending instructions.
- $L$ – A set of *labels*. These labels are associated with transitions between locations and describe the condition under which a transition can be taken, as well as the actions executed while taking the transition.
  We distinguish between *input* labels and *internal action* labels. An input label has the form $a/F^1, \ldots, F^k := op$, for some $a \in \Sigma$ and $F^1, \ldots, F^k \in \mathcal{F}$. An internal action label has the form *act*, where *act* is an *action* which is a list of assignments of the form $R_i := F_f(R_j, R_k)$.
- $E \subseteq Q \times L \times Q$ – A set of *transitions*.
- $q_0 \in Q$ – An *initial location*.
  An *atomic formula* for $A$ is an equation of the form $R_i = t$ where $R_i \in \mathcal{R}$ and $t$ is a term which refers only to the constant $c$, to the variables $R_1, \ldots, R_b$ and to the function registers $F_1, \ldots, F_p$. For example $R_3 = F_2(R_1, R_2)$ and $R_3 = R_4$ are atomic formulas. An *expressible assertion* is a conjunction of atomic formulas. Let $\mathcal{L}$ be the set of all expressible assertions.
- $\mathcal{A} : Q \mapsto \mathcal{L}$ - An acceptance condition. With each location $q \in Q$, we associate the expressible assertion $\mathcal{A}(q)$, which imposes a condition on the automaton states that are accepted while at location $q$. For most locations the acceptance condition $\mathcal{A}(q)$ is just *false*.

When there is no danger of confusion, we will refer to an execution automaton simply as an automaton. The input to an EA is a $(\Xi-)$*program* $\pi$ which is a sequence of *instructions* of the form $\pi$: $\langle h^1, a_1\rangle, \langle h^2, a_2\rangle, \langle h^3, a_3\rangle, \ldots, \langle h^k, a_k\rangle$, where $h^i \in \mathcal{O}ps$ and $a_i \in \Sigma$, for every $i = 1, \ldots, k$. The function symbol $h^i$ is the operation, while $a_i$ is the register designation field of the instruction.

For example, in Fig. 1, we present an execution automaton SEQ equipped with five control locations ($\{q_0$, $q_{111}$, $q_{112}$, $q_{212}$, $q_{222}\}$), two data registers ($R_1$, $R_2$) and one function register ($F_1$). Location $q_0$ is both initial and accepting.

In the simple cases that all the acceptance conditions are either *true* or *false* we adopt the graphic convention by which locations with a true acceptance conditions are drawn as double circles, while all other locations are drawn as single circles.

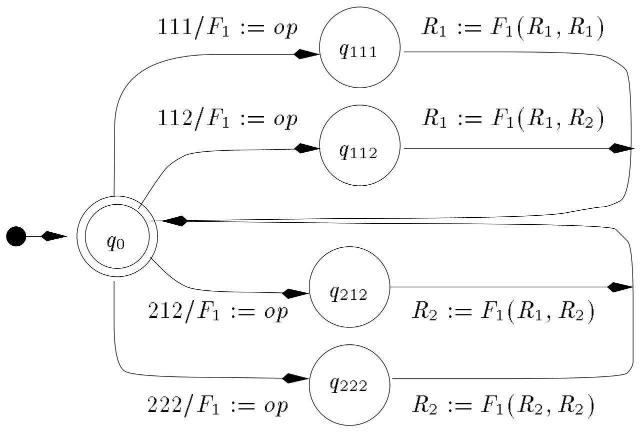


**Fig. 1.** An execution automaton SEQ.

The automaton of Fig. 1 represents the sequential execution of programs by a machine with 2 registers: $R_1$ and $R_2$. Each register designation in the input alphabet consists of three register numbers, ranging between 1 and 2. These three numbers specify the target register, the first-operand register, and the second-operand register, respectively. Starting at $q_0$, the automaton reads the next instruction and branches to one of 4 locations, while storing the operation in function register $F_1$. The 4 locations $q_{111}, \ldots, q_{222}$ correspond to 4 different modes of instructions selecting a target register and two operand registers out of the set $\{R_1, R_2\}$. Once in any of these locations, the automaton proceeds to execute the recently read instruction, applying the operation in $F_1$ to the appropriate source registers and storing the result in the appropriate target register.

A *structure* $\mathcal{B} = (\mathcal{D}, \alpha)$ for the signature $\Xi$ consists of a set of elements $\mathcal{D}$ called the *domain* of $\mathcal{B}$ and a mapping $\alpha$ which maps the constant symbol $c$ into a concrete element $c^\alpha \in \mathcal{D}$ and maps each function symbol $h_i \in \Xi$ into a concrete function $\alpha[h_i] = h_i^\alpha$ where, $h_i^\alpha : \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$ is a binary function from $\mathcal{D}$ to itself, for each $i = 0, 1, \ldots$.

An *execution engine* (EE) $(A, \mathcal{B})$ consists of an execution automaton $A$ and a structure $\mathcal{B}$.

A *state* $s = \langle q; d_1, \ldots, d_b; f_1, \ldots, F_p\rangle$ of an EE $(A, \mathcal{B})$ consists of a location $q \in Q$, a list $d_1, \ldots, d_b$ of $\mathcal{D}$-elements representing the current contents of the data registers, and a list $f_1, \ldots, F_p \in \mathcal{O}ps$ of operations (function symbols) representing the current contents of the function registers. We can view $s$ as a record with fields $q : Q$, $R_1, \ldots, R_b : \mathcal{D}$, and $F_1, \ldots, F_p : \mathcal{O}ps$.

The initial state of EE $(A, \mathcal{B})$ is given by $s_0$: $\langle q_0;\ \underbrace{c^\alpha, \ldots c^\alpha}_{b};\ \underbrace{h_0, \ldots, h_0}_{e}\rangle$.

Let $\mathcal{S}$ denote the set of all states of an EE $(A, \mathcal{B})$. We denote by $X = (\mathcal{O}ps \times \Sigma) \cup \Lambda$ the set of *input elements* which can be either an instruction $\langle h, a\rangle$ or the empty word $\Lambda$. We define a *transition function* $\delta : \mathcal{S} \times X \mapsto 2^{\mathcal{S}}$, by requiring that $s' \in \delta(s, x)$ iff one of the following holds:

1. The input element is $x = \langle f, a \rangle$, $s.q = q_i$, there exists a transition $\langle q_i;\ a/F^1, \ldots, F^k := op;\ q_j \rangle$, and
$s' \quad = \quad s$ **with** $q := q_j,\ F^1 := f, \ldots, F^k := f$.
2. The input element is $x = \Lambda$, $s.q = q_i$, there exists a transition $\langle q_i;\ R_k := F_l(R_m, R_n);\ q_j \rangle$, $s.F_l = f$, and
$s' \quad = \quad s$ **with** $q := q_j,\ R_k := f^{\alpha}(s.R_m, s.R_n)$.
3. Variants of case 2, where the action contains several assignments.

