A Framework for Microprocessor Correctness Statements

Mark D. Aagaard[1], Byron Cook[2], Nancy A. Day[3], and Robert B. Jones[4]

[1] Electrical and Computer Engr., University of Waterloo, Waterloo, ON, Canada
markaa@swen.uwaterloo.ca
[2] Prover Technology, Portland, OR, USA
byron@prover.com
[3] Computer Science, University of Waterloo
nday@cs.uwaterloo.ca
[4] Strategic CAD Labs, Intel Corporation, Hillsboro, OR, USA
rjones@ichips.intel.com

Abstract Most verifications of out-of-order microprocessors compare state-machine-based implementations and specifications, where the specification is based on the instruction-set architecture. The different efforts use a variety of correctness statements, implementations, and verification approaches. We present a framework for classifying correctness statements about safety that is independent of implementation representation and verification approach. We characterize the relationships between the different statements and illustrate how existing and classical approaches fit within this framework.

1 Introduction

The increased parallelism provided by *out-of-order execution* in microprocessors has made correctness statements for verification complicated, varied, and even controversial. We studied published verifications of out-of-order microprocessors and discovered a wide variety of correctness statements, verification techniques, and processor implementations. Some correctness statements initially appear to be similar, such as the ones based on Burch-Dill style flushing [BD94], but differences emerge after close examination. Other statements are difficult to compare at first, but later reveal similarities. The goal of this work is to provide a foundation for clarifying the meaning of individual correctness statements; precisely comparing different statements; and analyzing the interaction between processor features, verification strategy, and correctness statements.

Most recent verification efforts verify a state-machine-based microarchitectural implementation against a state-machine-based instruction-set architecture. The verification efforts focus on safety; liveness is usually dealt with as a secondary concern. In keeping with these trends, we focus on the verification of safety between microarchitectural implementations and instruction-set architectures. We include deterministic and non-deterministic state machines with finite or infinite state spaces. We do not yet include specifications that are collections of properties, e.g. [BB94,McM98,PJB99].

The result of our investigation and analysis is a framework that precisely describes and classifies correctness statements about safety between state machines. It allows correctness statements to be analyzed independent of verification techniques and microarchitectural features. In this paper, we introduce the framework, present its mathematical basis, and describe how existing out-of-order microprocessor correctness statements fit within the framework.

2 Modeling with State Machines

We assume that both the specification and implementation have program memories as part of their state. Therefore, our state machines do not take instructions as inputs. Approaches that take instructions as inputs in their correctness statements (e.g. [BD94,JSD98,BBCZ98]) can be augmented with program memories that produce the input trace. Interrupts can also be treated as part of the state space by adding appropriate control circuitry to read the interrupt input trace from a store. We assume that state machines generate infinite traces, where "termination" of a program is denoted by repeating the final state of the program. Definition 1 shows the formalism we use to describe state machines.

Definition 1 (State machines). A state machine M is a triple (Q, Q°, N) where:
- Q is the set of possible state values and is a Cartesian product of internal (hidden) state components and externally-visible state components.
  Q_{e} is the set of possible external state values.
  $\mathit{\Pi}_{\mathrm{e}} : Q \to Q_{\mathrm{e}}$ is the corresponding projection function.
  $q_1 \overset{\mathit{\Pi}}{=} q_2$ says that q_1 and q_2 have equivalent external state: $\mathit{\Pi}_{\mathrm{e}}(q_1) = \mathit{\Pi}_{\mathrm{e}}(q_2)$.
- $Q^{\circ} \subseteq Q$ is the set of initial states.
- $N \subseteq Q \times Q$ is the next-state relation.
  $N^k(q, q')$ means q' is reachable from q in k steps of N.
  When N is a function, we write it as n.

The components of a state machine M will be subscripted with "s" for specification and "i" for implementation. We assume a machine can always make a transition, i.e. $\forall\, q \in Q.\, \exists\, q' \in Q.\ N(q, q')$. We allow machines to *self-loop*, that is transition from a given state back to itself.

In verification, the state space of the implementation often needs to be limited to reachable states, or an over-approximation of reachable states. This challenging task is done by finding and proving invariants. Invariants are treated with varying degrees of emphasis in the literature. In our framework we consider the invariants to be encoded in Q, the set of states for the machine.

3 Correctness Statements

A well-established definition of correctness is that of *trace containment*: every trace of external observations generated by the implementation can also be generated by the specification. A disadvantage of trace containment is that verifying it can require information about an entire trace. Another traditional definition of correctness is *simulation*: if an implementation state is externally equal to a specification state, then executing one instruction in both the implementation and specification results in states that are externally equal. Simulation is usually easier to verify than trace containment, because simulation refers to individual transitions, rather than entire traces. Formal verification of sequential microprocessors has generally been done using simulation-style correctness statements. Similar correctness statements are also used in other domains such as cache-coherence protocols (e.g. [PD96,SA97,NG98]).

Pipelining and other optimizations increase the gap between the behavior of the implementation and the specification, thus making it more difficult to consider only one step within the implementation and specification traces. Pipelined machines begin executing new instructions before previous ones retire. Machines with out-of-order retirement retire instructions in a different order than the specification. A superscalar machine may externally appear to do nothing for a number of steps and then, in a single step, update the register file with the results of several instructions.

