A Compositional Theory of Refinement for Branching Time

Panagiotis Manolios

Georgia Institute of Technology, CERCS Lab
801 Atlantic Drive
Atlanta, Georgia, 30332, USA
manolios@cc.gatech.edu
http://www.cc.gatech.edu/~manolios

Abstract. I develop a compositional theory of refinement for the branching time framework based on stuttering simulation and prove that if one system refines another, then a refinement map always exists. The existence of refinement maps in the linear time framework was studied in an influential paper by Abadi and Lamport. My interest in proving analogous results for the branching time framework arises from the observation that in the context of mechanical verification, branching time has some important advantages. By setting up the refinement problem in a way that differs from the Abadi and Lamport approach, I obtain a proof of the existence of refinement maps (in the branching time framework) that does not depend on any of the conditions found in the work of Abadi and Lamport *e.g.*, machine closure, finite invisible nondeterminism, internal continuity, the use of history and prophecy variables, etc. A direct consequence is that refinement maps always exist in the linear time framework, subject only to the use of prophecy-like variables.

1 Introduction

Computing systems are ubiquitous, controlling everything from cars and airplanes to financial markets and the distribution of information. Such systems tend to be very complicated and often contain costly errors. One approach to dealing with this complexity is to specify a sequence of related systems, starting with an abstract system, the specification, and ending with a concrete system, the implementation. One then proves that every pair of adjacent systems is related, via a suitable, compositional notion of correctness, thereby establishing that the specification is correctly implemented. For example, we can imagine verifying a netlist description of a pipelined microprocessor, the implementation, by relating it via a sequence of refinements to an instruction set level specification—the assembly programmer's view of the processor.

Two important concepts that notions of correctness must account for are:

– **Stuttering.** Since the specification is defined at a more abstract level than the implementation, notions of correctness should allow for *stuttering*, where the implementation may require several steps before matching a single step of the specification [14].

– **Refinement.** The implementation may contain more state components and may use different data representations than the specification. *Refinement maps* are used to show how to view an implementation state as a specification state [1].

The classic paper on the topic by Abadi and Lamport [1], which has motivated the work appearing in this paper, contains an in-depth discussion of these topics. The main idea is to use refinement maps to prove that systems have related infinite computations, by reasoning *locally*, about states and their successors, instead of *globally*, about infinite paths. Abadi and Lamport prove a theorem about when such refinement maps exist in the linear time framework, where the semantics of systems and properties correspond to sets of infinite sequences.

My approach differs in that I work in the branching time framework, where the semantics of systems are given by sets of infinite trees. Even so, the results can be applied to the linear time framework, as I explain later.

The theorem proved by Abadi and Lamport holds only under certain conditions. Briefly, they allow one to add history and prophecy variables to the implementation, they require that the implementation is machine closed, and they require that the specification has finite invisible nondeterminism and is internally continuous. My theorems do not depend on these conditions, but there are important differences between the two approaches that are explored in depth later.

There are two main reasons why I chose to work in the branching-time framework. The first is that in the simple case where one is dealing with finite-state systems, it makes sense to use algorithms that can check if one finite-state system refines another. For example, in [17] we use algorithms for deciding stuttering bisimulation to complete a proof of correctness for the alternating bit protocol (this is an infinite-state problem that was reduced to a finite-state problem using a theorem prover). The branching time notions of simulation and bisimulation, due to Milner and Park [18, 21], can be decided in polynomial time [20, 7]. In contrast, the corresponding linear time notions, trace equivalence and trace containment, are both PSPACE-complete problems [26].

Second, refinement maps allow one to show that one system *simulates* another. This is inherently a branching time notion which has the advantage of being structural and local. However, in order to use refinement maps in a linear time setting other mechanisms are needed to, in essence, hide the branching structure of systems. Thus, we expect the branching time case to be simpler than the linear time case. Obvious questions arise. How much simpler? What conditions in the Abadi and Lamport theorem are there for this purpose? It turns out that by using only prophecy-like variables, which have the effect of destroying the branching structure of systems, we can get a completeness theorem for the linear time.

Stuttering simulation is based on the notions of simulation and bisimulation, which have had a deep impact on how we think about specifications. The literature on this topic is vast and contains many fine surveys [23, 15, 6]. In addition, there have been various extensions of the Abadi and Lamport result [1], including [5, 9, 2, 8]. In related previous work, Namjoshi [19] gives a sound and complete proof rule for symmetric stuttering bisimulations which has heavily influenced my work; however, Namjoshi does not consider simulations and does not deal with refinement. Stuttering bisimulations and the related notion of WEBs (Well-founded Equivalence Bisimulations) were used to link theorem proving and model checking and to mechanically verify the alternating bit protocol in [17]. In [16], I proposed a notion of correctness for pipelined machines based on WEBs and I showed that the variant of the Burch and Dill notion of correctness [4] in [24, 25] can be satisfied by machines that deadlock. In addition, I used the ACL2 theorem prover [12, 11, 10] to automate much of the verification. I also verified variants of the pipelined machine including machines with exceptions, interrupts (which lead to non-determinism), and netlist (gate-level) descriptions and showed that my notion of correctness applies to these extensions. Many of the variant machines were verified in stages, using the WEB compositional proof rule. Unfortunately, stuttering bisimulation and WEBs are often too strong a notion, just as trace equivalence is often too strong a notion in the linear time case. I expect stuttering simulation to be much more applicable, hence my interest in the topic.

The paper is organized as follows. In section 2, I describe my notational conventions and review background material. In section 3, I develop a theory of refinement based on stuttering simulation. In section 4, I discuss refinement in the linear time framework and compare my work with that of Abadi and Lamport; some readers may want to start by skimming this section first. I conclude in section 5.