These conditions can be illustrated on some cases of automaton SEQ of Fig. 1 with the signature $\Xi = \{c, h_+\}$ and the $\Xi$-structure $\mathcal{B}_+ : (\mathsf{N}, \alpha)$, where $c^{\alpha} = 1$, and $h_+$ is mapped by $\alpha$ into integer addition. Let $s_0 : \langle q_0;\ 1, 1;\ - \rangle$ be an initial state. Then, $s_1 : \langle q_{212};\ 1, 1;\ h_+ \rangle \in \delta(s_0,\ \langle h_+, 212 \rangle)$, due to the transition connecting $q_0$ to $q_{212}$ and labeled by $212/F_1 := op$

In a similar way, $s_2 : \langle q_0;\ 1, 2;\ h_+ \rangle \in \delta(s_1,\ \Lambda)$, due to the transition connecting $q_{212}$ to $q_0$ and labeled by $R_2 := F_1(R_1, R_2)$.

The *concatenation* of input elements $x_1, \ldots, x_n$, denoted by $x_1 * \cdots * x_n$, can be defined in the obvious way as the concatenation of all non-empty elements among $x_1, \ldots, x_n$.

The state sequence $s_1, \ldots, s_k$ is defined to be a *run* of the execution engine $(A, \mathcal{B})$ over the program $\langle h^1, a_1 \rangle, \ldots, \langle h^n, a_n \rangle$ if $s_1$ is the initial state and there exist input elements $x_1, \ldots, x_{k-1}$, such that $\langle h^1, a_1 \rangle, \ldots, \langle h^n, a_n \rangle = x_1 * \cdots * x_{k-1}$ and $s_{i+1} \in \delta(s_i, x_i)$ for every $i = 1, \ldots, k{-}1$.

For example, $\langle q_0;\ 1, 1;\ - \rangle$, $\langle q_{111};\ 1, 1;\ h_+ \rangle$, $\langle q_0;\ 2, 1;\ h_+ \rangle$ is a run of the EE $(\text{SEQ}, \mathcal{B}_+)$ over the (1-instruction) program $\langle h_+, 111 \rangle$.

The state sequence $w = s_1, \ldots, s_k$ is defined to be an *accepting run* of $(A, \mathcal{B})$ over a program $\pi = \langle h^1, a_1 \rangle, \ldots, \langle h^n, a_n \rangle$ if $w$ is a run of $(A, \mathcal{B})$ over $\pi$ and the accepting condition of $s_k.q \in Q$ is satisfied by $s_k$, i.e., $s_k \models \mathcal{A}(s_k.q)$.

For example, the two state sequences $\langle q_0;\ 1, 1;\ - \rangle$, $\langle q_{111};\ 1, 1;\ h_+ \rangle$ and $\langle q_0;\ 1, 1;\ - \rangle$, $\langle q_{111};\ 1, 1;\ h_+ \rangle$, $\langle q_0;\ 2, 1;\ h_+ \rangle$ are both runs of $(\text{SEQ}, \mathcal{B}_+)$ over the program $\langle h_+, 111 \rangle$, but only the second is accepting.

An EE $(A, \mathcal{B})$ is defined to be *universal* if it has a run over every program $\pi$ and every run over program $\pi$ can be extended to an accepting run over $\pi$. The *automaton* $A$ is said to be *universal* if $(A, \mathcal{B})$ is universal for every structure $\mathcal{B}$. For example, SEQ is a universal automaton.

### 2.1 Equivalent Engines and Automata

For a run $w = s_1, \ldots, s_k$ of EE $(A, \mathcal{B})$, we denote by *outcome*$(w)$ the value of register $R_1$ at the last state of $w$. That is, *outcome*$(w) = s_k.R_1 \in \mathcal{D}_{\mathcal{B}}$.

Let $E_1 = (A_1, \mathcal{B})$ and $E_2 = (A_2, \mathcal{B})$ be two execution engines with the same structure. The two execution engines $E_1$ and $E_2$ are defined to be *equivalent*, denoted $E_1 \sim E_2$, if *outcome*$(w_1) =$ *outcome*$(w_2)$, for every $w_1$, an accepting run of $E_1$ over a program $\pi$, and every $w_2$ an accepting run of $E_2$ over the same program $\pi$. That is, $E_1$ and $E_2$ are equivalent if they produce the same final value in their respective $R_1$ registers after executing (reading) the same program.

The two automata $A_1$ and $A_2$ are said to be *equivalent*, denoted $A_1 \sim A_2$, if $(A_1, \mathcal{B}) \sim (A_2, \mathcal{B})$ for every structure $\mathcal{B}$.

Automaton SEQ was introduced as an example of a reference machine that serves as a specification. In Fig. 2, we present automaton PERM, which is an example of a possible out-of-order design. Automaton PERM employs two function registers.

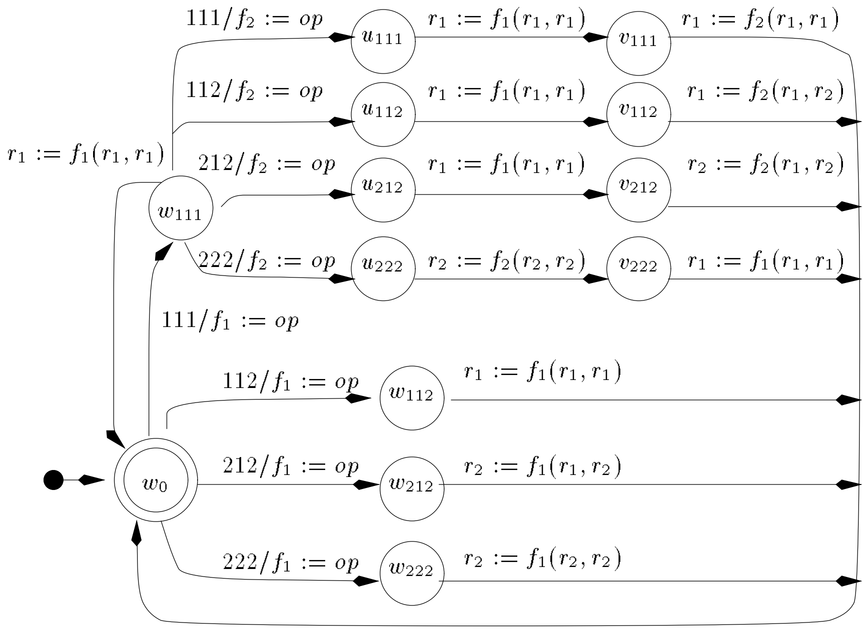


**Fig. 2.** Execution automaton PERM

While not representing any interesting design, the overly simplistic automaton PERM captures at least some elements of out-of-order execution, in that whenever it detects two consecutive instructions of the form $r_1 := f(r_1, r_1)$; $r_2 := g(r_2, r_2)$, the automaton performs these two instructions in reverse order, i.e., first assigning $g(r_2, r_2)$ to $r_2$ and then assigning $f(r_1, r_1)$ to $r_1$.