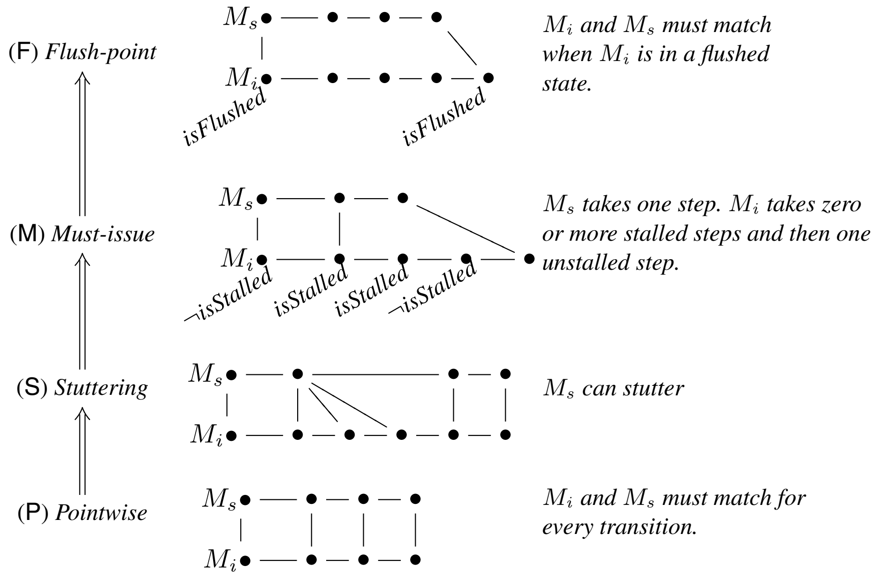
To describe how out-of-order verifications use simulation-style correctness statements, we separate the notions of 1) how to align the implementation trace against the specification trace to determine which states should match, and 2) what it means for an implementation state to successfully match a specification state.

When verifying non-pipelined machines, the traces can be aligned at every transition and two states match if the externally-visible state components are equal. To verify pipelined machines, the alignment often needs to be at looser intervals than every transition, or external equivalence needs to be replaced by a looser relationship. With out-of-order microprocessors, the notions of alignment and matching are necessarily even more complicated. A common alignment technique is to check the implementation when it is in a *flushed* state (i.e. no in-flight instructions). A common matching relationship is Burch-Dill style flushing [BD94], which uses an abstraction function to retire all in-flight instructions in the implementation and project the externally-visible state, and then checks for equality with the specification state.

Our framework uses four parameters to characterize a correctness statement: alignment, match, implementation execution, and specification execution. *Alignment* is the method used to align the executions of the implementation and specification (Sect. 3.1). *Match* is the relation established between the aligned implementation and specification states (Sect. 3.2). *Implementation execution* and *specification execution* describe the type of state machines used (Sect. 3.3). Section 4 shows how the correctness statements of existing work fit within our framework.

3.1 Alignment

Alignment describes which states in the execution traces are tested for matching. Figure 1 illustrates the four kinds of alignment that we have found used in out-of-order microprocessor verification. *Pointwise alignment* (P) is the classic commuting diagram, which compares every transition. *Stuttering alignment* (S) allows the specification to *stutter*, i.e. two or more consecutive implementation states can match the same specification state. In *must-issue alignment* (M), the specification takes a single step, and the implementation takes as many steps as are necessary to reach an unstalled state, and then issue an instruction. A predicate *isStalled* indicates when the implementation cannot take a “productive” step, and is generally defined to be true when the implementation cannot issue an instruction. Finally, *flush-point alignment* (F) says that if there is a trace between flushed implementation states, then there must exist a trace in the specification between any pair of states that match the flushed implementation states. A predicate *isFlushed* indicates flushed implementation states. Instruction-set architectures execute one instruction per step; therefore all of their states are flushed.


(F) Flush-point
M_s
M_i
isFlushed
isFlushed
M_i and M_s must match when M_i is in a flushed state.
(M) Must-issue
M_s
M_i
¬isStalled
isStalled
isStalled
¬isStalled
M_s takes one step. M_i takes zero or more stalled steps and then one unstalled step.
(S) Stuttering
M_s
M_i
M_s can stutter
(P) Pointwise
M_s
M_i
M_i and M_s must match for every transition.

In each diagram, the horizontal lines between states are the specification and implementation traces. The vertical lines between states show the where the implementation state must match the specification state.

Fig. 1. Options and total order for the alignment parameter

We place the four kinds of alignment in a total order as illustrated by the arrows in Fig. 1. This order is based on generality where alignments higher in the order are weaker. For example, stuttering correctness implies flush-point for any instance of the predicate *isFlushed*.

3.2 Match

Instantiations for the match parameter are relations $\mathcal{R}$ over an implementation state q_i and specification state q_s that mean "q_i is a correct representation of q_s". Figure 2 shows the matches that we found used in out-of-order microprocessor verification. The arrows show the total order, where definitions lower in the order are instances of higher values.

A *general match* (G) is any relation between implementation and specification states. The *abstraction match* (A) uses a function (*abs*) to map an implementation state to a point that is externally equivalent to the specification state. The *equality match* (E) requires that the implementation and specification states be externally equivalent. The tightest match is the *refinement map* (R), which is an abstraction function that preserves the externally-visible part of the implementation state. Refinement differs from equality because the refinement map is a function, so each implementation state matches exactly one specification state. The literature overloads words such as "refinement" and "abstraction". The mathematics in Fig. 2 give the precise definitions that we use.