2 Notation and Mathematical Preliminaries

$\mathbb{N}$ and ω both denote the natural numbers, *i.e.*, $\{0, 1, \ldots\}$. The ordered pair whose first component is i and whose second component is j is denoted $\langle i, j\rangle$. $[i..j]$ denotes the closed interval $\{k \in \mathbb{N} : i \le k \le j\}$; parentheses are used to denote open and half-open intervals, *e.g.*, $[i..j)$ denotes the set $\{k \in \mathbb{N} : i \le k < j\}$. The disjoint union operator is denoted by $\uplus$. Cardinality of a set S is denoted by $|S|$. $\mathcal{P}(S)$ denotes the powerset of S. Function application is sometimes denoted by an infix dot ".". For any binary relation R: I abbreviate $\langle s, w\rangle \in R$ by sRw, I write $R(S)$ for the *image* of S under R (*i.e.*, $R(S) = \{y : \langle \exists x : x \in S : xRy\rangle\}$), and $R|_A$ denotes R *left-restricted* to the set A (*i.e.*, $R|_A = \{\langle a, b\rangle : (aRb) \;\wedge\; (a \in A)\}$). The *composition* of binary relations R and T is denoted $R;T$ or $T \circ R$, *i.e.*, $R;T = T \circ R = \{\langle r, t\rangle : \langle \exists x :: rRx \;\wedge\; xTt\rangle\}$. The *inverse* of binary relation R is denoted R^{-1} and is defined to be $\{\langle a, b\rangle : bRa\}$.

$\langle Qx : r : b\rangle$ denotes a quantified expression, where Q is the quantifier, x the bound variable, r the range of x (**true** if omitted), and b the body. I sometimes write $\langle Qx \in X : r : b\rangle$ as an abbreviation for $\langle Qx : x \in X \;\wedge\; r : b\rangle$, where r is **true** if omitted, as before. From highest to lowest binding power, we have: parentheses, function application, binary relations (*e.g.*, sBw), equality

($=$) and membership ($\in$), conjunction ($\wedge$) and disjunction ($\vee$), implication ($\Rightarrow$), and finally, binary equivalence ($\equiv$).

Spacing is used to reinforce binding: more space indicates lower binding.

A binary relation, $B \subseteq X \times X$, is *reflexive* if $\langle \forall x \in X :: xBx \rangle$. B is *symmetric* if $\langle \forall x, y \in X :: xBy \quad \Rightarrow \quad yBx \rangle$. B is *antisymmetric* if $\langle \forall x, y \in X :: xBy \quad \wedge \quad yBx \quad \Rightarrow \quad x = y \rangle$. B is *transitive* if $\langle \forall x, y, z \in X :: xBy \quad \wedge \quad yBz \quad \Rightarrow \quad xBz \rangle$. A binary relation is a *preorder* if it is reflexive and transitive. A preorder that is also symmetric is an *equivalence relation*.

A *finite sequence* is a function from $[0..n)$ for some natural number n. An *infinite sequence* is a function from $\mathbb{N}$. When I write $x \in \sigma$, for a sequence σ, I mean that x is in the range of σ. A *well-founded structure* is a pair $\langle W, \lessdot \rangle$ where W is a set and $\lessdot$ is a binary relation on W such that there are no infinitely decreasing sequences on W, with respect to $\lessdot$. I use $<$ to compare natural numbers and $\prec$ to compare ordinal numbers.

A *transition system* (TS) is a structure $\langle S, \dashrightarrow, L \rangle$, where S is a set of states, $\dashrightarrow \subseteq S \times S$ is the *transition relation*, L is the *labeling function*: its domain is S and it tells us what is observable at a state. I also require that $\dashrightarrow$ is *left-total*: for every $s \in S$, there is some $u \in S$ such that $s \dashrightarrow u$. Notice that a transition system is a labeled graph where the nodes are states and are labeled by L.

A *path* σ is a sequence of states such that for adjacent states s and u, $s \dashrightarrow u$. A path, σ, is a *fullpath* if it is infinite. $fp.\sigma.s$ denotes that σ is a fullpath starting at state s and σ^i denotes the suffix fullpath $\langle \sigma.i, \sigma(i+1), \ldots \rangle$. I use the symbol ";" for concatenation of paths where the left path is finite, *e.g.*, $a; ab = aab$.

Temporal logic was proposed as a formalism for specifying the correctness of computing systems in a landmark paper by Pnueli [22]. I assume that the reader is familiar with temporal logic.

3 Stuttering Simulation Refinement

Stuttering simulation depends on the notion of matching I now define. I start with an informal account. Given a relation B on a set S, we say that an infinite sequence σ (of elements from S) matches an infinite sequence δ (of elements from S) if the sequences can be partitioned into non-empty, finite segments such that elements in related segments are related by B. For example, if the first segment of σ has three elements and the first segment of δ has seven elements, then each of the three elements is related by B to each of the seven elements. I use matching, where the infinite sequences are fullpaths of a transition system, to define stuttering simulation.

Definition 1. *(match)* Let i range over $\mathbb{N}$. Let INC be the set of strictly increasing sequences of natural numbers starting at 0; formally, $INC = \{\pi : \pi : \mathbb{N} \rightarrow \mathbb{N} \quad \wedge \quad \pi.0 = 0 \quad \wedge \quad \langle \forall i \in \mathbb{N} :: \pi.i < \pi(i+1) \rangle \}$. The i^{th} segment of an infinite sequence σ with respect to $\pi \in INC$, ${}^{\pi}\sigma^i$, is given by the sequence $\langle \sigma(\pi.i), \ldots, \sigma(\pi(i+1)-1) \rangle$.