Proving that automaton PERM is a correct implementation of automaton SEQ amounts to showing that the two automata are equivalent.

### 2.2 Herbrand Engines

Note that when we consider verification of concrete designs, we are always interested in a particular structure and a particular set of operations. For example, the design may be intended to perform floating point arithmetical operations.

The approach adopted in this paper generalizes the problem of establishing equivalence between execution engines. Instead of asking the question with respect to a concrete data domain and operations, such as a particular representation of floating point numbers and their operations, we pose the more general question of when are two engines equivalent with respect to *all interpretations* of the data domains and the operations, i.e. all structures. This generalization is justified by two observations:

- In most cases, designs of out-of-order execution engines are in fact correct over all interpretations. The only compatibility required is that whenever the implementation applies a particular operation to two arguments, so does the reference machine, independently of the nature of the operation.
- While the equivalence problem for concrete data structures such as integers and operations that include multiplication is known to be undecidable, the generalization to the more demanding notion of *equivalence over all interpretations* while bounding the number of applications of each function symbol makes the problem decidable as we show in this paper.

A *Herbrand Engine* (HE) is an execution engine over a special structure $\mathcal{H} = (\mathcal{D}, \alpha)$, called the *Herbrand structure*, where $\mathcal{D}$ consists of all the symbolic terms over the signature $\Xi$ and $h_i^\alpha(t_1, t_2) = h_i(t_1, t_2)$. That is, the application of $h_i^\alpha$ to the terms $t_1$ and $t_2$ is the symbolic expression $h_i(t_1, t_2)$.

A program $\pi$: $\langle h_1, a_1\rangle, \langle h_2, a_2\rangle, \langle h_3, a_3\rangle, \ldots, \langle h_k, a_k\rangle$, is called *discriminating* if $h_i \neq h_j$ for every $i \neq j$. That is, no function symbol appears more than once in the program.

Let $\mathcal{B}$ be a structure $\mathcal{B} = (\mathcal{D}, \alpha)$. The mapping $\alpha$ assigns to the constant symbol $c \in \Xi$ the concrete value $c^\alpha \in \mathcal{D}$ and to every function symbol $h_i \in \Xi$ a concrete function $h_i^\alpha : \mathcal{D} \times \mathcal{D} \mapsto \mathcal{D}$. It is straightforward to extend $\alpha$ to map an arbitrary symbolic term $t$ into a concrete element $\alpha[t] \in \mathcal{D}$.

Let $(A, \mathcal{H})$ be a Herbrand engine, and $(A, \mathcal{B})$ be an execution engine with the same automaton $A$. Using the mapping $\alpha_{\mathcal{B}}$ associated with the structure $\mathcal{B}$, it is possible to map the Herbrand state $s = \langle q; t_1, \ldots, t_b; f_1, \ldots, f_p\rangle$ into the $\mathcal{B}$-state $\alpha_{\mathcal{B}}[s] = \langle q; \alpha_{\mathcal{B}}[t_1], \ldots, \alpha_{\mathcal{B}}[t_b]; \alpha_{\mathcal{B}}[f_1], \ldots, \alpha_{\mathcal{B}}[f_p]\rangle$. For an $(A, \mathcal{H})$-run $r = s_0, \ldots, s_k$, we denote by $\alpha_{\mathcal{B}}[r]$ the state sequence $\alpha_{\mathcal{B}}[s_0], \ldots, \alpha_{\mathcal{B}}[s_k]$.

An interesting question is the relation between runs of a Herbrand engine $(A, \mathcal{H})$ over a program $\pi$ and the runs of any other execution engine $(A, \mathcal{B})$ over the same program. In one direction there is an obvious implication, stated by the following claim:

**Claim 1.** *If $r$ is a run of $(A, \mathcal{H})$ over the program $\pi$, then $\alpha_{\mathcal{B}}[r]$ is a run of $(A, \mathcal{B})$ over the same program.*

The ability to map runs of a Herbrand engine to the runs of any other engine, lead to the following claim:

**Claim 2.** *The automata $A_1$ and $A_2$ are equivalent, i.e., $A_1 \sim A_2$, iff $(A_1, \mathcal{H}) \sim (A_2, \mathcal{H})$.*

Claim 2 states that in order to establish the equivalence of automata $A_1$ and $A_2$ it is sufficient to establish their equivalence over one special structure: the Herbrand structure $\mathcal{H}$.

**Claim 3.** *For $A_1$ and $A_2$ two automata, $(A_1, \mathcal{H}) \sim (A_2, \mathcal{H})$ over all programs iff $(A_1, \mathcal{H}) \sim (A_2, \mathcal{H})$ over all discriminating programs.*

The proof of this claim relies on the fact that discriminating programs are least likely to produce "accidental" equal terms. Two Herbrand terms (in the different automata) can be equal as a result of executing a discriminating program only if the two automata applied identical operations to the same arguments throughout the execution (perhaps at different orders between independent threads of computation).

Similar to the property of equivalence, also universality of an automaton can be determined based on the examination of the corresponding Herbrand engine over discriminating programs.

**Claim 4.** *The automaton $A$ is universal iff the Herbrand engine $(A, \mathcal{H})$ is universal over all discriminating programs.*

A key observation in the proof of Claim 4 is that the Herbrand terms equality $t_1 = t_2$ implies the $\mathcal{B}$-equality $\alpha_{\mathcal{B}}[t_1] = \alpha_{\mathcal{B}}[t_2]$, for every structure $\mathcal{B}$.