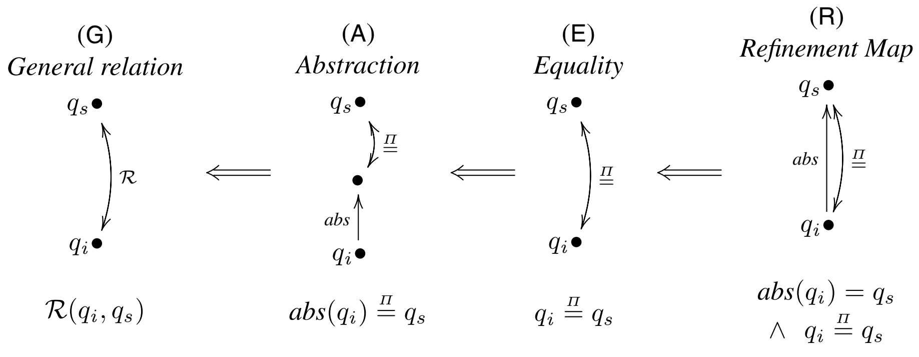


Fig. 2. Options and total order for the match parameter

If the specification does not have any internal state (i.e. all of the state components are externally visible), then equality and refinement both reduce to $\Pi_e(q_i) = q_s$. Because refinement is a tighter match than equality, we will call such cases refinement.

In addition to the instances listed here, other options are possible. For example, Pnueli *et al.* [DP97,PA98,AP99,AP00] have a matching relation that uses both concretization and abstraction functions. Two states match if concretizing the specification state produces the same result as abstracting the implementation state. In their examples, their concretization functions are identity or projection, so their match specializes to abstraction in our framework.

3.3 Execution

The third and fourth parameters of the framework are the methods for describing the traces of the implementation and specification. In the literature we find both *deterministic* (D) and *non-deterministic* (N) implementations and specifications. In a deterministic machine, the transition relation is instantiated as: $N(q, q') \equiv q' = n(q)$. Implementations are often modeled with non-determinism because of scheduling circuitry. On the other hand, most instruction-set architectures are deterministic, so most specification machines are deterministic. Exceptions include specifications with imprecise exceptions or external interrupts. For our purposes, we consider deterministic machines as instances of non-deterministic machines in the total order for the execution parameters.

3.4 Correctness Space

By choosing different values of the parameters, we arrive at a variety of correctness statements. We use four-letter acronyms to describe the values of the parameters:

<alignment> <match> <impl. execution> <spec. execution>

For example, "PADD" denotes point-wise alignment (P), abstraction match (A), and deterministic implementation (D) and specification (D).

Each parameter has a total order on its instantiations. Together, these total orders induce a partial order over correctness statements, which serves to map out the space of correctness statements for microprocessor implementations (Fig. 3). The partial order is based on the generality of the correctness statements. For example, FGNN (at the top of the partial order) is more general than PADD because pointwise alignment implies flush-point alignment; an abstraction function is an instance of a general relation; and deterministic machines are instances of non-deterministic ones. Correctness statements lower in the order are less general in that they apply to fewer systems. We do not advocate any points in the correctness space over others. The classification serves to highlight the differences and similarities among approaches.

3.5 Mathematical Formulation

In this section we describe the mathematical formulations of correctness statements in the framework. We use $M_i \preccurlyeq_{\mathcal{R}} M_s$ to mean "M_i is correct with respect to M_s via the relation $\mathcal{R}$". All of the correctness statements have the general form of Definition 2. The base clauses remain largely unchanged from the one shown in Definition 2, so in the remainder of the paper, we will discuss only the induction clauses.

Definition 2 (General form of correctness statement).

$$(Q_i, Q_i^\circ, N_i) \preccurlyeq_{\mathcal{R}} (Q_s, Q_s^\circ, N_s) \equiv \left[\wedge \begin{array}{l} \forall\, q_i^\circ \in Q_i^\circ.\ \exists\, q_s^\circ \in Q_s^\circ.\ \mathcal{R}(q_i^\circ, q_s^\circ) \\ \langle \textit{inductive clause} \rangle \end{array} \right]$$

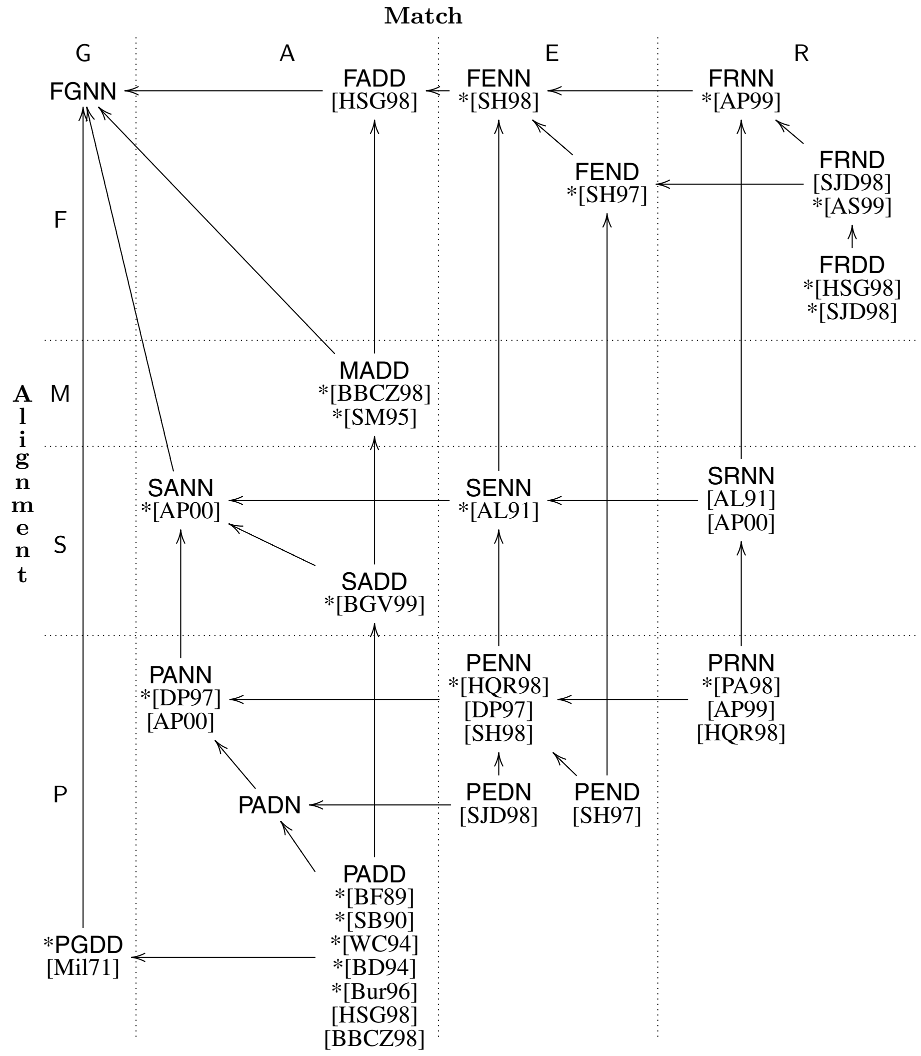
The alignment parameter determines the form of the correctness statement. We show the correctness statements for the various values of the alignment parameter together with the most-general values for the other parameters (i.e. non-deterministic machines with a general relation match).

