A Refinement-based Approach to Reason About Optimized Reactive Systems

Mitesh Jain

Abstract

We introduce refinement-based methods for analyzing the correctness of reactive systems. We propose *skipping refinement*, a new notion of refinement that extends the domain of applicability of refinement to include optimized reactive systems, systems that can run "faster" than their abstract high-level specifications. We develop a theory of skipping refinement and associated proof methods that are amenable to mechanized reasoning using existing verification tools. We also show that refinement can be used as part of an effective simulation-based testing methodology for reactive systems. We evaluate our work using several case studies.

1 Introduction

The purpose of this thesis is to investigate a refinement-based approach for analyzing the correctness of reactive systems using formal verification and simulation-based testing. Reactive systems are non-terminating systems that maintain an ongoing interaction with their environment. Examples of such systems include safety-critical systems like automotive controllers and communication networks, and omnipresent systems like microprocessors and operating systems. Reactive systems differ from transformational systems; their behaviors cannot be formalized and reasoned about using relations between the input states and output states. Since reactive systems are not expected to terminate, we cannot refer to their final, output states. Moreover, the need to analyze the ongoing interaction of a reactive system with its environment enforces a view of the behavior less abstract than the relational view of the behavior of a transformational system. As a result, a formal description of behaviors of a reactive system is based on infinite computations.

In the refinement-based reasoning, a high-level abstract system $\mathcal{A}$ serves as a specification for a concrete system $\mathcal{C}$ described at a lower level of abstractions. We say that $\mathcal{C}$ *refines* (implements) $\mathcal{A}$ iff all observable behaviors of $\mathcal{C}$ are behaviors allowed by $\mathcal{A}$. But, the abstract system and the concrete system are described at different level of abstractions. So, what is an appropriate notion of refinement to relate their behaviors? Observe that a concrete system $\mathcal{C}$ describes the behavior in more detail than the abstract system $\mathcal{A}$. So it is often that $\mathcal{C}$ requires multiple steps to perform a task that is described in a single step in $\mathcal{A}$. This phenomenon is commonly known as *stuttering*. An appropriate notion of refinement must directly account for such stuttering behavior. On the other hand, drive to build ever more efficient systems has led to highly-optimized implementations. A single step in such an optimized implementation performs a task that is described as multiple steps in the abstract system. A notion of refinement that only accounts for stuttering are unduly

restrictive and prohibits analysis of such optimized implementations. To illustrate the inadequacy of existing notion of refinement, let us consider the example of a common compiler optimization.

Example 1 (Superword Compiler Optimization). An effective way to improve the performance of multimedia programs running on modern commodity architectures is to exploit Single-Instruction Multiple-Data (SIMD) instructions (*e.g.*, the SSE/AVX instructions in x86 microprocessors). Compilers analyze programs for *superword level parallelism*, and when possible replace multiple scalar instructions with a compact SIMD instruction that concurrently operates on multiple data [33].

$$\begin{array}{ccccc} a & = & b & + & c \\ d & = & e & + & f \end{array} \rightarrow \begin{bmatrix} a \\ d \end{bmatrix} = \begin{bmatrix} b \\ e \end{bmatrix} +_{SIMD} \begin{bmatrix} c \\ f \end{bmatrix}$$

$$\begin{array}{ccccc} u & = & v & \times & w \\ x & = & y & \times & z \end{array} \rightarrow \begin{bmatrix} u \\ x \end{bmatrix} = \begin{bmatrix} v \\ y \end{bmatrix} \times_{SIMD} \begin{bmatrix} w \\ z \end{bmatrix}$$

Figure 1: Superword Parallelism

The compilation is correct only if the semantics of the source program, consisting of only scalar instructions, is preserved by the optimized compiled program consisting of mix of scalar and SIMD instructions. However, it is not possible to directly prove that the compiled program *refines* the source program using existing notions of refinement. This is because when the compiled program executes a SIMD instruction, a step neither corresponds to a stuttering step nor corresponds to a single step of the source program.

The above example is a representative of a common occurrence when engineering an optimized system: as a result of optimizations an implementation runs faster than the specification. For example, in order to reduce memory latency and effectively utilize memory bandwidth, memory controllers often buffer requests to memory. The pending requests in the buffer are analyzed for address locality and then at some time in the future, multiple locations in the memory are read and updated simultaneously. Similarly, to improve instruction throughput, superscalar processors fetch multiple instructions in a single cycle. These instructions are analyzed for instruction-level parallelism (*e.g.*, the absence of data dependencies) and, where possible, are executed in parallel, leading to multiple instructions being retired in a single cycle. In both these examples, in addition to stuttering, a single step in the implementation may perform the work of multiple abstract steps, *e.g.*, by updating multiple locations in memory and retiring multiple instructions in a single cycle. Existing notions of refinement only account for stuttering and are not appropriate for reasoning about such optimized systems. In [30], we introduced *skipping simulation*, a new notion of correctness, that directly accounts for skipping (and stuttering) exhibited by optimized reactive systems. Skipping can be thought of as the dual of stuttering: stuttering allows us to "stretch" executions of the specification system and skipping allows us to "squeeze" them. Based on skipping simulation, we develop a theory of skipping refinement and show that it enjoys several useful algebraic properties. In particular, we show that skipping refinement is compositional and, therefore, aligns with the stepwise refinement verification methodology [52, 14]. Notice since the new notion directly accounts for skipping, it enables us to keep the specification as-simple-as-possible and not change it as a result of domain-specific optimizations introduced in an implementation.

Refinement-based formal verification We are developing the theory of skipping refinement with an aim to mechanically analyze optimized reactive systems. However, typically a reactive system is non-terminating and to prove its correctness based on skipping refinement requires us to reason about quantifiers over infinite computations: we have to show that every observable behavior of a concrete system is an observable behavior of its specification up to finite skipping. The support for such reasoning is rather limited in existing tools for automated reasoning. We are developing proof methods that avoid reasoning about infinite computations and enable us to prove skipping refinement by checking only local properties *i.e.*, properties involving states and their successors. Our proof methods are widely applicable, and can be used to reason about reactive systems with arbitrary state space sizes and arbitrary non-determinism. Together, the theory of skipping refinement and the proof methods provide a general framework to effectively analyze large class of reactive systems.