We define the *comparison automaton* $\mathcal{C}(A_1, A_2)$ of the execution automata $A_1 = \langle \Sigma, Q^1, \mathcal{R}^1, \mathcal{F}^1, L^1, E^1, q_0^1, \mathcal{A}^1 \rangle$ and $A_2 = \langle \Sigma, Q^2, \mathcal{R}^2, \mathcal{F}^2, L^2, E^2, q_0^2, \mathcal{A}^2 \rangle$, to be the automaton $A = \langle \Sigma, Q, \mathcal{R}, \mathcal{F}, L, E, q_0, \mathcal{A} \rangle$, where

- $Q = \{(q^1, q^2) \mid q^1 \in Q^1, q^2 \in Q^2\}$.
- $\mathcal{R} = \mathcal{R}^1 \cup \mathcal{R}^2$.
- $\mathcal{F} = \mathcal{F}^1 \cup \mathcal{F}^2$.
- $L$ and $E$ are defined as follows:
  - For every pair of an $A_1$-transition $\langle q_i;\ a/F^1 := op;\ q_u \rangle \in E^1$ and an $A_2$-transition $\langle q_j;\ a/F^2 := op;\ q_v \rangle \in E^2$, we include in $E$ the transition
    $\langle (q_i, q_j);\ a/F^1, F^2 := op;\ (q_u, q_v) \rangle$.
    The generalization to the case that one or both of the transitions read the function symbol into more than one function register is straightforward.
  - For every pair of an $A_1$-transition $\langle q_i;\ r^1 := t_1;\ q_u \rangle \in E^1$ and an $A_2$-transition $\langle q_j;\ r^2 := t_2;\ q_v \rangle \in E^2$, we include in $E$ the transition
    $\langle (q_i, q_j);\ r^1 := t_1, r^2 := t_2;\ (q_u, q_v) \rangle$.
    The generalization to the case that one or both of the transitions perform more than one assignment is straightforward.

  Note that we only combine two input transitions and two internal transitions. In the case that one of the automata is ready to perform an internal transition and the other automaton can perform both an input and an internal transition, we also admit a combination of the internal transition of the first with an idling (do-nothing) transition of the second. This allows the second automaton to wait until both are ready to perform a joint input transition.
- $q_0 = (q_0^1, q_0^2)$.
- $\mathcal{A} : Q \mapsto \mathcal{L}$ - the acceptance condition, where $\mathcal{L}$ is the set of conjunctions of expressible assertions, and for $q = (q^1, q^2), q^1 \in Q^1, q^2 \in Q^2$ we define $\mathcal{A}((q^1, q^2)) = \mathcal{A}^1(q^1) \wedge \mathcal{A}^2(q^2) \wedge R_1^1 = R_1^2$.

The comparison automaton of SEQ and PERM is presented in Fig. 3.

## 3 Checking for Equivalence

In this section, we present a sketch of the algorithm for checking the equivalence of two Herbrand engines over all discriminating programs.

For a natural number $n$, we say that an automaton $A$ has an *application bound* $n$, if in any execution of a discriminating program no function symbol is applied more than $n$ times. For example, both SEQ and PERM have application bound of 1. An Automaton $A$ is said to be an *Application Bounded Automaton* (an ABA) if it has an application bound of $n$ for some $n > 0$. We are now ready to state our main result.

**Theorem 1.** *It is decidable whether two* ABA*'s are equivalent over all discriminating programs.*

The proof of this theorem is constructive and forms the basis for the tool OOO-CHECK which checks this equivalence.

Given ABA's $A_1$ and $A_2$, the main steps in the decision algorithm can be summarized as follows:

1. Construct the comparison automaton $C = \mathcal{C}(A_1, A_2)$.
2. Construct the finite-state automaton (FSA) $\nu(C)$ by annotating the execution automaton $C$.
3. Check that $\nu(C)$ is universal in the sense that, from every reachable location within $\nu(C)$, there exists an input-free path leading to an accepting state.

Thus, step (2) of the algorithm depends on the construction of the annotated automaton $\nu(C)$ and the establishment of its properties. Note that, if both $A_1$ and $A_2$ have an application bound of $n$, then $C$ has an application bound of $2n$.

### 3.1 Construction of the Annotated Automaton

As mentioned earlier, the intended meaning of an assertion is that it states a relation between the contents of the data and function registers in some reachable state. It is often used as an invariant associated with a control location. For example, the assertion $R_1 = f_1(r_1, r_1)$ is always true on visits to location $(q_0, w_{111})$ of the comparison automaton $\mathcal{C}(A_1, A_2)$.

An *annotation* for an ABA $A$ is a set of assertions over $A$, each of which is a conjunction of expressible atomic formulas of the forms $R_k = t$ or $F_i = F_j$. A term $t$ is called *non-trivial* if it refers to at least one function register. An annotation $\nu$ is said to be *acyclic* if we can associate with each data register $R_i$ a natural number $\kappa_i$, called the *rank* of $R_i$, such that if $\nu$ contains an assertion $R_i = t$, where $t$ is a non-trivial term referring to the data register $R_j$, then $\kappa_i > \kappa_j$. For example, an annotation that contains the assertion $R_i = F_k(R_i, R_j)$ is cyclic.

A standard finite state automaton $B = \langle \Sigma_B, Q_B, L_B, E_B, q_0^B, \mathcal{A}_B \rangle$ is said to be an *annotation automaton* corresponding to $A$ if the following hold:

- $\Sigma_B = \Sigma_A$.

- $Q_B$, the set of locations (control states) of automaton $B$ is a set of pairs $(q, \nu)$, where $q \in Q_A$ is a location of $A$ and $\nu$ is an acyclic $A$-annotation.
- $L_B = \Sigma \cup \{\epsilon\}$. That is, the labels of transitions in $B$ consist of the input alphabet $\Sigma = \Sigma_B$ or the silent label $\epsilon$ corresponding to a transition that does not read an additional input.
- There exists a transition $\langle (q_1, \nu_1); \lambda_B; (q_2, \nu_2)\rangle$ in automaton $B$ iff there exists a transition $\langle q_1; \lambda_A; q_2\rangle$ in the ABA $A$ such that the two transitions are *compatible* as defined below.
- $q_0^B = (q_0^A, R_1 = \cdots = R_b = c \wedge F_1 = \cdots = F_p = h_0)$.
- $\mathcal{A}_B : \{(q, \nu) \mid \nu \rightarrow \mathcal{A}_A(q)\}$. That is, the accepting states of $B$ are all states $(q, \nu)$ such that $\nu$ implies the accepting condition $\mathcal{A}_A(q)$ associated with location $q$.

