REFINEMENT-BASED REASONING OF OPTIMIZED REACTIVE SYSTEMS

A dissertation submitted by

Mitesh Jain

to the the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

Version: February 6, 2018

Abstract

We show that the correctness of a large class of optimized reactive systems can be effectively analyzed using refinement. Reasoning about reactive systems using refinement involves showing that any (infinite) behavior of a low-level, concrete *implementation* system is a behavior of the high-level abstract *specification* system. Existing notions of refinement do directly account for the differences in the unobservable behaviors (stuttering) of a concrete implementation and its abstract specification. However, they do not directly account for the differences in the observable behaviors of an optimized implementation and its abstract specification. Towards this we introduce two new notions of correctness, skipping simulation and reconciling simulation and develop a theory of refinement based on it. We study their algebraic properties and present several sound and complete proof-methods that can be used to effectively reason about them. The proof-methods reduce global reasoning about infinite computations of reactive systems to local reasoning about states and their successors and therefore are amenable to mechanical reasoning using existing verification tools.

Contents

Introduction **i**

1 Preliminaries **1**

1.1 Notations 1

1.2 Transition systems 2

1.3 Notions of correctness 3

1.4 Summary 7

2 Skipping Simulation **9**

2.1 Running Example 9

2.2 Skipping Simulation 11

2.2.1 Algebraic Properties 13

2.3 Skipping Refinement 17

2.4 Mechanised Reasoning 23

2.4.1 Reduced Well-Founded Skipping Simulation 23

2.4.2 Well-founded Skipping Simulation 29

2.4.3 Reduced Local Well-founded Skipping Simulation 31

2.4.4 Local well-founded Skipping Simulation 34

2.5 Summary 37

3 Reconciling Simulation **39**

3.1 Running Example (Continued) 39

3.2 Reconciling Simulation 40

3.3 Algebraic Properties 42

3.4 Reconciling refinement 43

3.5 Mechanised reasoning 45

3.5.1 Reduced Well-founded Reconciling Simulation 45

3.5.2 Well-founded Reconciling Simulation 49

3.5.3 Well-founded Reconciling Simulation with Explicit Stuttering 52

3.6 Summary 60

4 Case Studies **63**

4.1 Superword Level Parallelism with SIMD instructions 64

4.2 JVM-inspired Stack Machine 69

4.3 Memory Controller 76

4.4 Event Processing System 79

4.5 Conclusion 92

5 Related Work **95**

5.1 Notions of equivalences and refinement 95

5.2 Applications 101

5.2.1 Processor Verification 101

5.2.2 Software 101

Conclusions and Future Work **107**

List of Figures

2.1 Event Processing System 11

2.2 An example TS to show that SKS is not closed under intersection . . 14

2.3 Reduced well-founded skipping simulation (RWFSK) 23

2.4 Well-founded skipping simulation (WFSK) 29

2.5 Reduced local well-founded skipping simulation (RLWFSK) 33

2.6 Local well-founded skipping simulation (LWFSK) 35

3.1 An example to show that reconciling simulation is not compositional 43

3.2 Reduced well-founded reconciling simulation (RWRS) 46

3.3 Well-founded reconciling simulation (WRS) 49

3.4 Well-founded reconciling simulation with explicit stuttering (WRSS) 54

4.1 An example of superword parallelism optimization 65

4.2 Syntax and Semantics of Scalar and Vector Program 66

4.3 Syntax and Semantics of Stack and Buffered Stack Machine 71

4.4 Running time (log scale) of model-checkers for stack machine 76

4.5 Syntax and Semantics of MEMC and OptMEMC 78

4.6 Running time (log scale) of model-checkers for memory controller . . 80

Introduction

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. The main difference between a reactive system and a transformational system is the property of non-termination. Transformational systems are terminating; hence their correctness can be expressed using a relation between their inputs and outputs. In contrast, non-termination is an essential characteristic of reactive systems and their formal semantics are described using infinite objects such as infinite traces or computation trees. Hence, methods based on input-output relations are inadequate for reasoning about reactive systems. Moreover, the need to analyze the ongoing interaction of a reactive system with its environment enforces a view of its behaviors that is often less abstract than the input-output relational view of the behaviors of a transformational system.