Refinement-based testing Formal verification techniques provide guarantees about correctness of a system, but in spite of great advancements they are often intractable for large, complex system designs. On the other hand, dynamic verification based on testing, though incomplete, scales well for systems of arbitrary complexity. We are actively working on a new approach to dynamic verification, based on refinement. The idea behind our method is simple. We compile a refinement conjecture into a runtime check performed during simulation. This allows us to check for functional correctness during testing using only this one check. Our approach addresses several challenges facing industry [22, 12, 10, 20]. First, we target functional correctness. According to a recent study by Foster [23], 50% of flaws resulting in respins are due to logic or functional correctness flaws. Second, it is difficult to determine if the set of properties and tests under consideration is complete: the Foster study shows that over 40% of functional flaws are due to incomplete or incorrect specifications. Third, defining oracles for tests is expensive and error-prone: the Foster study shows that verification engineers spend 24% of their time creating tests and running simulations. Finally, properties are defined in terms of low-level designs, so modifications during the design cycle lead to, possibly significant, changes to the properties being tested. Furthermore, similar to property-based testing, the refinement conjecture can be effectively analyzed both with a simulation-based workflow and a formal verification workflow.

Proposal Summary The working hypothesis of this proposal is that refinement-based reasoning can effectively analyze the correctness of a large class of reactive systems. To support the hypothesis, (1) we introduce skipping simulation, a new notion of correctness that directly accounts for both finite stuttering and finite skipping. This extends the domain of applicability of existing refinement-based methods to reason about optimized reactive systems; (2) we develop a theory of refinement that supports a top-down stepwise refinement verification methodology; (3) we develop sound and complete proof methods to prove skipping refinement using existing verification tools. The completeness result also answers in affirmative a fundamental question in the theory of refinement: given a concrete system that *implements* an abstract system, does there exists a refinement map that can be used to prove it?; and (4) we introduce and develop a novel refinement-based testing methodology.

The research outlined in the proposal is interwoven with case studies to evaluate its effectiveness.

We evaluate the cost and the benefit of our methodology on a broad range of hardware and software systems using interactive and automated verification tools.

2 Related Work

We briefly survey the related work and compare the notion of skipping refinement on following criteria: notion of correctness, local reasoning and existence of refinement maps, and applications of refinement methodology to analyze correctness of hardware and software systems. Finally, we briefly review simulation-based testing methodology. This is not an exhaustive survey, but only an overview of the work which helped us comprehend the vast area of program equivalence and refinement, and the important questions to consider in development of a theory of refinement. In particular, our work on skipping refinement is inspired by [37] and can be viewed as its extension.

Notions of correctness Notions of correctness for reasoning about reactive systems have been widely studied. We refer the reader to excellent surveys on this topic [48, 50, 35]. Milner introduced the notion of *simulation* to precisely define when a system (deterministic, possibly non-terminating) may be considered as a *realization* (implementation) of another system [44]. However, notion of refinement based on (bi)simulation are too strong and do not directly account for stuttering behavior, a common phenomenon when relating two systems described at different levels of abstraction. *Weak (bi)simulation* does account for the stuttering behavior, but it allows an implementation to exhibit infinite stuttering, and hence, cannot distinguish a deadlock from divergence. Notions of refinement based on *stuttering (bi)simulation* [38] remedy this by disallowing an implementation that can stutter infinitely. However, even these notions are too strong to reason about a large class of optimized reactive systems that can run faster than their corresponding high-level specifications. Skipping simulation directly accounts for finite stuttering and finite skipping and is an appropriate notion for analyzing such optimized implementations. To the best of our knowledge, no similar notion of refinement has been reported in the literature. An important class of optimization that is not directly accounted by skipping refinement occurs when a concrete system is allowed to enforce a less restrictive ordering between observable actions than the ordering enforced by the abstract system.

Local Reasoning and existence of refinement maps A key ingredient in refinement proofs is the notion of a refinement map, a function that maps a concrete state to an abstract state and reduces the problem of reasoning about infinite behaviors to local reasoning, *i.e.*, about states and their successors. Then a fundamental question about the technique is the following: given a concrete system $\mathcal{C}$ that *implements* an abstract system $\mathcal{A}$, under what constraints on the systems does there exist a refinement map that can be used to prove refinement? The answer essentially determines the domain of applicability of the refinement-based technique to reason about reactive systems. Abadi and Lamport [8] studied the problem of existence of a refinement map in the linear-time framework, where a behavior of a system is described as an infinite sequence of states and *implements* is defined as language containment. They showed that if the system satisfies a number of conditions (in particular, finite non-determinism), one can add *history* and *prophecy* variables to $\mathcal{C}$ and construct a refinement map to prove that $\mathcal{C}$ implements $\mathcal{A}$. Klarlund also independently studied

the problem in the linear-time framework, and showed that in the case when $\mathcal{A}$ has bounded (finite) non-determinism and $\mathcal{C}$ is a *complete*[1] automaton, one can prove language containment without adding history and prophecy variables using *ND measures*, a mapping from a state of the system to a set of states of the system [31]. Manolios studied the problem in the branching-time framework where the behavior of a system is defined as a computation tree and *implements* is defined using the notion of stuttering simulation [38]. It was shown that in the branching-time framework, a refinement map always exists without depending on any of the conditions on the system present in the Abadi and Lamport approach. We also develop the theory of skipping refinement in the branching-time framework and show that a refinement map always exists. As a result, skipping refinement is a widely applicable notion of correctness.

Applications Correctness of microprocessors has been extensively studied and several variants of correctness theorems have been proposed [51, 28, 7, 49]. These variants can be broadly classified on the basis of whether they support (1) a deterministic or nondeterministic abstract system, (2) a deterministic or nondeterministic concrete system, and (3) the kinds of refinement maps used. In comparison, the theory of skipping refinement provides a general framework for analyzing both deterministic and nondeterministic systems and any choice of refinement map; in all the above cases, we prove the same theorem. We argue that a uniform notion of correctness crucially increases the trust and also eases the verification effort.