To complete the definition, we present the conditions under which the $B$-transition $\langle (q_1, \nu_1); \lambda_B; (q_2, \nu_2)\rangle$ is defined to be compatible with the $A$-transition $\langle q_1; \lambda_A; q_2\rangle$. The conditions are:

1. If $\lambda_A$ is an input label of the form $a/F^1, \ldots, F^k := op$ for some $a \in \Sigma$, then $\lambda_B = a$. For all other cases, $\lambda_B = \epsilon$.
2. The annotation $\nu_2$ is the result of applying the transformation implied by the label $\lambda_A$ to the annotation $\nu_1$. Without loss of generality, we can assume that the transformation implied by $\lambda_A$ can be described by a transition relation of the form $\rho$: $F_1' = \mathcal{E}_1 \wedge \cdots \wedge F_b' = \mathcal{E}_b \ \wedge \ R_1' = t_1 \wedge \cdots \wedge R_d' = t_d$, where each $\mathcal{E}_i$ is either $F_i$ or the special symbol *new*, and each $t_j$ is a term over the $A$-vocabulary: $c, R_1, \ldots, R_b; F_1, \ldots, F_p$. To obtain the transformed annotation $\nu_1 \circ \rho$ we can, in principle, follow the following simplistic algorithm:
   - Write the combined annotation $E$ consisting of $\nu_1$ and the set of conjuncts included in $\rho$.
   - Add to $E$ all the logical consequences of the assertions, using transitivity, symmetry, and substitution of equals for equals. If $\nu_1$ is acyclic, this closure is also finite.
   - Omit from the extended $E$ all assertions referring to unprimed variables $(R_i, F_j)$ or to the special symbol *new*.
   - Un-prime all variables and function symbols. That is, replace each occurrence of $R_i'$ by $R_i$ and each occurrence of $F_j'$ by $F_j$.

   Compatibility requires that the annotation resulting from this process be logically equivalent to $\nu_2$.

For example, in Fig. 4 we present an annotated automaton corresponding to the comparison automaton of SEQ and PERM denoted by $\mathcal{C}(\text{SEQ}, \text{PERM})$.

In this annotation automaton we use the Statecharts-inspired encapsulation convention, by which an annotation that is common to several states labels a *super-state* enclosing all these states. For example, the assertion $F_1 = f_1$ is common to the four states: $(q_{222}, w_{222})$, $(q_{212}, w_{212})$, $(q_{112}, w_{112})$, and $(q_{111}, w_{111})$.

**Proposition 1.** *For every* ABA *$A$, there exists a finite annotation automaton corresponding to $A$.*

The proof of the proposition is also constructive. Starting with the initial state $q_0^B$, we incrementally add to $B$ all successors of every state $(q, \nu)$ already belonging to $B$, based on the successors of location $q$ in the ABA $A$. Since $A$ has a bounded applicability, say $k > 0$, it is not difficult to establish that all reachable annotations can only use terms which contain at most $k$ occurrences of each function register $F_i$. Consequently, all terms appearing in the annotation have nesting depth of at most $d \cdot k$, where $d$ is the number of the function registers. Consequently, the number of reachable annotations is finite, and the incremental construction of $B$ must eventually terminate. We denote by $\nu(A)$ the unique (up to logical equivalence) annotation automaton corresponding to the ABA $A$.

### 3.2 Correctness

In the following, we describe a state of a Herbrand engine $(A, \mathcal{H})$ by the record $\langle q, I \rangle$, where $q \in Q$ is an $A$-location and $I$ is a Herbrand interpretation, assigning Herbrand terms to data registers and function symbols to the function registers.

We say that the run $r : \langle q_0, I_0 \rangle, \langle q_1, I_1 \rangle, \ldots, \langle q_k, I_k \rangle$ of the Herbrand engine $(A, \mathcal{H})$ for an ABA $A$ *corresponds* to the run $r^\nu : (q_0, \nu_0), (q_1, \nu_1), \ldots, (q_k, \nu_k)$ of the annotation automaton $\nu(A)$, if for every $i = 1, \ldots, k$ and every expressible assertion $\beta$, $I_i \models \beta$ iff $\beta$ is a logical consequence of $\nu_i$.

**Claim 5.** *Let $A$ be an ABA. There exists a run $r : \langle q_0, I_0 \rangle, \langle q_1, I_1 \rangle, \ldots, \langle q_k, I_k \rangle$ of the engine $(A, \mathcal{H})$ over a discriminating program $\pi$: $\langle h_1, a_1 \rangle, \langle h_2, a_2 \rangle, \ldots \langle h_m, a_m \rangle$ iff there exists a corresponding run $r^\nu : (q_0, \nu_0), (q_1, \nu_1), \ldots, (q_k, \nu_k)$ of the annotation automaton $\nu(A)$ over the projected input word $a_1 a_2 \cdots a_m$.*

**Corollary 1.** *An ABA $A$ is universal iff its annotated version $\nu(A)$ is.*

As a consequence, two ABA's $A_1$ and $A_2$ are equivalent if and only if $\nu(\mathcal{C}(A_1, A_2))$ is universal. It is straightforward to check whether a given annotation automaton $\nu(A)$ is universal. It is sufficient to check that, from any reachable location, there exists a path labeled only by $\epsilon$ labels which leads to an accepting location. For example, the annotation automaton presented in Fig. 4 is obviously universal. We can conclude that the two automata SEQ and PERM are equivalent.

### 3.3 Implementation Details

The derivation of the transformed annotation $\nu \circ \rho$ as described above is used only for presentation purposes. In the actual implementation, we keep only the kernel of each annotation and never explicitly compute the full closure. The kernel of an annotation consists of a partition of the data registers and function registers into several equality classes. Only one representative of each class participates in additional assertions of the form $R_i = t$. Representations of the annotations are kept in a canonical form so that equality between annotations will be easy to check. The complexity of the algorithm is of the order of the number of possible annotations. For an ABA with $b$ data registers and $p$ function registers, we can compute a bound of $O(\frac{p^{p+1}}{e^p}(p + b)^{b+2p})$ on the number of different annotations.

## 4 The Case of 1-Bounded ABA's

In this section, we present a more efficient algorithm for checking equivalence between two ABA's for the case that the application bound for each of the automata is 1. This model is sufficient for most out-of-order-execution algorithms we consider because in these algorithms a function symbol which resides in a function register is applied only once before dispatching a new instruction to the same register.

As before, we start by forming the comparison automaton $C = \mathcal{C}(A_1, A_2)$. It remains to check that $C$ is universal over all discriminating programs.

We now proceed to show that there exists some *finite* structure $\mathcal{B}$ such that $(C, \mathcal{B})$ is universal iff $(C, \mathcal{H})$ is universal over all discriminating programs.