For $B \subseteq S \times S$, $\pi, \xi \in INC$, $i, j \in \mathbb{N}$, and infinite sequences σ and δ, I abbreviate $\langle \forall s, w : s \in {}^{\pi}\sigma^i \wedge w \in {}^{\xi}\delta^j : sBw \rangle$ by $({}^{\pi}\sigma^i)B({}^{\xi}\delta^j)$.

In addition: $corr(B,\sigma,\pi,\delta,\xi) \equiv \langle \forall i \in \mathbb{N} :: ({}^{\pi}\sigma^i) B ({}^{\xi}\delta^i) \rangle$ and $match(B,\sigma,\delta) \equiv \langle \exists \pi, \xi \in INC :: corr(B,\sigma,\pi,\delta,\xi) \rangle$.

Lemma 1. *Given set S, $B \subseteq S \times S$, and infinite sequences σ and δ,*

$$\langle \exists \pi, \xi \in INC :: corr(B,\sigma,\pi,\delta,\xi) \rangle$$
$$\equiv$$
$$\langle \exists \pi', \xi' \in INC :: corr(B,\sigma,\pi',\delta,\xi') \;\; \wedge \;\; \langle \forall i \in \mathbb{N} :: |{}^{\pi'}\sigma^i| = 1 \;\; \vee \;\; |{}^{\xi'}\delta^i| = 1 \rangle \rangle$$

The above lemma allows us to reason about segments using case analysis, where the three cases are: both segments are of length 1, the right segment is of length 1 and the left of length greater than 1, and the left segment is of length 1 and the right of length greater than 1.

3.1 Stuttering Simulation

A relation on $B \subseteq S \times S$ where $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ is a stuttering simulation, if for every s, w such that sBw, s and w are identically labeled and every fullpath starting at s can be matched by some fullpath starting at w.

Definition 2. *(Stuttering Simulation (STS))* $B \subseteq S \times S$ is a stuttering simulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$ iff for all s, w such that sBw:

(Sts1) $L.s = L.w$

(Sts2) $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B,\sigma,\delta) \rangle \rangle$

Lemma 2. $(B \subseteq C) \quad \Rightarrow \quad [match(B,\sigma,\delta) \Rightarrow match(C,\sigma,\delta)]$

Lemma 3. *Let $\mathcal{C}$ be a set of STS's on TS $\mathcal{M}$, then $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ is an STS on $\mathcal{M}$.*

Corollary 1 *For every TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$.*

Lemma 4. *If R and S are STS's, so is $T = R;S$.*

Lemma 5. *The reflexive, transitive closure of an STS is an STS.*

Theorem 1. *Given TS $\mathcal{M}$, there is a greatest STS on $\mathcal{M}$, which is a preorder.*

Theorem 2. *Let B be a STS on $\mathcal{M}$ and let sBw. For every* $\mathrm{ACTL}^* \setminus \mathsf{X}$ *formula f, if $\mathcal{M}, w \models f$ then $\mathcal{M}, s \models f$.*

3.2 Well-Founded Simulation

In order to check that a relation is an STS, we have to show that infinite sequences "match". This can be problematic when using computer-aided verification techniques. I present the notion of a *well-founded simulation* to remedy this situation. To show that a relation is a well-founded simulation, we need only check local properties; this is analogous to proving program termination by exhibiting a function that maps states into a well-founded relation and showing that the function decreases during every step of the program. As mentioned previously, the intuition is that for every pair of states s, w that are related by an STS and u such that $s \dashrightarrow u$, there are essentially three cases: either there is a v such that $w \dashrightarrow v$ and u is related to v, or u is related to w, or there is a v such that $w \dashrightarrow v$ and s is related to v. In the last two cases, we must also ensure that we do not have an infinite sequence of states, each of which is related to a single state. This is where the well-founded relation comes in: we must show that in these cases there is an appropriate measure function into a well-founded relation that decreases. Formally, we have:

Definition 3. *(Well-Founded Simulation (WFS))* $B \subseteq S \times S$ is a well-founded simulation on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff:

(Wfs1) $\langle \forall s, w \in S : sBw : L.s = L.w\rangle$

(Wfs2) There exists functions, $\mathit{rankt} : S \times S \to W, \mathit{rankl} : S \times S \times S \to \mathbb{N}$, such that $\langle W, \lessdot\rangle$ is well-founded, and

$\langle \forall s, u, w \in S : sBw \quad \wedge \quad s \dashrightarrow u :$

(a) $\langle \exists v : w \dashrightarrow v : uBv\rangle \quad \vee$

(b) $(uBw \quad \wedge \quad \mathit{rankt}(u, w) \lessdot \mathit{rankt}(s, w)) \quad \vee$

(c) $\langle \exists v : w \dashrightarrow v : sBv \quad \wedge \quad \mathit{rankl}(v, s, u) < \mathit{rankl}(w, s, u)\rangle\rangle$

3.3 Equivalence

In this section, I show that well-founded simulation completely characterizes stuttering simulation. Thus, we can think of well-founded simulation as a sound and complete proof rule.