Refinement methodology has also been used to verify the correctness of programs and program transformations. Several back-end compiler transformations are proven correct in CompCert [34]. A transformation is correct if it is semantic preserving, *i.e.*, if a behavior (sequence of observable events) of a compiled program is a behavior of the source program. The correctness of transformation in CompCert is proved using *star simulation* under the assumptions that the source and the target languages are deterministic and transformation does not result in skipping observable events. Namjoshi et. al. use an approach similar to translation validation and a notion of correctness based on stuttering simulation, a notion strictly weaker than skipping simulation, to analyze correctness of several compiler transformations [46]. In a subsequent paper, they describe a novel application of refinement methodology to improve the effectiveness of compiler optimizations in LLVM. The witness for refinement that is used to prove a transformation correct can often also be used to transform an invariant of the source program, discovered using a static analyzer, to an invariant of the target program [24].

Recently, refinement in conjunction with Floyd-Hoare style verification methodology has also been successfully applied to verify the correctness of practical distributed systems [27] and a general-purpose operating system microkernel [32]. We believe that skipping refinement, combined with the proof methods that are amenable for automated reasoning, will further extend the domain of applicability of the refinement methodology.

Refinement-testing Dynamic validation consists of monitoring an individual execution (finite) trace of a system for a violation of the specification. It has been widely studied in the literature [26, 21] and used in the industry [23, 12, 16, 45, 10]. The spectrum of approaches to monitoring an execution trace ranges from checking a *predicate assertion* for a violation at a single program

[1] An automaton $\mathcal{A}$ is complete if every state of $\mathcal{A}$ is some run of $\mathcal{A}$.

location, to checking a *temporal assertion* for a violation at multiple program locations, and to checking an *invariant* at all program locations. Temporal assertions are specified using (a subset of) linear temporal logic [5, 6] and compiled to state machines that are simulated along with the design. In general, these state machines can be exponential in the size of the temporal logical formula and can result in significant overhead during simulation. To our knowledge, we are the first to propose and systematically study the refinement-based methodology for dynamic verification. The local proof methods for proving refinement enable us to design effective checkers/monitors with low overheads.

3 Notation

Function application is sometimes denoted by an infix dot "." and is left-associative. For a binary relation R, we often write xRy instead of $(x, y) \in R$. The composition of two relations R and S, is denoted by $R;S$, which is equivalent to $S \circ R$. The composition of relation R with itself i times (for $0 < i \leq \omega$) is denoted as R^i ($\omega = \mathbb{N}$ and is the first infinite ordinal). Given a relation R and $1 < k \leq \omega$, $R^{<k}$ denotes $\bigcup_{1 \leq i < k} R^i$ and $R^{\geq k}$ denotes $\bigcup_{\omega > i \geq k} R^i$. $\uplus$ denotes the disjoint union operator. Quantified expressions are written as $\langle Qx : r : p \rangle$, where Q is the quantifier (*e.g.*, $\exists, \forall$), x is the bound variable, r is an expression that denotes the range of x (*true* if omitted), and p is the body of the quantifier.

4 Running Example

Example 2 (Discrete-time Event Scheduler). In this example, we describe a discrete-time event simulation (DES) system to illustrate the notion of skipping refinement-testing. An abstract high-level specification of DES is described as follows. Let E be set of events and V be set of state variables. Then a state of abstract DES is a three-tuple $\langle t, Sch, A \rangle$, where t is a natural number denoting current time; Sch is a set of pairs (e, t_e), where $e \in E$ is an event scheduled to be executed at time $t_e \geq t$; A is an assignment to variables in V. The transition relation for the abstract DES system is defined as follows. If at time t there is no $(e, t) \in Sch$, *i.e.*, there is no event scheduled to be executed at time t, then t is incremented by 1. Else, we (nondeterministically) choose and execute an event of the form $(e, t) \in Sch$. The execution of event may result in modifying A and also adding finite number of new pairs (e', t') in Sch. We require that $t' > t$. Finally execution involves removing the executed event (e, t) from Sch.

Now, consider an optimized, concrete implementation of the abstract DES system. As before, a state is a three-tuple $\langle t, Sch, A \rangle$. However, unlike the abstract system which just increments time by 1 when no events are scheduled for the current time, the optimized system uses a priority queue to find the next event to execute. The transition relation is defined as follows. An event (e, t_e) with the minimum time is selected, t is updated to t_e and the event e is executed, as in the abstract DES. Notice that when no events are scheduled for execution at the current time, the optimized implementation of the discrete-time event simulation system can run faster than the abstract specification system by *skipping* over states of the abstract DES system. This is not a stuttering step as it results in an observable change in the state of the concrete DES system (t is update to t_e). Also, it does not correspond to a single step of the specification. Therefore, it is not

possible to prove that the implementation *refines* the specification using notions of refinement that only allow stuttering because that just is not true. But, intuitively, there is a sense in which the optimized DES system *does* refine the abstract DES system.

5 Skipping Simulation

We first introduce the notion of skipping simulation using a generic model of a labeled transition system. A transition system model of a reactive system captures the concept of a state, what is observable in a state, and atomic transitions that modify state during the course of a computation. Any system with a well defined operational semantics can be mapped to a labeled transition system [47]. Hence it is well-suited to describe and analyze a large class of reactive systems. Furthermore, since our exposition is purely semantic it is agnostic to a particular programming language used to describe a system.

Definition 1 (Labeled Transition System)**.** A labeled transition system (TS) is a structure $\langle S, \rightarrow, L\rangle$, where S is a non-empty (possibly infinite) set of states, $\rightarrow \subseteq S \times S$, is a left-total transition relation (every state has a successor), and L is a labeling function whose domain is S.

A path is a sequence of states such that for adjacent states, s and u, $s \rightarrow u$. A path σ starting at state s is a *fullpath*, denoted by $fp.\sigma.s$, if it is infinite.

A transition system is parameterized with a domain of observation and L tells us what is observable in a state. Notice that we do not place any restriction on the state space sizes and the branching factor of the transition relation, and both can be of arbitrary infinite cardinalities. This generality is helpful in modeling systems that exhibit unbounded non-determinism, for example, the random assignment statement $x :=?$, which sets x to an arbitrary integer [11].

Our definition of skipping simulation relation is based on a notion of *matching*, which we define below. Informally, we say that a fullpath σ *matches* a fullpath δ under a binary relation B, if the fullpaths can be partitioned into non-empty, finite segments such that all states in a segment of σ are related to the first state in the corresponding segment of δ.