We define a $\mathcal{B}_4$-structure to be a structure $\mathcal{B}$: $(\{0,1\}, \alpha)$. The domain of such a structure is the boolean domain $\{0,1\}$, $c^\alpha = 0$, and for each function symbol $h_i \in \Xi$ , $h_i^\alpha \in \mathcal{G} = \{g_0(x,y) = 0,\ g_1(x,y) = x,\ g_2(x,y) = y,\ g_3(x,y) = 1\}$. The concrete function set $\mathcal{G}$ consists of the two constant functions $g_0$ and $g_3$, yielding the respective constants 0 and 1, and two projection functions: $g_1$ reproducing the first argument as a result, and $g_2$ reproducing the second argument as a result.

A Herbrand term $t$ is called *discriminating* if it does not contain two occurrences of the same function symbol. It is not difficult to establish that all terms occurring in runs of a comparison automaton formed out of two 1-bounded ABA's over a discriminating program are discriminating. An important property of the structure $\mathcal{B}_4$ is that it can always distinguish between two unequal discriminating terms. This is established by the following lemma.

**Lemma 1.** *For every $t_1$ and $t_2$, unequal discriminating Herbrand terms over the signature $\Xi$, there exists a $\mathcal{B}_4$-structure $\mathcal{B} = \{\{0,1\}, \alpha\}$ such that $\alpha[t_1] \neq \alpha[t_2]$.*

**Proof.** The claim is proven by induction on the size of the smaller term among $t_1$ and $t_2$. With no loss of generality, assume that $t_1$ is the bigger of the two.

For the base case, assume that the size of $t_2$ is 1. This is possible only when $t_2 = c$. Since $t_1 \neq t_2$, $t_1$ must be of the form $h(u,v)$, where $u$ and $v$ are terms. Pick an interpretation $\alpha$, such that $\alpha[h] = g_3$. Thus, we obtain

$$\alpha[t_1] = \alpha[h(u,v)] = g_3(\alpha[u], \alpha[v]) = 1 \neq 0 = \alpha[c] = \alpha[t_2].$$

For the induction step, let $t_1 = h_1(u_1, v_1)$ and $t_2 = h_2(u_2, v_2)$. We distinguish two cases. If $h_1 \neq h_2$, then the interpretation $\alpha$ such that $\alpha[h_1] = g_3$ and $\alpha[h_2] = g_0$ leads to the following equalities/inequality chain:

$$\alpha[t_1] = \alpha[h_1(u_1,v_1)] = g_3(\alpha[u_1], \alpha[v_1]) = 1 \neq 0 = g_0(\alpha[u_2], \alpha[v_2]) = \alpha[t_2].$$

In the other case, $h_1 = h_2 = h$. Since $t_1 \neq t_2$, this implies that either $u_1 \neq u_2$ or $v_1 \neq v_2$. With no loss of generality, assume that $u_1 \neq u_2$. By the induction hypothesis, there exists an interpretation $\alpha$ such that $\alpha[u_1] \neq \alpha[u_2]$. Let $\widetilde{\alpha}$ be the interpretation agreeing with $\alpha$ on the interpretation of all functions, except possibly on the interpretation of $h$ which $\widetilde{\alpha}$ interprets as $g_1$. Since $t_1$ and $t_2$ are discriminating, the function symbol $h$ does not appear in $u_1$ or $u_2$ and, therefore, $\widetilde{\alpha}$ and $\alpha$ must agree on the interpretation of $u_1$ and $u_2$. Consequently, we have the chain

$$\widetilde{\alpha}[t_1] \;=\; \widetilde{\alpha}[h(u_1, v_1)] \;=\; g_1(\widetilde{\alpha}[u_1], \widetilde{\alpha}[v_1]) \;=\; \widetilde{\alpha}[u_1] \;=\; \alpha[u_1] \;\neq \\ \alpha[u_2] \;=\; \widetilde{\alpha}[u_2] \;=\; g_1(\widetilde{\alpha}[u_2], \widetilde{\alpha}[v_2]) \;=\; \widetilde{\alpha}[h(u_2, v_2)] \;=\; \widetilde{\alpha}[t_2]$$

▪

The following claim states that in order to test for universality of the automaton $C$ it is sufficient to consider the engine $(C, \mathcal{B}_4)$ for a specific $\mathcal{B}_4$ structure. Without loss of generality, assume that the signature $\Xi$ contains the four function symbols $\{f_0, f_1, f_2, f_3\}$ and that the mapping associated with $\mathcal{B}_4$ maps each $f_i$ to $g_i$, for $i = 0, \ldots, 3$. We say that a program $\pi$ is a $\Xi_4$*-program* if the only function symbols appearing in $\pi$ are $f_0, \ldots, f_3$.

**Claim 6.** *Let $C$ be a comparison automaton formed out of two 1-bounded execution automata. The automaton $C$ is universal iff the engine $(C, \mathcal{B}_4)$ is universal over all $\Xi_4$-programs.*

**Proof.** Assume to the contrary, that $(C, \mathcal{B}_4)$ is universal over all $\Xi_4$-programs but $C$ is not universal.

By Claim 4, there exists some discriminating program $\pi$ and a run $r$ of $(C, \mathcal{H})$ over $\pi$ which cannot be extended to an accepting run without reading additional inputs. Let $q$ be the last location reached by $r$. Without loss of generality, we can assume that $r$ cannot be extended beyond $q$ by even a single additional input-free step.

The fact that $r$ is not accepting and cannot be extended beyond $q$ implies that one of the conjuncts in $\mathcal{A}(q)$ is not satisfied by the last state of $r$. This conjunct is of the form $R_i = R_j$ or $R_i = t$ and the Herbrand values of the left and right hand side of the equality are unequal in the last state of $r$. Without loss of generality, let us assume that the violated conjunct is $R_k = R_j$. Let us denote these two unequal Herbrand terms by $t^1$ and $t^2$.

By Lemma 1, there exists a mapping $\beta$ from $\Xi$ to $\mathcal{G}$, such that $\beta(t^1) \neq \beta(t^2)$. We form now a new program $\pi_\beta$ in the following way: every function symbol $h$ occurring in $\pi$ is replaced by $f_i$, where $i \in \{0..3\}$ is the index of the $\mathcal{G}$-function $g_i$ such that $\beta(h) = g_i$. Let $\alpha$ be the mapping of the canonic structure $\mathcal{B}_4$, which, in particular, maps each $f_i$ to $g_i$, for $i = 0, \ldots, 3$.