The most general combination in the correctness space is flush-point alignment with a general match and non-deterministic machines (FGNN, Definition 3). It says that if the implementation is in a flushed state q_i and can transition through some number of steps k to another flushed state q_i', then all specification states q_s that $\mathcal{R}$ says match q_i must transition through some number of steps j to some state q_s' that matches q_i'.

Definition 3 (Flush-point induction clause: FGNN).

$$\begin{array}{l} \forall\, q_i, q_i' \in Q_i.\ \forall\, q_s \in Q_s.\ \exists\, q_s' \in Q_s. \\ \quad \left[\wedge \begin{array}{l} \textit{isFlushed}(q_i)\ \wedge\ \exists\, k.\ N_i^k(q_i, q_i')\ \wedge\ \textit{isFlushed}(q_i') \\ \mathcal{R}(q_i, q_s) \end{array} \right] \Longrightarrow \left[\wedge \begin{array}{l} \exists\, j.\ N_s^j(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{array} \right] \end{array}$$

The most general case of must-issue alignment (Definition 4) is MGNN. In must-issue alignment, the specification takes one step and the implementation takes as many steps k as are necessary to become unstalled; it then takes one unstalled transition from q_i^k to q_i^{k+1}.



<alignment>	*<match>*	*<impl. execution>*	*<spec. execution>*
(F) Flush-point	(G) General	(N) Non-deterministic	(N) Non-deterministic
(M) Must-issue	(A) Abstraction	(D) Deterministic	(D) Deterministic
(S) Stuttering	(E) Equality		
(P) Pointwise	(R) Refinement Map		

Each point is annotated with citations for the works that use the particular correctness statements. Citations prefixed with ∗ denote top-level correctness statements; others are used as intermediate correctness statements during the proofs. Section 4 provides further explanation.

Fig. 3. Space of correctness statements

Definition 4 (Must-issue induction clause: MGNN**).**

$$\forall\, q_i^0, q_i^1, \ldots, q_i^{k+1} \in Q_i.\ \forall\, q_s \in Q_s.\ \exists\, q_s' \in Q_s.$$
$$\begin{bmatrix} \wedge\ \forall\, j < k.\ N_i(q_i^j, q_i^{j+1}) \ \wedge\ \mathit{isStalled}(q_i^j) \\ \wedge\ N_i(q_i^k, q_i^{k+1}) \ \wedge\ \neg\mathit{isStalled}(q_i^k) \\ \mathcal{R}(q_i^0, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\ N_s(q_s, q_s') \\ \mathcal{R}(q_i^{k+1}, q_s') \end{bmatrix}$$

Stuttering alignment results in a simpler correctness statement (Definition 5) because it considers only one step of execution and requires no special predicates. The specification is allowed to *stutter*, e.g. consecutive implementation states may align with the same specification state.

Definition 5 (Stuttering induction clause: SGNN**).**

$$\forall\, q_i, q_i' \in Q_i.\ \forall\, q_s \in Q_s.\ \exists\, q_s' \in Q_s.$$
$$\begin{bmatrix} \wedge\ N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\ (N_s(q_s, q_s') \ \vee\ (q_s' = q_s)) \\ \mathcal{R}(q_i', q_s') \end{bmatrix}$$

Pointwise alignment (Definition 6) drops the stuttering disjunct from Definition 5.

Definition 6 (Pointwise Induction Clause: PGNN**).**

$$\forall\, q_i, q_i' \in Q_i.\ \forall\, q_s \in Q_s.\ \exists\, q_s' \in Q_s.\quad \begin{bmatrix} \wedge\ N_i(q_i, q_i') \\ \mathcal{R}(q_i, q_s) \end{bmatrix} \implies \begin{bmatrix} \wedge\ N_s(q_s, q_s') \\ \mathcal{R}(q_i', q_s') \end{bmatrix}$$

Section 4 describes other correctness statements as points in the correctness space by optionally instantiating $\mathcal{R}$ and the next-state relations in each of the above four definitions. Instantiating next-state relations with functions removes the need for some quantified variables.