Proposition 1 *(Soundness) If B is a WFS, then it is an STS.*

Proof Let aBb; we need to show Sts1 and Sts2. $L.a = L.b$ since B is a WFS (Wfs1), thus Sts1 holds. We show $\langle \forall \sigma : \mathit{fp}.\sigma.a : \langle \exists \delta : \mathit{fp}.\delta.b : \mathit{match}(B, \sigma, \delta)\rangle\rangle$, namely that Sts2 holds. Suppose $\mathit{fp}.\sigma.a$. We define fullpath δ and increasing sequences π, ξ recursively as follows: $\delta.0 = b, \pi.0 = 0, \xi.0 = 0$. The idea is that from $\pi.i, \xi.i, \delta(\xi.i)$ we can define $\pi(i+1), \xi(i+1), {}^{\xi}\delta^{i}, \delta.\xi(i+1)$ with ${}^{\pi}\sigma^{i}$, ${}^{\xi}\delta^{i}$ matching. $\Box$

We now prove that every STS is a WFS. For the proof, we have to exhibit the rank functions as per the definition of WFS. Here is a high-level overview.

The value of $\mathit{rankt}(s, w)$ is important only if sBw, as otherwise there are no restrictions required by the definition of WFS. If sBw, then consider the largest subtree of the computation tree rooted at s such that no node in the subtree matches a successor of w. The "rank" (a kind of height) of this subtree is the value of $\mathit{rankt}(s, w)$. The "rank" of s is greater than the "rank" of any of its children in the tree, so case Wfs2b is satisfied.

The value of $\mathit{rankl}(w, s, u)$ is important only if sBw and $s \dashrightarrow u$, as otherwise there are no restrictions required by the definition of WFS. If sBw and $s \dashrightarrow u$, then $rankl(w, s, u)$ is the length of the shortest path from w that matches s, u. In the case of Wfs2c, we can choose the next successor of w in this path to satisfy the condition.

Given a TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, the notion of the *computation tree* rooted at a state $s \in S$ is standard. It is the tree obtained by unfolding $\mathcal{M}$ starting from s and can be defined as follows. The nodes of the tree are finite sequences over S. The tree is defined to be the smallest tree satisfying the following.

1. The root is $\langle s\rangle$.
2. If $\langle s, \ldots, w\rangle$ is a node and $w \dashrightarrow v$, then $\langle s, \ldots, w, v\rangle$ is a node whose parent is $\langle s, \ldots, w\rangle$.

Definition 4. *(tree)* Given an STS B, if $\neg(sBw)$, then $tree(s, w)$ is the empty tree, otherwise $tree(s, w)$ is the largest subtree of the computation tree rooted at s such that for every non-root node of the tree, $\langle s, \ldots, x\rangle$, we have that xBw and $\langle \forall v : w \dashrightarrow v : \neg(xBv)\rangle$.

Lemma 6. *Every path of $tree(s, w)$ is finite.*

Since the child relation on nodes in $tree.s$ is well-founded, we can recursively define a labeling function, l, that assigns an ordinal to nodes in the tree as follows: $l.n = \langle \cup c : c \text{ is a child of } n : (l.c) + 1\rangle$. This is the standard "rank" function encountered in set theory [13]. We use the convention that the label of a tree is the label of its root.

Lemma 7. *If $|S| \preceq \kappa$, where κ is an infinite cardinal (*i.e.*, $\omega \preceq \kappa$) then for all $s, w \in S$, $tree(s, w)$ is labeled with an ordinal of cardinality $\preceq \kappa$.*

Lemma 8. *If $sBw, s \dashrightarrow u, u \in tree(s, w)$ then $l.tree(u, w) \prec l.tree(s, w)$.*

Definition 5. *(length)* Given B, an STS, $length(w, s, u) = 0$ if $\neg(sBw)$ or $\neg(s \dashrightarrow u)$, otherwise $length(w, s, u)$ is the length of the shortest initial segment starting at w that matches $\langle s, u\rangle$. Formally:

$$length(w, s, u) = \langle min\ \sigma, \delta, \pi, \xi : fp.\sigma.s \ \wedge\ \sigma.1 = u \ \wedge\ fp.\delta.w \ \wedge\ \pi, \xi \in INC \ \wedge\ corr(B, \sigma, \pi, \delta, \xi) : |\,{}^{\xi}\delta^{0}\,|\rangle$$

As sBw and $s \dashrightarrow u$, the above range is non-empty and $length(w, s, u) \in \mathbb{N}$.

Lemma 9. *If $sBw, s \dashrightarrow u$ and $\neg\langle \exists \sigma, \delta, \pi, \xi : fp.\sigma.s \wedge \sigma.1 = u \wedge fp.\delta.w \wedge \pi, \xi \in INC : corr(B, \sigma, \pi, \delta, \xi) \ \wedge\ {}^{\xi}\delta^{0} = \langle w\rangle\rangle$, then $\langle \exists v : w \dashrightarrow v : length(v, s, u) < length(w, s, u) \ \wedge\ sBv\rangle$.*

Proposition 2 *(Completeness) If B is an STS, then B is a WFS.*

Proof Wfs1 follows from Sts1. Let $W = (|S| + \omega)^{+}$. Note that $+$ denotes cardinal arithmetic; we add ω to $|S|$ to guarantee that we have an infinite cardinal; κ^{+} is the successor cardinal to κ.