It can be shown that $\alpha(r)$ is a run of $(C, \mathcal{B}_4)$ over the new program $\pi_\beta$ and produces at the last state $\alpha(t^1) = \beta(t^1)$ and $\alpha(t^2) = \beta(t^2)$ as the values of the data registers $R_k$ and $R_j$, respectively. Since $\beta(t^1) \neq \beta(t^2)$ we conclude that $\alpha(r)$ is a run of $(C, \mathcal{B}_4)$ over $\pi_4$ which cannot be extended to an accepting run. Therefore, $(C, \mathcal{B}_4)$ is not universal over all $\Xi_4$-programs. ▪

The combination of Claim 4 and Claim 6 reduces the problem of checking the universality of $C$ to checking the universality of $(C, \mathcal{B}_4)$ over all $\Xi_4$-programs. Let us study the behavior of the engine $(C, \mathcal{B}_4)$ over $\Xi_4$-programs. A first observation is that, since $\Xi_4$ has only four function symbols $(f_0, \ldots, f_3)$, it has only finitely many input instructions. Consequently, we can view every $\Xi_4$-program as a finite word over the finite alphabet $4 \times \Sigma$.

Next, let us consider the states of $(C, \mathcal{B}_4)$ when executing a $\Xi_4$-program. Following the general definition, such a state has the form $s = \langle q; t_1, \ldots, t_b; f_1, \ldots, f_d \rangle$, where each $t_i$ is a value over the domain of $\mathcal{B}_4$, namely a boolean value, and every $f_j$ can assume one of the four values $f_0, \ldots, f_3$. Consequently, $(C, \mathcal{B}_4)$ can assume only finitely many different states when executing an $\Xi_4$-program.

Following these two observations, it is straightforward to transform the engine $(C, \mathcal{B}_4)$ into a conventional finite-state automaton $C_{\mathcal{B}_4}$ which is universal in the automata-theoretic sense iff $(C, \mathcal{B}_4)$ is universal over all $\Xi_4$-programs.

This leads to the following algorithm for checking the equivalence of the 1-bounded automata $A_1$ and $A_2$:

1. Construct the comparison automaton $C = \mathcal{C}(A_1, A_2)$.
2. Construct $C_{\mathcal{B}_4}$ the finite-state automaton representing the computations of $C$ over $\mathcal{B}_4$-programs.
3. Apply automata-theoretic methods to check that $C_{\mathcal{B}_4}$ is universal, in the sense used in this paper. That is, check that every run of the (possibly non-deterministic) automaton $C_{\mathcal{B}_4}$ over any input word can be extended into an accepting run.

This algorithm has been implemented within the symbolic model checker TLV ([14]) and has been used to successfully check the equivalence of Herbrand automata SEQ and PERM for the cases of 2 and 3 data registers, and the correctness of Tomasulo algorithm for out-of-order execution [7], for the case of 2 registers and 2 reservation stations. The state space of the system is of size less than $2^{28}$. We are currently trying to prove it for larger configurations of the Tomasulo algorithm.

## 5 Discussion

Obviously, the methods proposed here are based on the observation that abstracting a data-dependent verification problem by considering the more general case of *uninterpreted functions* is often more tractable than dealing with the specifics of particular data domains such as floating point numbers and their operations. The idea of using uninterpreted functions in verification of hardware designs is becoming increasingly more prevalent and many recently proposed methods are based on this approach, e.g., [3], [8].

One of the advantages of the methods proposed here is that they require no refinement mapping and check for a direct equivalence between the specification and implementation.

We can view the methods proposed here as an adaptation of the concepts of program schemata [11] and the methods developed for their analysis. In particular, the approaches developed in this paper utilize some of the ideas proposed first in [12], where a bound on the amount by which one machine may lag behind the other ensures tractability due to the bounded memory one has to maintain.

## References


1. G. Barrett and A. McIsaac. Model-checking in a microprocessor design project. In O. Grumberg, editor, *Proc. $9^{th}$ Intl. Conference on Computer Aided Verification (CAV'97)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1997. to appear.
2. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.
3. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. $6^{th}$ Intl. Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1994.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, 1992.
5. Y.A. Chen, E.M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J.OLeary, and X. Zhao. Verification of all circuits in a floating point unit using word-level modelchecking. In M. Srivas and A. Camilleri, editors, *Proc. $1^{st}$ Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lect. Notes in Comp. Sci.*, pages 1–18. Springer-Verlag, 1996.
6. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.
7. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, California, 2nd edition, 1996.
8. R. Hojati, A. Isles, D. Kirkpatrick, and R.K. Brayton. Verification using uninterpreted functions and finite instantiations. In *Proc. $1^{st}$ Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 218 – 232, 1996.
9. R.B. Jones, D.L. Dill, and J.R.Burch. Efficient validity checking for processor verification. In *Intl. Conf. on Computer-Aided Design.* IEEE, 1995.
10. K. Keutzer. The need for formal methods for integrated circuit design. In M. Srivas and A. Camilleri, editors, *Proc. $1^{st}$ Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lect. Notes in Comp. Sci.*, pages 1–18. Springer-Verlag, 1996.
11. D.C. Luckham, D.M.R. Park, and M.S. Paterson. On formalized computer programs. *J. Comp. Sys. Sci.*, 4(3):220–249, 1970.
12. Z. Manna and A. Pnueli. Synchronous schemes and their decision problems. In *Proc. 7th ACM Symp. Princ. of Prog. Lang.*, pages 62–67, 1980.
13. S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. In R. Kumar and T. Kropf, editors, *Proceedings of the Second Conference on Theorem Provers in Circuit Design*, pages 167–188. FZI Publication, Universität Karlsruhe, 1994. Preliminary Version.
14. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *Proc. $8^{th}$ Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci., pages 184–195. Springer-Verlag, 1996.
15. M. Srivas and S. Miller. Applying formal verification to the aamp5 microprocessor: A case study in the industrial use of formal methods. *J. on Formal Methods in System Design*, 8:153–188, 1996.