Definition 2 (Match)**.** Let INC be the set of strictly increasing sequences of natural numbers starting at 0. Given a fullpath σ, the i^{th} segment of σ with respect to $\pi \in INC$, written ${}^{\pi}\sigma^{i}$, is given by the sequence $\sigma(\pi.i), ..., \sigma(\pi.(i+1)-1)$. For $\pi, \xi \in INC$ and relation B, we define

$$corr(B, \sigma, \pi, \delta, \xi) \equiv \langle \forall i \in \omega :: \langle \forall s \in {}^{\pi}\sigma^{i} :: sB\delta(\xi.i)\rangle\rangle \ \ and$$
$$match(B, \sigma, \delta) \equiv \langle \exists \pi, \xi \in INC :: corr(B, \sigma, \pi, \delta, \xi)\rangle.$$

In Figure 2, we illustrate our notion of matching using DES (Example 2). Let the set of state variables V, be $\{v_1, v_2\}$ and the set of events, Sch, be $\{(e_1, 0), (e_2, 2)\}$, where e_i increments variable v_i by 1. In the figure, σ is a fullpath of the concrete system and δ is a fullpath of the abstract system. (We only show a prefix of the fullpaths.) The other parameter for $match$ is B, which is just the identity relation. In order to show that $match(B, \sigma, \delta)$ holds, we have to find π, ξ satisfying the definition. In the figure, we separate the partitions induced by our choice for π, ξ using $--$ and connect elements related by B with —-. Since all elements of a σ partition are related to the first element of the corresponding δ partition, $corr(B, \sigma, \pi, \delta, \xi)$ holds, therefore, $match(B, \sigma, \delta)$ holds.

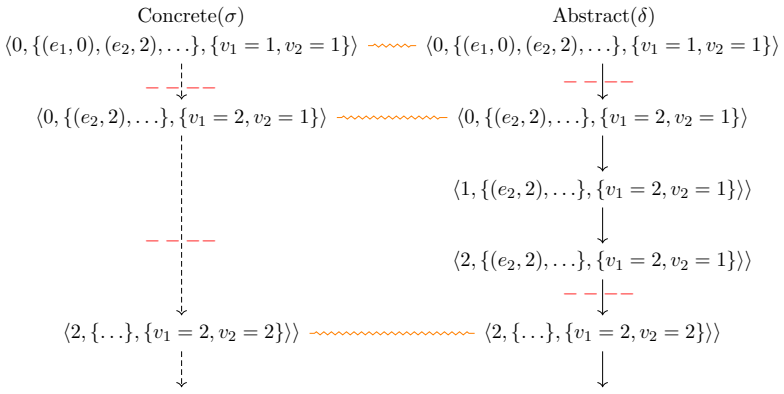


Figure 2: Event simulation system

Given a transition system (TS) $\mathcal{M} = \langle S, \rightarrow, L\rangle$, a relation $B \subseteq S \times S$ is a skipping simulation, if for any $s, w \in S$ such that sBw, s and w are identically labeled and any fullpath starting at s can be matched by some fullpath starting at w.

Definition 3 (Skipping Simulation)**.** $B \subseteq S \times S$ is a skipping simulation (SKS) on TS $\mathcal{M} = \langle S, \rightarrow, L\rangle$ *iff* for all s, w such that sBw, both of the following hold.

(SKS1) $L.s = L.w$

(SKS2) $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B, \sigma, \delta)\rangle\rangle$

Note that skipping simulation differs from the stuttering simulation [38]. The later is inadequate to reason about a concrete system that can run "faster" than the abstract system. In fact it can be shown that skipping simulation is a strictly weaker notion than stuttering simulation.

5.1 Properties of SKS

Next we show that skipping simulation enjoys useful algebraic properties. In particular, there is a greatest SKS and that the reflexive transitive closure of an SKS is an SKS. The transitivity property is useful for developing a compositional theory of refinement.

Lemma 1. For any TS $\mathcal{M}$, there is a greatest SKS on $\mathcal{M}$.

Lemma 2. If P and Q are SKS's on TS $\mathcal{M}$, so is $R = P; Q$.

Lemma 3. The reflexive transitive closure of an SKS is an SKS.

Theorem 4. Given a transition system $\mathcal{M}$, there is a greatest SKS on $\mathcal{M}$, which is a preorder.

6 Skipping Refinement

We now use the notion of skipping simulation, which is defined in terms of a single transition system, to define skipping refinement, a notion that relates two transition systems: an *abstract* transition system and a *concrete* transition system. The notion of skipping refinement is parameterized by a *refinement map*, a function that maps a state of the concrete system to a state of the abstract

system. Refinement maps, along with the labeling function, tell us what is observable at a concrete state from the viewpoint of an abstract system. If a concrete system is a skipping refinement of the abstract system, then the observable behavior of the former are the observable behavior of the later modulo skipping (which includes stuttering). For example, in our running example of the discrete-time event simulator, if the refinement map is the identity function, then any behavior of the optimized implementation is a behavior of the abstract system modulo skipping.

Let $\mathcal{M}_{\mathrm{A}} = \langle S_A, \xrightarrow{A}, L_A \rangle$ and $\mathcal{M}_{\mathrm{C}} = \langle S_C, \xrightarrow{C}, L_C \rangle$ be transition systems and let $r \colon S_C \to S_A$ be a *refinement map*. Notice that we place no restrictions on the refinement map.

Definition 4 (Skipping Refinement)**.** We say $\mathcal{M}_C$ is a *skipping refinement* of $\mathcal{M}_A$ with respect to refinement map r, written $\mathcal{M}_C \lesssim_r \mathcal{M}_A$, if there exists a relation $B \subseteq S_C \times S_A$ such that all of the following hold.

1. $\langle \forall s \in S_C :: sBr.s \rangle$ *and*
2. B is an SKS on $\langle S_C \uplus S_A, \xrightarrow{C} \uplus \xrightarrow{A}, \mathcal{L} \rangle$
   where $\mathcal{L}.s = L_A(s)$ for $s \in S_A$, and $\mathcal{L}.s = L_A(r.s)$ for $s \in S_C$.