Clearly, $(W, \prec)$ is well-founded. Let $rankt = l.tree$ and let $rankl = length$. Let sBw and $s \dashrightarrow u$. There are three cases:

1. $\langle \exists v : w \dashrightarrow v : uBv \rangle$. By lemma 1, if (1) does not hold, then for every σ, δ, π, ξ such that $\mathit{fp}.\sigma.s \wedge \sigma.1 = u \wedge \mathit{fp}.\delta.w \wedge \pi, \xi \in \mathit{INC} \wedge \mathit{corr}(B, \sigma, \pi, \delta, \xi)$, either s marks the end of ${}^{\pi}\sigma^0$ or w marks the end of ${}^{\xi}\delta^0$, but not both.
2. $\langle \exists \sigma, \delta, \pi, \xi : \mathit{fp}.\sigma.s \wedge \sigma.1 = u \wedge \mathit{fp}.\delta.w \wedge \pi, \xi \in \mathit{INC} \wedge {}^{\xi}\delta^0 = \langle w \rangle : \mathit{corr}(B, \sigma, \pi, \delta, \xi) \rangle$ and (1) does not hold. This implies that $|\,{}^{\pi}\sigma^0\,| > 1$, uBw, and $u \in \mathit{tree}(s, w)$; hence, $\mathit{rankt}(u, w) \prec \mathit{rankt}(s, w)$ by lemma 8.
3. If (1) and (2) do not hold, we must have $\neg\langle \exists \sigma, \delta, \pi, \xi : \mathit{fp}.\sigma.s \wedge \sigma.1 = u \wedge \mathit{fp}.\delta.w \wedge \pi, \xi \in \mathit{INC} : \mathit{corr}(B, \sigma, \pi, \delta, \xi) \wedge {}^{\xi}\delta^0 = \langle w \rangle \rangle$. By lemma 9 and the definition of rankl, $\langle \exists v : w \dashrightarrow v : \mathit{rankl}(v, s, u) < \mathit{rankl}(w, s, u) \wedge sBv \rangle$. $\square$

Theorem 3. *(Equivalence) B is an STS iff B is a WFS.*

A consequence of the above theorem is that all of the properties proved for STSs carry over to WFSs; I use this fact freely, without reference, in the sequel.

3.4 Refinement

Up to this point, I have developed a theory for relating states. I now show how to apply the theory to transition systems. In this section, I define a notion of refinement and show that STSs can be used in a compositional fashion. For states s and w, I write $s \sqsubseteq w$ to mean that there is an STS B such that sBw.

By theorem 1, $s \sqsubseteq w$ iff sGw, where G is the greatest STS. I now lift this idea to transition systems.

Definition 6. *(Simulation Refinement)* Let $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L' \rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a simulation refinement of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \sqsubseteq_r \mathcal{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sB(r.s) \rangle$ and B is an STS on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L} \rangle$, where $\mathcal{L}.s = L'(s)$ for s an S' state and $\mathcal{L}.s = L'(r.s)$ otherwise.

In the above definition, it helps to think of $\mathcal{M}'$ as the specification and $\mathcal{M}$ as the implementation. That $\mathcal{M}$ is a simulation refinement of $\mathcal{M}'$ with respect to r implies that every visible behavior of $\mathcal{M}$ (where what is visible depends on r) is a behavior of $\mathcal{M}'$. There are often other considerations, *e.g.*, it might be that $\mathcal{M}$ and $\mathcal{M}'$ have certain states that are "initial". In this case one might wish to show that initial states in $\mathcal{M}$ are mapped to initial states in $\mathcal{M}'$.

One has a great deal of flexibility in choosing refinement maps. The danger is that by choosing a complicated refinement map, one can bypass the verification problem all together. To make this point clear, let PRIME be the system whose single behavior is the sequence of primes and let NAT be the system whose single behavior is the sequence of natural numbers. We do not consider NAT to be an implementation of PRIME, but using the refinement map from NAT to PRIME that maps i to the i^{th} prime, we can indeed prove the peculiar theorem that NAT is a refinement of PRIME. The moral is that we must be careful to not bypass the verification problem with the use of such refinement maps. Simple refinement maps with a clear relationship between implementation states and their image under the map are best. The reason we do not place restrictions on refinement maps is that it is not a priori apparent what the "reasonable" relationships between implementation states and specification states might be, *e.g.*, suppose that the specification system represents numbers in decimal but the implementation system represents numbers in binary, or that numbers in the specification are spread across several registers in the implementation, and so on. Often refinement maps are especially clear, which makes it easy to check that they are in fact appropriate. Suppose that associated with states is a set of variables, each of a particular type. Furthermore, suppose that the variables in the implementation are a superset of the variables in the specification and that the refinement map just hides the implementation variables that do not appear in the specification. Then, it is clear that the refinement map is a reasonable one. More precisely, given TS $\mathcal{M} = \langle S, \dashrightarrow, L \rangle$, if L has the following structure, we say that $\mathcal{M}$ is *typed.*

Let VARS be a set and let TYPE be a function whose domain is VARS. Think of VARS as the variables of TS $\mathcal{M}$, where TYPE gives the type of the variables. For all $s \in S$, let $L.s$ be a function from VARS such that $L.s.v \in \mathit{TYPE}.v$. The lemma below shows why the appropriateness of refinement maps that hide some of the implementation variables is easy to ascertain.