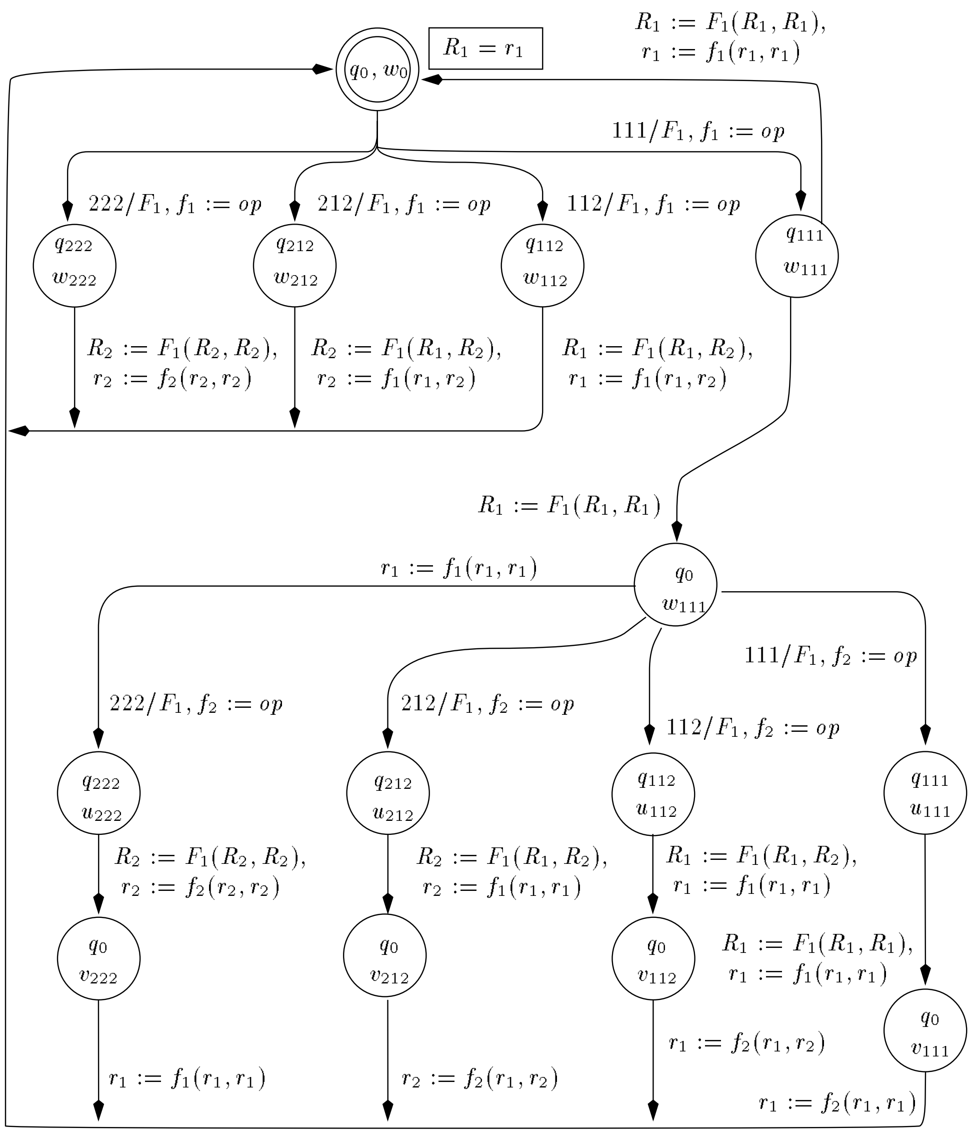


**Fig. 3.** Comparison Automaton of SEQ and PERM.

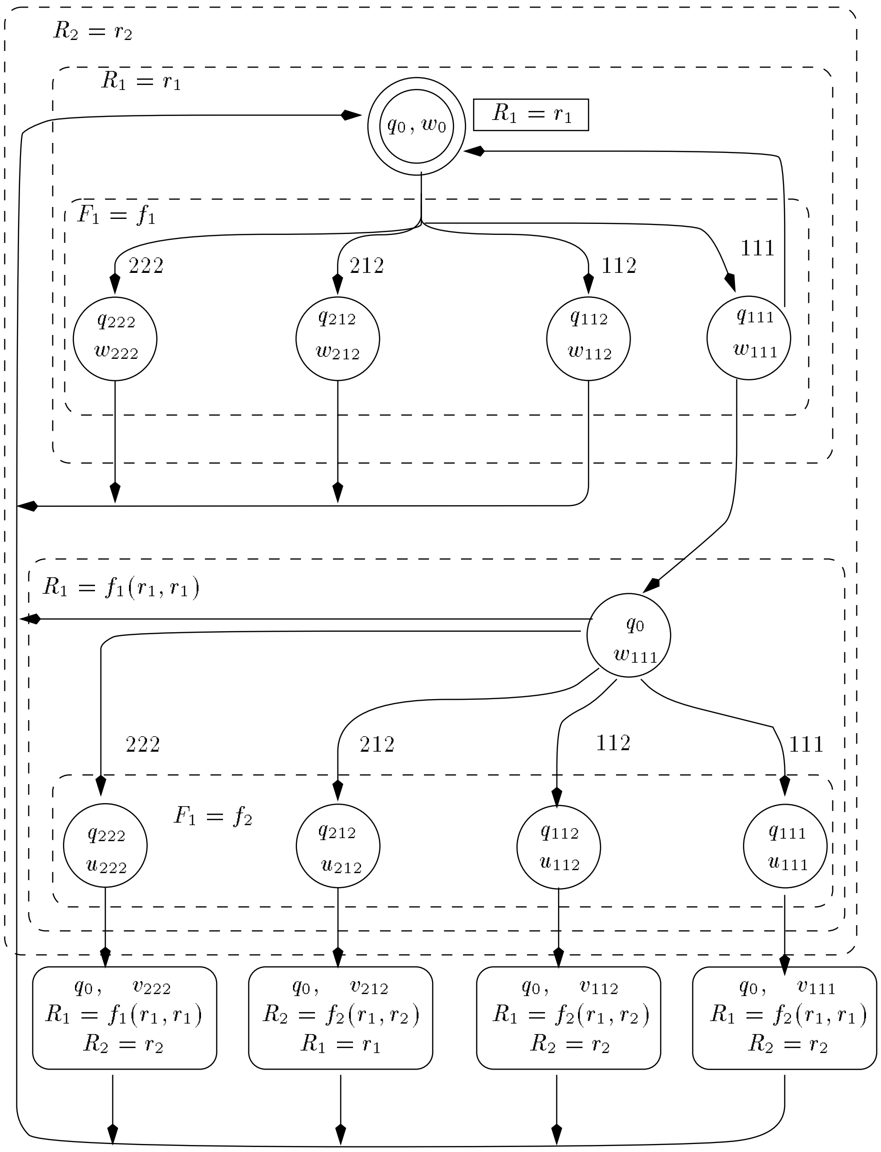


**Fig. 4.** The Annotated Automaton $\nu(\mathcal{C}(\textsc{seq}, \textsc{perm}))$.