One approach to analyze the correctness of a reactive system — the focus of this dissertation — is based on *refinement*. The concept of refinement for analyzing the correctness of reactive systems have been investigated since the early 1980s [32, 31, 34, 5]. In a refinement-based approach to reasoning, a high-level abstract system $\mathcal{A}$ serves as a specification for a concrete system $\mathcal{C}$ described at a lower level of abstraction. We say that $\mathcal{C}$ *refines* (implements) $\mathcal{A}$ *iff* all observable behaviors that

are allowed by $\mathcal{C}$ are the observable behaviors that are allowed by $\mathcal{A}$. Notice that the behaviors of $\mathcal{A}$ and $\mathcal{C}$ are described at different levels of abstraction. Hence a fundamental question in a refinement-based approach to reasoning is the following: what is an appropriate notion of refinement that relates observable behaviors of two systems described at different levels of abstraction? A concrete system describes its behaviors in more detail than the abstract system that serves as its specification. Hence it is often that $\mathcal{C}$ requires multiple steps to perform a task that is described in a single step in $\mathcal{A}$. Therefore, this phenomenon known as *stuttering*, must be directly accounted for in a notion of refinement [33].

The Problem The drive to build ever more efficient systems has led to highly-optimized implementations. These systems run "faster" than their simple abstract systems and in a single step perform the work of *multiple* abstract steps. For example, in order to reduce the memory latency and effectively utilize memory bandwidth, memory controllers in modern computer systems often buffer requests to the 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. As a further example, consider a modern superscalar microprocessor. To improve the instruction throughput multiple instructions are fetched in a single cycle. The fetched instructions are then analyzed for instruction-level parallelism and where possible (*e.g.*, in absence of data dependencies) are executed in parallel on the available compute units, eventually leading to multiple instructions being retired in a single cycle. Both the memory controller and the microprocessors exhibit stuttering behavior: a memory request is buffered for several steps before it eventually updates the memory location and a fetched instruction in a microprocessors travels through several pipeline stages before it eventually retires. But in addition to stuttering, a single step of these systems may perform work of multiple abstract

steps, *e.g.*, by updating multiple memory locations and retiring multiple instructions in a single cycle. The above examples are representative of a common occurrence: an optimized concrete system runs faster than its simple abstract system. Thus, existing notions of refinement that only account for stuttering are inadequate for analyzing the correctness of such optimized systems.

Next, consider the synchronization mechanisms used in concurrent systems. Traditionally, concurrent systems use locks to synchronize access to a resource that is shared among the concurrently running threads. Coarse-grained locks provide a simple mechanism to implement such synchronization: only one thread can access a shared resource at any time and operations "takes effect" in a single step. This limits the possible interleaving among concurrent threads and simplifies reasoning about the system. However, this also limits the available parallelism and scalability. In contrast, optimized implementations eschew locks and instead use a sequence of fine-grained synchronization primitives to update a shared resource. This mechanism reduces contention for a shared resource and increases the available parallelism. But it also allows more interleaving among concurrent threads and therefore makes reasoning difficult. What is an appropriate notion of refinement to show that such an optimized implementation refines the high-level abstract system with coarse-grained locks? Notice that because of the difference in possible interleavings among concurrent threads in the two systems, updates to the shared resources may happen in different order. Nevertheless, starting from related states, the two systems must eventually reach related states. If differences in updates to the shared resources are observable, the difference in behaviors of the optimized system and the abstract system cannot be classified as stuttering. Also, it is not the case that a step in the optimized system corresponds to multiple steps in the abstract system; they just happen in different order. Therefore, the differences in their behaviors cannot be

classified as skipping also. Thus, existing notions of refinement that only account for stuttering and skipping are inadequate to directly for analyzing the correctness of such optimized systems.

An appropriate and simple to understand notion of correctness is only part of the story. We also want to mechanically verify the correctness of optimized reactive systems. However, when reasoning about the correctness of reactive systems based on refinement, we have to analyze relationship between infinite computations of the concrete system and the abstract system. Support for such reasoning is rather limited in existing verification tools. Therefore, a notion of refinement must also admit an alternative proof-method that is amenable for mechanical reasoning.

Contirbutions and Structure of the Dissertation

In this dissertation, we develop a theory of refinement to effectively analyze the correctness of a large class of optimized reactive systems. We develop the theory using a generic model of a labeled transition system. Any system with a well-defined operational semantics can be mapped to a labeled transition system. Thus, our exposition is purely semantic and is agnostic with respect to the particular choice of language used to describe the systems. Moreover, we place no restrictions on the state space size or the branching factor of the transition system. Hence the theory is widely applicable.