Lemma 10. *If $\mathcal{M} = \langle S, \dashrightarrow, L \rangle \sqsubseteq_r \mathcal{M}' = \langle S', \dashrightarrow', L' \rangle$, both $\mathcal{M}$ and $\mathcal{M}'$ are typed TSs, and $L'(r.s) = L.s|_V$, then for every pair of states $s, r.s$ such that $s \in S$, and every* ACTL* $\setminus$ X *formula, f, built out of expressions that only depend on variables in V, we have $\mathcal{M}', r.s \models f \;\Rightarrow\; \mathcal{M}, s \models f$.*

Lemma 11. *If B is an STS on TS $\mathcal{M} = \langle S \supseteq S_1 \cup S_2, \dashrightarrow, L \rangle$, $S_1 \cap S_2 = \emptyset$, states in S_1 can only reach states in S_1, and states in S_2 can only reach states in S_2, then $\hat{B} = \{\langle s_1, s_2 \rangle : s_1 \in S_1 \;\wedge\; s_2 \in S_2 \;\wedge\; s_1 B s_2\}$ is an STS on $\mathcal{M}$.*

Theorem 4. *(Composition) If $\mathcal{M} \sqsubseteq_r \mathcal{M}'$ and $\mathcal{M}' \sqsubseteq_q \mathcal{M}''$ then $\mathcal{M} \sqsubseteq_{r;q} \mathcal{M}''$.*

4 The Linear Time Case

The theorem on the existence of refinement maps in the previous section does not apply to the linear time framework because simulation is a stronger property than trace containment. However, note that if we destroy the branching structure of transition system $\mathcal{M}$ to obtain transition system $\mathcal{M}'$, then $\mathcal{M}' \sqsubseteq_r \mathcal{N}$ iff the set of infinite sequences of $\mathcal{M}$, labeled by r, is a subset of the set of sequences of $\mathcal{N}$. We can destroy the branching structure of $\mathcal{M}$ by using an *oracle variable* to record values for every non-deterministic choice made along an infinite path in the computation tree of $\mathcal{M}$. We have thus sketched a proof of the existence of refinement maps in the linear time framework.

Theorem 5. *If the set of traces of $\mathcal{M}$ is a subset of the traces of $\mathcal{N}$, then there exists $\mathcal{M}'$, a transition system obtained from $\mathcal{M}$ by adding an oracle variable, and a refinement map r such that $\mathcal{M}' \sqsubseteq_r \mathcal{N}$.*

I now review the work of Abadi and Lamport on the existence of refinement maps. The review addresses the essential points, but is necessarily concise and readers are urged to read the full paper. I then present several examples, taken from Abadi and Lamport, that are used to justify the conditions appearing in their theorem. At the end of this section, I compare the two approaches.

4.1 Review of Abadi and Lamport Results

I begin by reviewing some initial definitions. A *behavior* is an infinite sequence and a *property* is a set of behaviors closed under finite stuttering. A *specification* is a (possibly infinite) state machine, consisting of externally visible components and internal components, and a *supplementary* property to represent fairness constraints. The *complete property* of a state machine is obtained by closing the set of behaviors allowed by the machine under (possibly infinite) stuttering. The *externally visible property* of a state machine is obtained by projecting the externally visible components of the complete property of the state machine. The *property* defined by a specification is obtained by intersecting the complete property of its state machine with the supplementary property. The *externally visible property* of a specification is obtained by projecting the externally visible components of the property of the specification.

We say that I, a "concrete" specification (the Implementation), *implements* S, an "abstract" specification (the Specification) if every externally visible behavior of I is also a behavior of S. Proving that I implements S can require reasoning about arbitrary sequences because one has to show that if I admits the behavior $\langle\langle e_0, z_0\rangle, \langle e_1, z_1\rangle, \ldots, \langle e_n, z_n\rangle, \ldots\rangle$, where the e_i correspond to the externally visible components and the z_i to the internal components, then S admits the behavior $\langle\langle e_0, y_0\rangle, \langle e_1, y_1\rangle, \ldots, \langle e_n, y_n\rangle, \ldots\rangle$. Notice that y_n can depend upon the entire sequence $\langle\langle e_0, z_0\rangle, \langle e_1, z_1\rangle, \langle e_2, z_2\rangle, \ldots\rangle$, which can make the proof difficult. We prefer to avoid such global reasoning and would rather reason locally *e.g.*, if there is a function f such that $\langle e_i, y_i\rangle = f(e_i, z_i)$, it can be used to prove that I preserves the safety property of S by reasoning about pairs of states instead of arbitrary sequences of states. If such a function also preserves liveness, it is called a *refinement mapping* and Abadi and Lamport prove the following completeness theorem, showing under what conditions refinement mappings exist.

Theorem 6. *If the machine-closed specification I implements S, a specification that has finite invisible nondeterminism and is internally continuous, then there is a specification I^h obtained from I by adding a history variable and a specification I^{hp} obtained from I^h by adding a prophecy variable such that there exists a refinement mapping from I^{hp} to S.*

The above theorem depends on various conditions, which I now explain. We say that a specification I is *machine-closed* if the supplementary property of I does not specify any safety property not already specified by the state machine of I. A specification S has *finite invisible nondeterminism* if for every finite prefix of every behavior allowed by S, whenever an infinite nondeterministic choice is made, all but a finite part of that choice is immediately revealed in the externally visible components of the resulting state. A specification S is *internally continuous* if for every behavior, if the behavior is not allowed by the specification, this can be determined by looking at the externally visible part of the behavior and some *finite* portion of the complete behavior.

A *history variable* is used to extend the state space of a specification with a component that records past information, but in a way that does not affect the externally visible behaviors of the specification. Abadi and Lamport give five conditions that must be satisfied in order to show that if a specification S^h is obtained from a specification S by adding a history variable, then the two specifications define the same externally visible property. A *prophecy variable* is the dual of a history variable. Instead of recording past information, it guesses future information. Abadi and Lamport give six conditions that must be satisfied in order to show that if a specification S^p is obtained from a specification S by adding a prophecy variable, then the two specifications define the same externally visible property.

4.2 Examples due to Abadi and Lamport

This section contains several examples that Abadi and Lamport use to explain the conditions found in their completeness theorem. After the examples are introduced, I show how they can be handled using in my framework.