3.6 Limitations

As with most formal verification approaches, the correctness framework presented in this paper does not exclude certain "pathologically bad" matching relations. A matching relation that includes all implementation and specification state pairings will result in a vacuously true correctness statement, as any pair of consecutive implementation states can be related to any pair of consecutive specification states. Another vacuous statement results from using stuttering alignment with an abstraction function that maps all implementation states to the same specification state.

4 Literature Survey

In this section, we show how a variety of correctness statements for out-of-order microprocessors are points in the correctness space created by the framework. While the phrase "we use the correctness statement of Burch and Dill [BD94]" appears in many papers, detailed examinations reveal that this is something of an approximation. For conciseness, we discuss only the inductive clauses of the correctness statements.

4.1 Historical Perspective

Milner's [Mil71] work in software verification led to a definition of simulation that is *pointwise alignment* of a general relation between a deterministic implementation and a deterministic specification (PGDD). Milner does not include a base clause. PGDD is derived from Definition 6 (pointwise alignment) by substituting next-state functions (n_s and n_i) for the next-state relations (N_s and N_i).

Definition 7 (Milner's *simulation*: PGDD).

$$\forall\, q_i \in Q_i.\, \forall\, q_s \in Q_s.\;\; \mathcal{R}(q_i, q) \implies \mathcal{R}(n_i(q_i),\; n_s(q))$$

Abadi and Lamport [AL91] define *refinement maps*, which in our parlance is *stuttering refinement* between non-deterministic machines (SRNN). They use refinement as a verification strategy to prove SENN (stuttering equivalence), which they call *implements*. Both SRNN and SENN are derived from Definition 5 (SGNN). For SRNN, refinement ($abs(q_i) = q_s \,\wedge\, q_i \overset{\Pi}{=} q_s$) is substituted for the match $\mathcal{R}$, which results in Definition 8.

Definition 8 (Abadi and Lamport's *refines*: SRNN).

$$\forall\, q_i, q_i' \in Q_i.\quad \begin{bmatrix} N_i(q_i, q_i') \\ \wedge\; q_i \overset{\Pi}{=} abs(q_i) \end{bmatrix} \implies \begin{bmatrix} N_s(abs(q_i'), abs(q_i)) \,\vee\, abs(q_i') = abs(q_i) \\ \wedge\; q_i' \overset{\Pi}{=} abs(q_i') \end{bmatrix}$$

Their main result is that if SENN holds, then it is possible to construct an intermediate model from an implementation using history and prophecy variables such that the intermediate model will satisfy SRNN with the specification.

Several verifications of scalar pipelines use correctness statements that are relevant to this paper. Bose and Fischer [BF89] used PADD in the verification of a pipelined stack. In the first published verification of pipelined microprocessor, Srivas and Bickford [SB90], and Windley and Coe [WC94] verified PADD between their implementations and specifications. Srivas and Miller [SM95] proved MADD between a pipelined microprocessor at the instruction and micro-instruction levels of abstraction. All of the abstraction functions in these efforts were manually constructed. The complexity of superscalar microprocessors has led to efforts to automatically (Sect. 4.2) or systematically (Sect. 4.4) create abstraction functions.

4.2 Flushing

Burch and Dill [BD94] use PADD (pointwise abstraction) as their correctness statement. They observed that for in-order pipelines, it is possible to construct an abstraction function by forcing the implementation to issue a stream of *bubbles* that *flush* all of the in-flight instructions out to retirement. In this case, $abs \equiv flush$. Notions similar to flushing are also used for correctness criteria in other domains, such as cache-coherence protocol verification (e.g. [PD96,SA97,NG98]).

To flush a microprocessor, the implementation must be forced to issue a bubble. An implementation $\hat{n_i}$ must be constructed that has the necessary input and control circuitry for flushing. The function *flush* and the next-state function n_i used in the correctness statement are defined as shown below, where k is the number of bubbles used to flush the pipeline, i.e. the maximum latency of the pipeline:

$$\mathit{flush}(q_i) \equiv \hat{n_i}^k(q_i, \texttt{True})$$
$$n_i(q_i) \equiv \hat{n_i}(q_i, \texttt{False})$$

Stalls complicate the flushing abstraction, because the implementation cannot fetch an instruction to take a "productive" step. The framework provides us with several options for handling this. The first two are instances of PADD. The other three options; which use PADN, SADD, and MADD; are slightly more general than PADD.

Explicit Stalling of Specification. The first option is to construct a revised specification ($\hat{n_s}$) that takes a stall input s to indicate whether to self-loop or not:

$$\hat{n_s}(q_s, s) \;\equiv\; \texttt{if}\; s \;\texttt{then}\; q_s \;\texttt{else}\; n_s(q_s)$$

The correctness statement provides this stall information to the specification (Definition 9) [BD94,Bur96,HSG98,JSD98].

Definition 9 (Burch and Dill flushing: an instance of PADD).

$$\forall q_i \in Q_i.\; \hat{n_s}(\mathit{flush}(q_i), \mathit{isStalled}(q_i)) = \mathit{flush}(n_i(q_i))$$

This definition is derived from Definition 6 by replacing $\mathcal{R}$ with a match abstraction that uses *flush* as the abstraction function and using next-state functions rather than relations. Burch generalized this idea to superscalar microprocessors [Bur96], by generalizing the *isStalled* predicate to be a function that indicates to the specification how many instructions the implementation issued.