In Chapter 2, we first introduce the notion of *skipping refinement*, a new notion that extends the domain of applicability of a refinement-based approach to the class of reactive systems, where a concrete implementation can execute "faster" than its abstract high-level specification. This is the first notion of refinement that we know of that can be used to directly analyze such optimized reactive systems. Skipping can be thought of as a dual of stuttering. Stuttering "stretches" a computation in

the sense that many steps of a concrete system corresponds to a single step of its specification. In contrast, skipping "squeezes" a computation in the sense that a single step in a concrete system corresponds to many steps of its specification. To show that skipping refinement is amenable for mechanical reasoning, we develop four sound and complete proof-methods to reason about skipping refinement. These proof-methods reduce reasoning about infinite computations to local reasoning about states and their successors. Moreover, the completeness results implies that if a system $\mathcal{M}_1$ is a refienemnt of $\mathcal{M}_2$ then we can always locally reason about it. We also study algebraic properties of skipping refinement. In particular, we show that it is *compositional*, *i.e.*, if system $\mathcal{M}_1$ is a skipping refinement of system $\mathcal{M}_2$ and system $\mathcal{M}_2$ is a skipping refinement of system $\mathcal{M}_3$, we can infer that that $\mathcal{M}_1$ is a skipping refinement of $\mathcal{M}_3$. Thus, skipping refinement aligns with a *top-down* design methodology: starting with a high-level abstract system that is as-simple-as-possible, design one feature at a time, eventually ending with an optimized concrete system. Just as the stepwise approach to design leads to a conceptually manageable design process for a complex system, it also leads to a manageable and scalable verification process.

In Chapter 3, we introduce the notion of *reconciling refinement*, that can be used to directly reason about the correctness of a concrete system that may differ from its high-level abstract specification in the intermediate states of the computations but eventually reconcile and reach related states. Reconciling refinement is a strictly weaker notion than skipping refinement in the sense that if $\mathcal{M}_1$ is a skipping refinement of $\mathcal{M}_2$, then $\mathcal{M}_1$ is also reconciling refinement of $\mathcal{M}_2$, but not the other way. To show that reconciling refinement is amenable for mechanical reasoning, we develop three sound and complete proof-methods to reason about reconciling refinement. These methods reduce reasoning about infinite computations to local

reasoning about states and their successor. We also study algebraic properties of reconciling refinement. In particular, we show that it is not compositional.

In Chapter 4, we present case studies that highlight the limitations of existing notions of refinement to reason about optimized reactive systems. We discuss considerations in choosing an appropriate notion of refinement and a proof-method to analyze it. We consider four case studies: (1) a vectorizing compiler transformation (2) a JVM-inspired stack machine, (3) an optimized memory controller, and (4) an asynchronous event-processing system. The first three case studies are examples of systems that exhibit bounded skipping while the fourth case study exhibits unbounded skipping. To facilitate understanding and focus on evaluating the theory of refinement, we only model certain aspects of the systems. We discuss why existing notions of refinement are not adequate for reasoning about such optimized systems. We also illustrate the use of suggested proof-methods and how the specific knowledge of the system under verification can be used to simplify the proof obligations. Furthermore, we provide experimental evidences showing that current model-checking and automated theorem proving tools have difficulty automatically analyzing these systems based on existing notion of correctness. In contrast, when skipping refinement is used, these verification tools can be used to analyze correctness of systems of much larger size.

In Chapter 5, we discuss the related work and finally conclude with remarks on future work.

Chapter 1

Preliminaries

This chapter contains notations used in this thesis. It also includes definition of a transition system and notions of correctness such as simulation and bisimultaion, stuttering simulation and bisimultaion, and associated proof-methods.

1.1 Notations

We first describe the notational conventions used in this dissertation. The set of natural number is denoted by ω and is the first infinite ordinal. The disjoint union operator is denoted by $\uplus$. Cardinality of a set is denoted by $\#S$. 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 relation R with itself i times (for $0 < i \leq \omega$) is denoted R^i. Given a relation R and $1 < k \leq \omega$, $R^{<k}$ denotes $\bigcup_{1 \leq i < k} R^i$ and $R^{\geq k}$ denotes $\bigcup_{\omega > i \geq k} R^i$. Instead of $R^{<\omega}$ we often write the more common R^+. $\uplus$ denotes the disjoint union operator. Quantified expressions are written as $\langle Qx \colon r \colon t \rangle$, where Q is the quantifier (*e.g.*, $\exists, \forall, min, \bigcup$), x is a bound variable, r is an expression that denotes the range of variable x (*true*, if omitted), and t is a term. The set of all elements x satisfying the predicate P is denoted either

as $\{x : P(x)\}$ or $\{x \mid P(x)\}$.