In the first example, system $\mathcal{S}$ is a three-bit clock, where only the low-order bit is externally visible and system $\mathcal{I}$ is a one-bit clock. $\mathcal{I}$ implements $\mathcal{S}$ since they have the same traces (up to stuttering). However, no refinement mapping can be used to show this because there is no way to define the internal state of $\mathcal{S}$: consider an arbitrary refinement mapping, r, and suppose that $r(\langle 0 \rangle) = \langle 0, y_0 \rangle$ and $r(\langle 1 \rangle) = \langle 1, y_1 \rangle$, then either $\langle 0, y_0 \rangle$ does not transit to $\langle 1, y_1 \rangle$ or $\langle 1, y_1 \rangle$ does not transit to $\langle 0, y_0 \rangle$. This is one reason for introducing history variables and they are used to resolve the dilemma as follows. A history variable is added to $\mathcal{I}$ and the variable "remembers" what $\mathcal{I}$ did in the past. The result is that the state space of $\mathcal{I}$ is expanded so that there are enough states to define an appropriate refinement mapping.

Using the approach outlined in this paper, we find that history variables are not needed as we can define a refinement map that maps the state in $\mathcal{I}$ whose counter is 0 to any state in $\mathcal{S}$ whose low-order bit is 0 and similarly with the other state in $\mathcal{I}$. The equivalence relation that relates states with the same low-order bit in the disjoint union of the two systems is a stuttering simulation.

The second example is used to motivate the need for prophecy variables. System $\mathcal{S}$ chooses ten values non-deterministically and displays each in turn, whereas system $\mathcal{I}$ chooses each value as it is displayed. $\mathcal{I}$ implements $\mathcal{S}$ since they have the same traces, but there is no refinement mapping that can be used to show this, as should be clear. This example highlights that proofs based on refinement mappings are based on simulation, a branching time notion. Thus, when $\mathcal{I}$ is not a stuttering simulation of $\mathcal{S}$, one cannot directly use refinement mappings to prove that $\mathcal{I}$ implements $\mathcal{S}$ (in the linear time sense). This is one reason for introducing prophecy variables and they are used to resolve the dilemma as follows. A prophecy variable is added to $\mathcal{I}$ and the variable "guesses" what $\mathcal{I}$ will decide to do in the future. There is now a refinement map, based on this prophecy variable, that can be used to show that $\mathcal{I}$ implements $\mathcal{S}$. What is happening is that the prophecy variables allow one to push all of the branching in the computation tree of $\mathcal{I}$ up to the root, thereby destroying the branching structure of $\mathcal{I}$.

This example shows why oracle variables are used in theorem 5. Note that from the branching point of view $\mathcal{I}$ does not implement $\mathcal{S}$, *e.g.*, from the initial state in $\mathcal{I}$, there is a successor that has more than one possible future, a branching-time expressible property that does not hold in the initial state of $\mathcal{S}$. It seems that any refinement-based approach will need a mechanism for dealing with this issue, whether it is by destroying the branching structure of implementations, by adding branching structure to specifications, or by some combination thereof.

The third example shows why a prophecy variable is needed to slow down an implementation that runs faster than a specification, even though the specification is just stuttering. Both $\mathcal{I}$ and $\mathcal{S}$ specify clocks in which the hours and minutes are externally visible, whereas the seconds are internal. Furthermore, $\mathcal{I}$ increments the clock by one second, whereas $\mathcal{S}$ increments the clock by ten seconds. Both $\mathcal{I}$ and $\mathcal{S}$ have the same externally visible behaviors and proving that $\mathcal{S}$ implements $\mathcal{I}$ using refinement mappings is easy. However, there is no way to show that $\mathcal{I}$ implements $\mathcal{S}$, because there is a behavior of $\mathcal{S}$ such that the minute hand changes every six steps, but any behavior of $\mathcal{I}$ requires at least sixty steps between minute hand changes.

In my formulation, the implementation is allowed to run faster than the specification, as we can both add and remove stuttering steps, thus it is easy to deal with the third example.

Abadi and Lamport present examples showing why the conditions of finite invisible nondeterminism and internal continuity are required. The examples are similar in that the implementation, $\mathcal{I}$, has the same externally visible behaviors as the specification, $\mathcal{S}$, but $\mathcal{I}$ has a richer branching structure than $\mathcal{S}$, *i.e.*, $\mathcal{S}$ is a simulation refinement of $\mathcal{I}$, but not the other way. As we have seen in the second example, above, prophecy variables can be used to deal with this problem. However, in these examples there are states in $\mathcal{I}$ that are related to an infinite number of states in $\mathcal{S}$, and Abadi and Lamport's prophecy variables cannot be used in this case (see their paper for the full details). To summarize, the conditions of internal continuity and finite invisible nondeterminism in the completeness theorem of Abadi and Lamport can be traced to the *branching* structure of the systems involved.

Oracle variables can be used in my approach to deal with these examples. The intuition is that oracle variables allow us to quantify over every possible nondeterministic choice and can be used to transform $\mathcal{I}$ into a linear time equivalent system in which all nondeterministic choices have been made at the onset.

4.3 Comparison with the Approach of Abadi and Lamport

There are various differences between my approach and that of Abadi and Lamport. A major difference is that I deal with branching time notions because in the context of mechanical verification they provide certain advantages, as outlined above. However, in order to simplify the comparison, in this section I consider only the linear time aspects of my results.