In the above definition, it helps to think of $\mathcal{M}_A$ and $\mathcal{M}_C$ as the abstract and the concrete system respectively. There are often other considerations *e.g.*, it might be that $\mathcal{M}_A$ and $\mathcal{M}_C$ have certain states that are "initial". In this case, one might wish to show that initial states in $\mathcal{M}_C$ are mapped by the refinement map to initial states in $\mathcal{M}_A$. There are no restrictions on refinement maps, in particular, it is not restricted to a simple *projection function* [8] that projects the observable component of a concrete state. The generality of refinement map is often useful. For example, in the case studies considered in this proposal a simple refinement map that is a projection function would not have sufficed. However, by choosing a complicated refinement map, one can bypass the verification problem. Thus to prove that the concrete system is a skipping refinement of the abstract system, one should be prudent in taking advantage of the flexibility in choice of refinement map.

The next theorem shows that skipping refinement is compositional.

Theorem 5 (Composition)**.** If $\mathcal{M}_1 \lesssim_f \mathcal{M}_2$ and $\mathcal{M}_2 \lesssim_g \mathcal{M}_3$ then $\mathcal{M}_1 \lesssim_{f;g} \mathcal{M}_3$.

This allows us to use skipping simulation in a stepwise refinement approach, a verification methodology that can significantly increases scalability and reduce verification times [42]. We start with the simplest, high-level abstract system $\mathcal{M}_A$ and progressively design a sequence of intermediate lower level systems leading to the most concrete system. Formally, if we can establish that $\mathcal{M}_A = \mathcal{M}_0 \lesssim_{r_0} \mathcal{M}_1 \lesssim_{r_1} \ldots \lesssim_{r_{n-1}} \mathcal{M}_n = \mathcal{M}_C$, we can infer from Theorem 5, that $\mathcal{M}_A \lesssim_r \mathcal{M}_C$, where $r = r_0; r_1; \ldots; r_{n-1}$.

7 Automated Reasoning

An appropriate notion of refinement is only part of the story. Our aim is to develop a proof method that advances state-of-the-art in mechanical reasoning of optimized reactive systems. However, using Definitions 4 to prove a concrete system $\mathcal{M}_C$ is skipping refinement of an abstract system $\mathcal{M}_A$, requires us to show that for any fullpath (an infinite sequence of states) in $\mathcal{M}_C$ we can find a

"matching" fullpath in $\mathcal{M}_A$. Reasoning about the existence of infinite sequences can be problematic using automated tools. In order to avoid such reasoning, we introduce the notion of well-founded skipping simulation. This notion allows us to prove skipping refinement by checking only "local" properties, *i.e.*, properties involving states and their successors. The intuition (Figure 3) is, for any pair of states s, w, which are related by a binary relation (denoted in orange in Figure 3) and a state u such that $s \rightarrow u$, there are four cases to consider: (a) either we can match the move from s to u right away *i.e.*, there is a v such that $w \rightarrow v$ and u is related to v, or (b) there is stuttering on the left, or (c) there is stuttering on the right, or (d) there is skipping on the right.

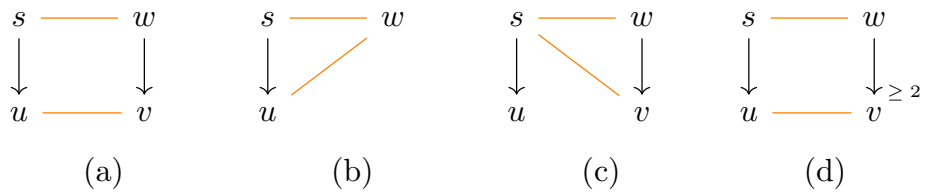


Figure 3: Well-founded skipping simulation

Definition 5 (Well-founded Skipping Simulation)**.** $B \subseteq S \times S$ is a well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

(WFSK1) $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$

(WFSK2) There exist functions, $rankt \colon S \times S \rightarrow W$, $rankl \colon S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded such that
$\langle \forall s, u, w \in S : s \rightarrow u \wedge sBw$:
(a) $\langle \exists v \colon w \rightarrow v \colon uBv \rangle \vee$
(b) $(uBw \wedge rankt(u, w) \prec rankt(s, w)) \vee$
(c) $\langle \exists v \colon w \rightarrow v \colon sBv \wedge rankl(v, s, u) < rankl(w, s, u) \rangle \vee$
(d) $\langle \exists v : w \rightarrow^{\geq 2} v \colon uBv \rangle \rangle$

In the above definition, notice that condition (2d) requires us to check that there exists a v such that v is *reachable* from w and uBv holds. Reasoning about reachability is not local in general. However, it is often the case that we can reason about reachability using local methods because the number of abstract steps that a concrete step corresponds to is bounded by a constant. As an example, the number of scalar instructions that a SIMD instruction can execute in parallel is a constant that is determined early in the design. However, notice in case the concrete system skips large or possibly unbounded (but finite) number of abstract steps, this approach will in fact require us to reason about reachability. We discuss this in more detail in proposed work (§10.)

Next, we show that the notion of well-founded skipping simulation is equivalent to SKS and can be used as a sound and complete method to prove skipping refinement.

Theorem 6 (Soundness [30])**.** If B is an WFSK, then B is an SKS.

Theorem 7 (Completeness [30])**.** If B is an SKS, then B is an WFSK.

A basic question in a theory of refinement is whether refinement maps exist: if a concrete system implements an abstract system, does there exists a refinement map that can be use to prove it? Abadi and Lamport [8] showed that in the linear-time framework, a refinement map exists provided the systems satisfy a number of complex conditions. In [37], it was shown that for stuttering refinement, a branching-time notion, the existence of refinement maps does not depend on any of the conditions found in the work of Abadi and Lamport and that the result can be extended to the linear-time case [38]. The completeness result (Theorem 7) shows that a refinement map always exists and like in the stuttering refinement, its existence does not depend on any conditions on the system.

8 Experimental Evaluation

We evaluate the applicability of the theory of skipping refinement using three simple case studies. Though simple, these models are sufficient to exhibit limitations of existing notions of correctness, scalability issues in current verification tools, and how skipping refinement and the associated proof method addresses these limitations. The systems and the results that we provide here are described and analyzed in more detail in the relevant papers [30, 29].