Modified Pointwise. The second option is to use the *isStalled* predicate directly in the PADD correctness statement rather than modifying the specification:

$$\bigwedge \begin{array}{rcr} \mathit{isStalled}(q_i) & \Longrightarrow & \mathit{flush}(q_i) = \mathit{flush}(n_i(q_i)) \\ \neg\mathit{isStalled}(q_i) & \Longrightarrow & n_s(\mathit{flush}(q_i)) = \mathit{flush}(n_i(q_i)) \end{array}$$

Non-deterministic Specification. The third option is to use a non-deterministic specification that includes a state element that says whether the specification self-loops (PADN). The abstraction function maps the value of the implementation predicate *isStalled* to the new specification state element.

Stuttering. The fourth option is to use stuttering abstraction with flushing as the abstraction function. This method avoids the need to alter the specification. However, this permits the specification to stutter at anytime, not just when the implementation stalls. This is the approach taken by Bryant *et al.* [BGV99] and by Arvind and Shen [AS99]. Bryant *et al.* use SADD to verify a superscalar machine. Stuttering alignment for this kind of implementation allows the specification to issue 0–k instructions, where k is the maximum number of instructions the implementation can issue in one cycle. Arvind and Shen prove FRND for an out-of-order implementation with in-order retirement. Both the specification and implementation are represented as term rewriting systems. They omit the details of their proof, but it appears to rely on stuttering abstraction (SAND) between the implementation and specification.

Must-Issue. The fifth option is to use must-issue alignment with flushing as an abstraction function (Definition 10). Berezin *et al.* [BBCZ98] prove must-issue abstraction (MADD) for a processor with out-of-order retirement. The model is deterministic but some of the scheduling is left underspecified. They introduce intermediate models of the implementation and specification that are optimized for model-checking efficiency. They prove MADD between the intermediate implementation and intermediate specification. They relate this result to the real specification and implementation by proving pointwise abstraction (PADD) between each of the intermediate models and its respective concrete counterpart.

Definition 10 (Must-issue abstraction with flushing: an instance of MADD).

$$\begin{aligned}&\forall q_i \in Q_i.\, \forall\, k.\;\; \forall\, j < k.\;\; \mathit{isStalled}(n_i^j(q_i)) \;\wedge\; \neg \mathit{isStalled}(n_i^k(q_i))\\ &\qquad \Longrightarrow\;\; n_s(\mathit{flush}(q_i)) \overset{\Pi}{=} \mathit{flush}(n_i^{k+1}(q_i))\end{aligned}$$

When the abstraction function used in MADD is *flush*, then an implementation step from a stalled state should result in a state that matches the same specification state, i.e. $\mathit{isStalled}(q_i) \implies \mathit{abs}(q_i) = \mathit{abs}(n_i(q_i))$ which is equivalent to SADD.

4.3 Trace Tables

Sawada and Hunt [SH97] verified that a non-deterministic processor with out-of-order retirement satisfies *flush-point equality* with a deterministic specification (FEND, Definition 11). FEND results from substituting the equality match ($q_i \overset{\Pi}{=} q_s$) for $\mathcal{R}$ and a next-state function n_s for the next-state relation in Definition 3 (FGNN).

Definition 11 (Flush-point equality: FEND).

$$\begin{aligned}&\forall\, q_i, q_i' \in Q_i.\, \forall\, q_s \in Q_s.\\ &\left[\wedge \begin{array}{l}\mathit{isFlushed}(q_i) \;\wedge\; \exists\, k.\; N_i^k(q_i, q_i') \;\wedge\; \mathit{isFlushed}(q_i')\\ q_i \overset{\Pi}{=} q_s\end{array}\right] \Longrightarrow \left[\exists\, j.\; q_i' \overset{\Pi}{=} n_s^j(q_s)\right]\end{aligned}$$

In later work, they enhanced their implementation to support in-order retirement, external interrupts, and precise exceptions [SH98,SH99]. The inclusion of interrupts led them to add non-determinism to their specification, to account for the problem of predicting how many instructions the implementation will have completed when an interrupt occurs. They kept flush-point equality as their alignment and match criteria, making their correctness statement FENN.

Throughout this work, their verification strategy was to build an intermediate model with history variables. The intermediate model contains an unbounded table, called a MAETT, with one entry for each issued instruction. In their first work [SH97], they prove pointwise equality (PEND) between the implementation and intermediate model and FEND (flush-point equality) between the intermediate model and specification, which together imply FEND. Similarly for their second model, they prove PENN and FENN respectively to conclude FENN.

4.4 Completion Functions

Hosabettu, Srivas, and Gopalakrishnan [HSG98,HSG99] prove that a deterministic out-of-order implementation satisfies *flush-point refinement* with a deterministic specification where the match is projection (Definition 12).

Definition 12 (Flush-point refinement with projection: an instance of FRDD).
$\forall q_i \in Q_i.\ \forall k.\ \textit{isFlushed}(q_i) \land \textit{isFlushed}(n_i^k(q_i)) \implies \exists j.\ n_s^j(\Pi_{\mathrm{e}}(q_i)) = \Pi_{\mathrm{e}}(n_i^k(q_i))$

Because their verification is completely within a theorem prover, they are able to use underspecified next-state functions (rather than relations) for their scheduler. They prove FRDD in three steps. They prove *pointwise abstraction* (PADD) and then apply induction to prove *flush-point abstraction* (FADD). They go from FADD to FRDD by proving that the abstraction of a flushed state is equivalent to projection. The abstraction is constructed with *completion functions*. Completion functions describe the effect of the completion of each in-flight instruction on the observable state, and are composed in program order. Hosabettu *et al.* [HGS00] also use the same correctness statement to verify an implementation with speculative execution and precise exceptions.