There are differences in how stuttering is dealt with; namely, Abadi and Lamport allow infinite stuttering, whereas I do not. Consider the example of pipelined machine verification. Using the Abadi and Lamport approach, we would define the instruction set architecture using a state machine, say where every component is externally visible. By definition, the property generated by the state machine includes infinite stuttering, *e.g.*, it includes the behavior where nothing happens. Thus, a supplementary property would be used to rule out such behaviors by requiring that non-stuttering steps are eventually taken, a liveness property. In contrast, in my approach, every step of the transition system modeling the instruction set architecture corresponds to the execution of an instruction, with the stuttering being handled by the definition of stuttering simulation. Notice that no supplementary property is required. In addition, the condition that a pipelined machine makes progress is now a safety property, because the number of steps required is bounded by the number of stages in the pipeline [16].

Lamport and Abadi require that systems have the same externally visible states. They make the point that one cannot say whether the value 11111100 corresponds to -3 without knowing how to interpret a sequence of bits as an integer. They go on to say that given such an interpretation, they can translate the externally visible states to the appropriate representation. In my case, instead of having a separate interpretation phase, I allow refinement maps to alter the labels of states directly. I have found that in practice this extra power is necessary. For example, when proving that a pipelined machine implements the instruction set architecture, I have used refinement maps that either modify the value of the program counter (when using my "commit" approach to correctness) or modify the register file and memory (using the Burch and Dill "flushing" approach to correctness) [16]. The point is that when using my commit approach to correctness, if we consider the program counter to be externally visible then we cannot use the Abadi and Lamport approach to prove that a pipelined machine implements the instruction set architecture. Similarly, when using the Burch and Dill approach, if we consider the register file or memory to be externally visible, then we cannot use the Abadi and Lamport approach to prove that a pipelined machine implements the instruction set architecture.

The refinement mappings of Abadi and Lamport are required to preserve the supplementary property of the specification. As they point out, this is not a local condition, but one can apply local methods such as well-founded induction for the proof. Unfortunately, they do not provide any guidance on constructing such arguments. In my case, the proof of proposition 2 (if B is an STS, then B is a WFS) shows how to construct the appropriate well-founded relations and measure functions, *rankt* and *rankl*. The proof also shows that two measure functions, one from pairs of states and one from triples of states to the naturals, are enough regardless of the transition systems involved.

Finally, my theorems are stronger than the ones given by Abadi and Lamport. For example, they show that even when $\mathcal{S}$ is not internally continuous a refinement map exists to show that $\mathcal{I}$ satisfies the safety property specified by $\mathcal{S}$. They continue "We do not know if anything can be said about proving arbitrary liveness properties." Since my refinement theorems apply to any systems, a simple corollary is that, with my approach, refinement maps can always be used to prove both safety and liveness properties. This is something that we used in [17] where we used theorem proving to reduce an infinite-state system to a finite-state system in such a way that stuttering-insensitive properties, including liveness, were preserved. We then model checked the reduced system and were able to lift the results to the original system.

5 Conclusions

I have introduced compositional notions of refinement for stuttering simulation. I have shown that if one system refines another in the branching time framework, then a refinement map always exists, without relying on any of the conditions present in the approach taken by Abadi and Lamport, *e.g.*, machine closure, finite invisible nondeterminism, internally continuity, the use of history and prophecy variables, etc. I also showed that refinement maps always exist in the linear time framework, subject only to the use of oracle variables.

My main motivation is the mechanical verification of systems. Notions of refinement based on stuttering bisimulation have proved useful for this purpose [17, 16]. However, stuttering bisimulation is applicable only in limited contexts, as usually specifications contain more nondeterminism than implementations. Thus, I expect that stuttering simulation will turn out to be more useful than stuttering bisimulation.

References

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

[2] P. C. Attie. Liveness-preserving simulation relations. In *ACM Symposium on the Principles of Distributed Computing (PODC)*, pages 63–72, May 1999.

[3] T. Basten. Branching bisimilarity is an equivalence indeed. *Information Processing Letters*, 58(3):141–147, 1996.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, vol. 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[5] K. Engelhardt and W.-P. de Roever. Generalizing Abadi & Lamport's method to solve a problem posed by A. Pnueli. In *FME '93: Industrial-Strength Formal Methods*, vol. 670 of *LNCS*, pages 294–313. Springer-Verlag, 1993.

[6] R. J. v. Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.

[7] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[8] W. Hesselink. Eternity variables to simulate specifications. In *Proceedings Mathematics of Program Construction, Dagstuhl, 2002*, vol. 2386 of *LNCS*, pages 117–130. Springer-Verlag, 2002.

[9] B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. *Distributed Computing*, 12(2–3):129–149, 1999.

[10] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[11] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[12] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[13] K. Kunen. *Set Theory - An Introduction to Independence Proofs*, vol. 102 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1980.

[14] L. Lamport. What good is temporal logic? *Information Processing*, 83:657–688, 1983.

[15] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations – part I: untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[16] P. Manolios. Correctness of pipelined machines. In *Formal Methods in Computer-Aided Design–FMCAD 2000*, vol. 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[17] P. Manolios, K. Namjoshi, and R. Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In *Computer-Aided Verification–CAV '99*, vol. 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.

[19] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, vol. 1346 of *LNCS*, pages 284–296, 1997.

[20] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[21] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, vol. 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[22] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, 1977. IEEE.

[23] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings of 12th International Colloquium on Automata, Languages and Programming*, vol. 194 of *LNCS*, pages 15–32. Springer-Verlag, 1985.

[24] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Dec. 1999.

[25] J. Sawada. Verification of a simple pipelined machine model. In Kaufmann et al. [10], pages 137–150.

[26] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *STOC: ACM Symposium on Theory of Computing (STOC)*, pages 1–9, 1973.