JVM-inspired stack machine: We define BSTK, a simple hardware implementation of part of the Java Virtual Machine (JVM) [25]. BSTK models an instruction memory, an instruction buffer, and a stack. It supports a small subset of JVM instructions, including *push*, *pop*, *top*, and *nop*. STK is the high-level specification with respect to which we verify the correctness of BSTK. STK fetches an instruction from the instruction memory, executes it, increases the program counter and possibly modifies the stack. The state of BSTK is similar to STK, except that it also includes an instruction buffer. The capacity of the instruction buffer is a constant positive integer. BSTK fetches an instruction from the instruction memory, and as long as the fetched instruction is not *top* and the instruction buffer is not full, it enqueues it to the end of the buffer and increments the program counter. If the fetched instruction is *top*, or if the buffer is full, the machine executes all buffered instructions in the order they were enqueued, thereby draining the buffer and obtaining a new stack.

Optimized Memory controller: Modern microprocessors operate at a higher clock frequency than their main memories. Hence, it is essential for a memory controller, the interface between the CPU and main memory, to buffer requests and responses and synchronize communication between the CPU and memory. Moreover, current memory controllers implement optimizations to maximize available memory bandwidth utilization. We define a simple memory controller, OptMEMC, which fetches a memory request from location *pt* in a queue of CPU requests (*reqs*). It enqueues the fetched request in the request buffer and increments *pt* to point to the next CPU request in *reqs*. If the fetched request is a *read* or the request buffer is full, then before enqueuing the request into *rbuf*, OptMEMC first analyzes the request buffer for consecutive write requests to the same address in the memory. If such a pair of writes exists in the buffer, it marks the older write requests in the request buffer as redundant. Then it executes all the requests in the request buffer except the marked (redundant) ones. Requests in the buffer are executed in the order they were enqueued.

MEMC is a high-level specification with respect to which we verify the correctness of OptMEMC. It simply fetches a request from *reqs* and services each request atomically in the sequence they arrive.

Superword Compilation optimization: For this case study we verify the correctness of a compiler transformation from a source language containing only scalar instructions to a target language containing both scalar and SIMD instructions. We model the transformation as a function that is given a program in the source language and generates a program in the target language. We use the translation validation approach to compiler correctness and prove that the target program implements the source program [15].

Results

For each case study, we define an appropriate refinement map and we prove the *same* correctness theorem: the implementation is a skipping refinement of the specification. Notice that each of the optimized systems above, BSTK, OptMEMC, and the compiled program, can skip multiple steps of the abstract system. Our goals were to evaluate the specification costs of using skipping refinement as a notion of correctness, and to determine the impact that the use of skipping refinement has on state-of-the-art verification tools in terms of capacity and verification times. BAT files, corresponding AIGs, ACL2s models, and ACL2s proof scripts are publicly available [4].

Model Checking: The finite state models (of various sizes) of the BSTK and OptMEMC were developed and compiled to sequential AIGs using the BAT tool [43][3], and then analyzed using TIP, IIMC, BLIMC, and SUPER_PROVE model-checkers [1]. SUPER_PROVE and IIMC are the top performing model-checkers in the single safety property track of the Hardware Model Checking Competition 2013 [1]. We chose TIP and BLIMC to cover tools based on temporal decomposition and bounded model-checking. To evaluate the computational benefits of skipping refinement, we created a benchmark suite that had anywhere from 24K gates and 500 latches to 2M gates and 23K latches. We use a machine with an Intel Xeon X5677 with 16 cores running at 3.4GHz and 96GB main memory. The timeout limit for model-checker runs is set to 900 seconds.

We compare the cost of proving correctness using skipping refinement with the cost of using input-output equivalence: if the specification and the implementation systems start in equivalent initial states and get the same inputs, then if both systems terminate, the final states of the systems are also equivalent. We chose I/O equivalence since that is the most straightforward way of using existing verification tools to reason about our case studies. We cannot use existing notions of refinement because they do not allow skipping and, therefore, are not applicable. Since skipping simulation is a stronger notion of correctness than I/O equivalence, skipping proofs provide more information, *e.g.*, I/O equivalence holds even if the concrete system diverges, but skipping simulation does not hold and would therefore catch such divergence errors.

In Fig. 4, we plot the running times for the four model-checkers used. The x-axis represents the running time using I/O equivalence and y-axis represents the running time using skipping refinement. A point with $x = \text{TO}$ indicates that the model-checker timed out for I/O equivalence

[3]The BAT tool was modified to generate sequential AIG's.

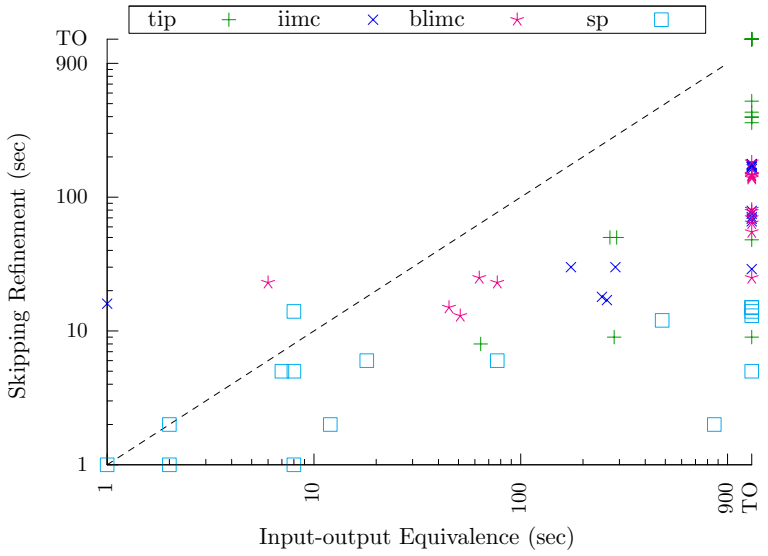


Figure 4: Performance of model-checkers on case studies

and $y = \text{TO}$ indicates that the model-checker timed out for skipping refinement. Our results show that model-checkers time out for most of the configurations when using I/O equivalence, and have difficulty automatically analyzing these systems. In contrast, all model-checkers except TIP can solve all the configurations using skipping refinement as the notion of refinement. Furthermore, there is an improvement of several orders of magnitude in the running time when using skipping refinement. Thus, skipping refinement acts as a scalability multiplier and allows us to extend the complexity of systems that can be automatically verified using state-of-the-art model-checkers.