4.5 Incremental Flushing

Skakkebæk *et al.* [SJD98] verify that a deterministic implementation with in-order retirement satisfies *flush-point refinement* with a deterministic specification (FRDD). They build a non-deterministic intermediate model that computes the result of each instruction when it enters the machine and queues the result for later retirement. This intermediate model has hidden state relative to the implementation. The verification of the implementation against the intermediate model shows PEDN (pointwise equality). The verification of the intermediate model against the specification establishes FRND (flush-point refinement) by incrementally decomposing the monolithic flushing abstraction function into a set of simpler flushing steps. In [JSD98], they use a non-deterministic intermediate model with an abstracted scheduler that provides fine-grained control over instruction progress. This reduces the amount of manual abstraction required by strengthening the simpler flushing steps.

4.6 Variations on Refinement

The four works by the authors Damm, Pnueli, and Arons use a wide range of correctness statements and implementations. Damm and Pnueli [DP97] prove PANN (pointwise abstraction) for an implementation with out-of-order retirement. Their non-deterministic specification (NONDET) generates all possible traces of a program that obey data-dependencies, which allows them to use pointwise alignment. They introduce an intermediate model with auxiliary variables (TOMASULO) and prove PENN (pointwise equality) between the implementation and the intermediate model, and PANN between the intermediate model and the specification. For PANN their abstraction projects the current implementation state if all instructions have retired and otherwise returns the initial implementation state.

Arons and Pnueli [AP99] prove FRNN (flush-point refinement) for an implementation with out-of-order retirement. The specification can self-loop at every state, but is otherwise deterministic. They use an intermediate model with history variables and prove that whenever the implementation is flushed, the history variables match the implementation (FRNN). They verify that the intermediate model satisfies pointwise refinement

(PRNN) with the specification. Subsequently, Pnueli and Arons change their synchronization point from instruction issue to instruction retirement, which allows them to tighten their top-level correctness statement to be PRNN (pointwise refinement) for an implementation with in-order retirement [PA98].

Arons and Pnueli [AP00] verify SANN (stuttering abstraction) for a machine with speculative execution, precise exceptions, and in-order retirement. Their abstraction computes the abstract program counter based on the contents of the reorder buffer. They perform two different verifications, one based on induction over the size of the reorder buffer and one using abstraction functions. In the inductive proof, they use three intermediate models, and prove SRNN (stuttering refinement, relying on the result of [AL91]), PANN (pointwise abstraction), and SANN to conclude SANN overall.

4.7 Assume-Guarantee

Henzinger *et al.* [HQR98,Qad99] use a top-level correctness statement of pointwise equality (PENN), which they call *trace containment*, to prove the correctness of an out-of-order retirement processor where both the specification and implementation may have internal state. Their specification includes a non-deterministic stall signal and the scheduling in their implementation is non-deterministic. They construct abstraction and witness modules to bridge the gap between the specification and implementation. Using assume-guarantee reasoning, they reduce the problem to smaller proof obligations where the specification has no internal state. In these cases (which they call *projection refinement*), they prove PRNN (pointwise refinement).

4.8 Related Correctness Statements

Manolios [Man00] defines correctness based on *well-founded bisimulation*. He allows both the specification and implementation to be non-deterministic and to stutter, but also includes a liveness property that guarantees that they will stutter for only finitely many steps. This approach has not yet been applied to out-of-order implementations. If we excise the liveness criteria from his correctness statement, his work can be characterized as verifying that the implementation satisfies *stuttering equivalence* (SENN) against the specification and that the specification satisfies SENN against the implementation.

Fox and Harman [FH98] define a correctness statement that uses explicit time and temporal abstraction. Their theory supports arbitrary alignments based on temporal abstraction, that they call *retimings*, but the only supported matching instantiation is abstraction. They have used this statement in the verification of a superscalar machine with in-order retirement. For superscalar implementations, they align both the implementation and specification to an intermediate clock.

5 Discussion

We have presented a framework for describing microprocessor correctness statements that helps us to compare existing correctness statements and to highlight the differences among them. Our classification is meant as a stepping stone towards understanding the links between an implementation's features, the desired "strength" of correctness statement, and the verification techniques. Indeed, the framework has led us to a number of observations that we now discuss.

Machines with out-of-order retirement are problematic, because they can reach states that are not possible when executing instructions sequentially. One possibility is to use equality match, a deterministic specification, and flush-point alignment. Two other approaches support point-wise alignment: a non-deterministic specification that allows different retirement orderings [DP97] or an abstraction function that retires all in-flight instructions (e.g. flushing [BD94], or completion functions [HSG98]).

Sawada and Hunt [SH97] have verified the same implementation using flush-point equality (Definition 11) and Burch-Dill style pointwise abstraction (Definition 9). They found flush-point equality to be significantly easier. We speculate that flush-point equality is a verification convenience, i.e. realistic machines that satisfy flush-point equality will also satisfy pointwise abstraction or stuttering abstraction. In the case of machines without external interrupts, a flushing-style abstraction function should suffice, while a machine with interrupts would require a more sophisticated abstraction function to keep the interrupt trace aligned between the specification and implementation.

Stalls complicate the alignment of the implementation and specification. Pnueli and Arons [PA98] use PRNN (pointwise refinement) with a specification that self-loops when no instruction retires. Many others use pointwise abstraction where the abstraction function flushes the implementation and the specification self-loops when no instruction is issued. An emerging trend is to use flush-point equality or flush-point refinement, where the implementation and specification are compared only when the implementation is in a flushed state.

Verifying machines with exceptions complicates the instantiation of the match parameter. Most approaches in the literature synchronize the implementation and specification machines at instruction issue. However, Damm and Pnueli [DP97] and Pnueli and Arons [AP00] synchronize at retirement, an approach that makes it easier to handle exceptions. The synchronization point is encapsulated in the definition of the match parameter and is not distinguished by our framework.

In Fig. 3 almost all of the intermediate correctness statements lead to the top-level correctness statements by tracing along the edges in the graph. The two exceptions are incremental flushing [SJD98] and completion functions [HSG98], whose use of mechanized theorem proving enables these more complicated verification strategies.

We are formalizing the framework in a theorem prover and mechanically verifying the partial order between correctness statements. For a general matching relation, we have verified that pointwise implies stuttering, and stuttering implies flushing. We are investigating the logical relationships between must-issue and the other three.

Our framework is not an end in itself. Rather, it should be used as a foundation for further investigations and a deeper understanding of developments in the formal verification of microprocessors. There are values for the framework's parameters that we have not enumerated, and we anticipate that some of these will find useful application. For example, as other approaches besides Sawada and Hunt [SH98] begin to include external interrupts, we anticipate that additional points in the correctness space will be explored. It remains to be determined what the framework indicates about the relative "quality" of correctness criteria. It would also be fruitful to explore the potential of using the framework to predict the difficulty of different verification approaches.

Acknowledgments. We thank Andrew Martin of Motorola and the anonymous reviewers for their helpful comments on this work. The first and third authors are supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 2(82):253–284, 1991.

[AP99] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. In *Int'l Conference on VLSI Design*, pp 92–99, 1999.

[AP00] T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution with exceptions. In *TACAS*, vol 1785 of *LNCS*, pp 487–502. Springer, 2000.

[AS99] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1999.

[BB94] D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *DAC*, pp 596–602, 1994.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD*, vol 1522 of *LNCS*, pp 369–386. Springer, 1998.

[BD94] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, vol 818 of *LNCS*, pp 68–80. Springer, 1994.

[BF89] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *ICCD*, pp 217–221, 1989.

[BGV99] R. Bryant, S. German, and M. Velev. Processor verification using efficient decision procedures for a logic of uninterpreted functions. In *TABLEAUX*, vol 1617 of *LNAI*, pp 1–13. Springer, June 1999.

[Bur96] J. Burch. Techniques for verifying superscalar microprocessors. In *DAC*, pp 552–557, 1996.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In *CHARME*, pp 23–47. Chapman and Hall, 1997.

[FH98] A. Fox and N. Harman. An algebraic model of correctness for superscaler microprocessors. In *Prospects for Hardware Foundations*, vol 1546 of *LNCS*, pp 138–183. Springer, 1998.

[HGS00] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *CAV*, vol 1855 of *LNCS*, pp 521–537. Springer, 2000.

[HQR98] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV*, vol 1427 of *LNCS*, pp 440–451. Springer, 1998.

[HSG98] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In *CAV*, vol 1427 of *LNCS*, pp 122–134. Springer, 1998.

[HSG99] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *CAV*, vol 1633 of *LNCS*, pp 47–59. Springer, 1999.

[JSD98] R. Jones, J. Skakkebæk, and D. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *FMCAD*, vol 1522 of *LNCS*, pp 2–17. Springer, 1998.

[Man00] P. Manolios. Correctness of pipelined machines. In *FMCAD*, vol 1954 of *LNCS*, pp 161–178. Springer, 2000.

[McM98] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, vol 1427 of *LNCS*, pp 110–121. Springer, 1998.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. of 2nd Int'l Joint Conf. on Artificial Intelligence*, pp 481–489. The British Comp. Soc., 1971.

[NG98] R. Nalumasu and G. Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. In *Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98)*, 1998.

[PA98] A. Pnueli and T. Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In *FMCAD*, vol 1522 of *LNCS*, pp 351–368. Springer, 1998.

[PD96] S. Park and D. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, vol 1102 of *LNCS*, pp 300–310. Springer, 1996.

[PJB99] V. Patankar, A. Jain, and R. E. Bryant. Formal verification of an ARM processor. In *Int'l Conf. on VLSI Design*, pp 282–287. IEEE; New York, NY, January 1999.

[Qad99] S. Qadeer. *Algorithms and Methodology for Scalable Model Checking*. PhD thesis, Elec. Eng. and Comp. Sci., University of California at Berkeley, 1999.

[SA97] X. Shen and Arvind. A methodology for designing correct cache coherence protocols for DSM systems. Technical Report CSG Memo 398 (A), MIT, June 1997.

[SB90] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Trans. on Software Engineering*, pp 52–64, September 1990.

[SH97] J. Sawada and W. Hunt. Trace table based approach for pipelined microprocessor verification. In *CAV*, vol 1254 of *LNCS*, pp 364–375. Springer, 1997.

[SH98] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *CAV*, vol 1427 of *LNCS*, pp 135–146. Springer, 1998.

[SH99] J. Sawada and W. Hunt. Results of the verification of a complex pipelined machine model. In *CHARME*, vol 1703 of *LNCS*, pp 313–316. Springer, 1999.

[SJD98] J. Skakkebæk, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV*, vol 1427 of *LNCS*, pp 98–109. Springer, 1998.

[SM95] M. K. Srivas and S. P. Miller. Applying formal verification to a commercial microprocessor. In *CHDL*, pp 493–502, August 1995.

[WC94] P. Windley and M. Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design*, pp 32–51. Springer, 1994.