Interactive Theorem Proving: We use ACL2s [17], an interactive theorem prover, to both model and verify infinite state models of the above three systems. We specify the state of machine using the data-definition framework in ACL2s [18], and formalize their operational semantics in a standard manner (by describing the effect of each instruction on the state of the machine), and define an appropriate refinement map. Once the definitions were in place, we proved that the optimized concrete implementation refines the abstract high-level specification using well-founded skipping simulation (Definition 5). We use the domain specific knowledge – size of internal buffers for the stack machine and the memory controller, and number of scalar instructions that can be packed in a single SIMD instruction for the Superword compiler transformation – to determine an upper bound on the maximum number of steps that the optimized concrete system can skip with respect to the high-level abstract system. The proof of correctness for the BSTK and OptMEMC with buffer size up to 3 was done using symbolic execution and no additional lemmas[4]. For the superword compiler transformation case study, we used a deductive verification method to prove the skipping refinement. A detailed exposition of the systems, their modeling and the proof of skipping refinement is given in more detail in our ACL2 2015 workshop paper [29].

[4]The proofs do depend on the theorems provided by standard ACL2 books.

9 A refinement-based approach to testing

Formal verification has been widely adopted in the industry as it provides strong guarantees of functional correctness. However, in spite of great advancements, techniques and tools that scale to large complex systems remains a challenge. As a result of this limitation, formal verification co-exists with dynamic validation. In fact, a recent study by Foster [23] suggests that dynamic validation dominates the design validation process in the industry. Though techniques based on dynamic validation are incomplete (*i.e.*, it cannot prove absence of bugs), they are lightweight and scalable to large designs. Due to coexistence of the two methodologies in a design validation process, it is highly beneficial that both formal verification and dynamic validation use the same specification language. Property-based validation methodology [22, 12, 10, 20] meets this requirement. A property is expressed in a specification language of choice, and is checked for violation during simulation of the system. This enables design validation to begin with the first property and progressively ramp up as designers write more properties. Furthermore, several tools based on model-checking and interactive theorem-provers are available to formally analyze these properties. As a result, it has become a de facto in the industry. However, this methodology faces several major challenges. First, it is difficult to determine if the set of properties under consideration is complete *i.e.*, the set of properties completely specifies the correctness of the system. Furthermore, as the design complexity increases, determination of completeness becomes harder. Second, defining oracles for tests (if the test passed or failed) is expensive and error-prone. And third, changes to implementation leads to, possibly significant, changes to the property. This is because properties are defined in terms of low-level implementation details, which evolves significantly during the life time of the design. And changing the specification of a design due to changes in the implementation is highly undesirable in a robust validation methodology.

In this work, we propose a new approach to dynamic validation that significantly alleviates the first and the second limitation and is comparably more robust to low-level changes in design. Our approach is based on the theory of refinement. Unlike in the property-based methodology where a large set of properties specifies the functional correctness of the implementation, in our approach a high-level executable abstract system and a *single* refinement conjecture specifies the functional correctness. The specification and the refinement check is compiled with the implementation and the refinement conjecture is checked during dynamic validation. The local proof methods [30, 36] that are amenable for mechanically analyzing refinement, also serves as a basis for designing an efficient algorithm for testing via refinement. The refinement problem which is naturally expressed in terms of infinite traces (that behaviors of a concrete system are allowed by the abstract system), is reduced to a problem expressed in terms of states and their successors.

9.1 Experimental Evaluation

We evaluate the effectiveness of our methodology in detecting bugs and the overhead costs of refinement checking during simulation for two systems: a simple 3-stage pipeline processors [41] and a simple hypervisor [9]. The refinement checking procedure is based on WEB refinement [36], an appropriate notion to specify the correctness of both systems. The models and the refinement-checking procedure are defined in a subset of Common Lisp in ACL2s. We now briefly describe the models and the result of the experiments.

9.1.1 Pipeline processor

We model a simple 3-stage pipeline processor. The stages of the processor are (1) fetch stage: the machine fetches an instruction pointed by the program counter; (2) load stage: the machine loads the source registers from the register file or the data memory; and (3) execute stage: the machine executes the instruction and updates the destination register in the register file or the data memory with the result. The refinement check for the processor is based on WEB refinement.

Mutations We manually injected 25 mutations in different components of the processor. Our test suite has four simple programs: copy an array of memory from one location to another, perform multiplication by iterative addition, perform exponentiation by iterative multiplication, and a short sequence of random addition and subtraction instructions. Notice that our test programs are generic and are not crafted to find any particular error. In an industrial setting, we would expect to have a larger set of programs, which will only make it easier to find errors. The test suite detected 18 out of the total 25 injected errors. Out of the 7 undetected mutations, 3 are non-functional errors (result in performance loss like stalling the pipeline for additional cycles). Although 2 of the undetected mutations could be caught with a more well-rounded test suite, the remaining 2 errors are *difficult-to-detect* with testing and required advanced counter-example generation techniques [19].

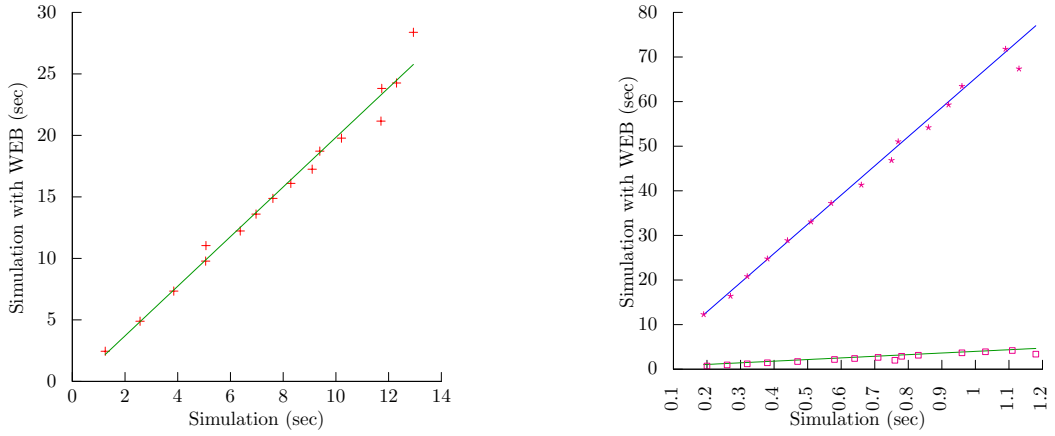
Overhead of refinement checker We analyze the overhead cost of the refinement check during simulation and plot the running times for simulating the processor in Figure 5a on x-axis *vs* the running time for the simulation *with* the refinement check on y-axis for number of simulation steps ranging from 10,000 to 100,000. The slowdown (slope of the fitting line) associated with checking the refinement conjecture during simulation is ~ 2 in for the processor

9.1.2 Simple Hypervisor

A hypervisor enables multiple operating systems (guests) to share resources without interfering with one another. It achieves this by virtualizing hardware resources (the host processor and the memory) of the system. As a result, any guest executing on a virtualized system only exhibits behaviors that are admissible when the guest is executing in isolation directly on the hardware. The refinement check for the hypervisor is based on WEB refinement.

Mutations We restrict our attentions to mutations in the hypervisor component of the virtualized system. We manually injected 14 mutations in the virtualized system and the test suite consists of short sequences of instructions and an appropriate setting of the guest page tables. The refinement checker found all of these errors.

Overhead of refinement checker We plot the running times for simulating the hypervisor in Figure 5b on x-axis *vs* the running time for the simulation *with* the refinement check on y-axis for number of simulation steps ranging from 10,000 to 100,000. The slowdown (slope of the blue fitting line) associated with checking the refinement conjecture during simulation is ~ 65 for the hypervisor. The reason for the large slow down is the refinement map: it extracts the guest memory from the host memory traverses the host memory and extracts the guest memory for each guest.



(a) 3-stage Pipeline Processor

(b) Hypervisor

Figure 5: Overhead cost of refinement checking

If the size of the memory is large, this is prohibitively expensive. To reduce the cost of computing the refinement map, we add a history variable to the virtualized system that records the guests accesses to the host memory. Note that augmenting the machine with this history variable does not modify its observable behavior. We then modify the refinement map to use the memory accesses recorded in the history variable to construct the updated guest memory from the initial guest memory. We again compare the running times for simulating the virtualized machine with and without the modified WEB refinement check. In this case the slope of the fitting line (green) is ~ 3.6, over 18 times speed up in the running time of the virtualized machine with the modified WEB refinement check. This experiment reaffirms that the refinement map plays a crucial role in efficiently analyzing refinement [40, 39].

Note that we do not claim that our approach completely eliminates the need for other testing methods, *e.g.*, low-level tests and properties that check performance are still needed. Also, we expect that directed tests will be needed to achieve sufficient code coverage. Nevertheless, we believe that the need for such tests will be significantly reduced if refinement-based testing is used.

10 Proposed Work and Schedule

10.1 Optimized reactive systems with unbounded skipping

In section §8, we demonstrated that WFSK can be used to effectively analyze optimized reactive systems with bounded skipping. However consider the optimized DES system (Example 2); at $t = 0$ let Sch be $\{(e_1, 0)\}$. Let execution of event e_1 add a new pair (e, k) to Sch, where k is an arbitrary large positive integer. At time $t = 0$, optimized DES system executes e_1. The priority queue then finds that the next event is scheduled to be executed at time $t = k$; hence it updates t to k. Next it executes event e. To show that the optimized DES refines its abstract high-level specification, using WFSK will require us to unroll the translation relation of the abstract DES k times. But, unlike the BSTK and the OptMEMC machines, we cannot place an upper bound on k for the optimized DES system; hence, unrolling is not a viable option. We propose to develop a proof method that extends the applicability of the notion of skipping refinement to effectively analyze reactive systems that may exhibit unbounded (but finite) skipping.

Notice that unrolling is not viable even in the case when there is an upper bound on skipping, but it is large. This limitation of WFSK is evident while proving the correctness of infinite state models of the BSTK and the OptMEMC machines in ACL2s using only symbolic execution (Section §8). The prover time for BSTK and OptMEMC with buffer capacity of 2 was ∼12 min and ∼2 min respectively, which increased to ∼2 hours and ∼3 hours respectively for the machines with buffer capacity of 3 [29]. We propose to evaluate the effectiveness of the proposed proof method by modeling variants of the BSTK and the OptMEMC machines with capacity of internal buffer greater than 3 and compare the scalability and running times of ACL2s for analyzing (using only symbolic reasoning) the correctness using WFSK and the new proof method.

10.2 Refinement-based testing

In section §9, we introduced a refinement-based testing methodology and showed that it can be effectively used to detect bugs in two simple systems. We now plan to appraise the effectiveness of the methodology to reason about a realistic implementation. For this, we propose to analyze the functional correctness of pipelined processors based on RISC-V instruction set architecture (ISA) [2]. RISC-V is an open source architecture and is being widely adopted both by the academia and industry alike. The community has developed an executable reference ISA model and a collection of processor models, called Sodor [3]. These models implement the RISC-V 32 bit integer base user-level instructions using different micro-architectural features. A model of the processor is described in Chisel, a hardware description language embedded in Scala, and automatically translated to C++ or Verilog; the later can then be compiled to an executable model [13]. We will analyze the functional correctness of the three available pipelined Sodor processors with respect to the reference ISA model using the refinement-based testing methodology. We will mutate the designs and evaluate the effectiveness of the methodology in detecting bugs (similar to Section §9.1).

The WEB refinement checker used to analyze the systems in section §9 is based on an online algorithm, *i.e.*, it checks for violations of the refinement conjecture as the concrete system and the abstract system are being simulated. However, the implementation of this algorithm requires that the checker has a mechanism to control the execution of the abstract and the concrete systems. It is often difficult to meet this requirement in practice. We propose to work on an alternative implementation that relaxes this requirement. In order to maintain a low overhead cost of the refinement checker during simulation, we intend to design the checker in a way that is amenable for parallelization. We will explore other techniques to reduce the overhead cost. Finally, we will prepare a technical report describing this work and submit it to a conference for publication.

11 Tentative Schedule

Proposal	April 2016
Skipping refinement (§10.1)	September 2016
Refinement-testing experiments (§10.2)	October-November 2016
Writing dissertation	December 2016-February 2017
Thesis Defense	March 2017

References