Let $R \subseteq S \times S$. The composition of binary relations P and Q is denoted $P;Q$ $= \{(x, y) : \langle \exists z : xPz \wedge zQy \rangle\}$. The *inverse* of R, written as R^{-1}, is defined as $\{(a, b) \mid bRa\}$. R is *reflexive iff* $\langle \forall s \in S :: sBs \rangle$. R is *symmetric iff* $\langle \forall s, w \in S :: sBw \Rightarrow wBs \rangle$. R is *antisymmetric iff* $\langle \forall s, w \in S :: sBw \wedge wBs \Rightarrow s = w \rangle$. R is transitive *iff* $\forall u, v, w \in S :: uBv \wedge vBw \Rightarrow uBw$. R is a *preorder iff* it is reflexive and transitive. A preorder that is also symmetric is an *equivalence* relation. A preorder that is also antisymmetric is a *partial order*.

A set with relations associated with it is called a *structure*. A *well-founded structure* $\langle W, \prec \rangle$[1], where W is a set and $\prec$ is a binary relation on W such that there are no infinitely decreasing sequences in W with respect to $\prec$.

1.2 Transition systems

A labeled transition system consists of a set of states that is a cartesian product of the domains of variables used to describe the system; a binary relation on states that models atomic transitions in the system; and a labeling function that tells what is observable in a state.

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 transition system is parameterized with a domain of observation and L tells us what is observable in a state. We allow the state space and the branching factor of the transition system to have arbitrary cardinalities. This generality is helpful in

[1]The syntax $\langle \rangle$ is overloaded and is used to define quantified expressions, structures (including transition systems), sequences, and segments in a transition system.

modeling a large class of reactive systems and the theory of refinement developed using it is broadly applicable. In particular, nondeterminism can be used to specify concurrent and distributed systems. Notice that our exposition is purely semantic and is agnostic to a particular programming language used to describe a system. Finally, the set of state and the transition relation can be represented either explicitly or use an assertion language with an ability to specify predicates and relations and support for validity and satisfiability checking.

The semantics of a program can be expressed as a transition system as follows. Let VAR be a (finite) set of program variables and $\mathbb{D}$ be the domain of interpretation of variables. A state is a mapping from VAR to D. Two states s and w are related by the transition relation $\rightarrow$, written $s \rightarrow w$, if it is possible to in one program *step* to go from s to w. The labeling function is an identity function. Notice that we place no restriction on the atomicity of a step and one must model the semantics of a program at a granularity that is appropriate for analysis.

Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system. An $\mathcal{M}$-path is a sequence of states such that for adjacent states, s and u, $s \rightarrow u$. The jth state in an $\mathcal{M}$-path is denoted by $\sigma.j$. An $\mathcal{M}$-path σ starting at state s is a *fullpath*, denoted by $fp.\sigma.s$, if it is infinite. An $\mathcal{M}$-segment, $\langle v_1, \ldots, v_k \rangle$, where $k \geq 1$ is a finite $\mathcal{M}$-path and is also denoted by $\vec{v}$. The length of an $\mathcal{M}$-segment $\vec{v}$ is denoted by $|\vec{v}|$. Let INC be the set of strictly increasing sequences of natural numbers starting at 0. The ith partition of a fullpath σ with respect to $\pi \in INC$, denoted by ${}^{\pi}\sigma^{i}$, is given by an $\mathcal{M}$-segment $\langle \sigma(\pi.i), \ldots, \sigma(\pi(i+1)-1) \rangle$.

1.3 Notions of correctness

In a refinement-based methodology, we say that a concrete low-level implementation system is *correct* with respect to an abstract high-level specific system if every

behavior allowed by the concrete system is a behavior of the abstract system. Several notions of correctness for analyzing reactive systems have been proposed in the past [53, 57]. Below we present the ones that have directly influenced the new notions proposed in this thesis.

Milner and Park [47, 51] first studied the notion of *simulation* and *bisimultaion* to compare behaviors of reactive systems.

Definition 2 (Simulation [47])**.** R is a simulation relation of a transition system $\mathcal{M} = \langle S, \rightarrow, L\rangle$ *iff* $R \subseteq S \times S$ and for all $s, w \in S$ such that sRw, both of the following conditions hold.

(SIM1) $L.s = L.w$

(SIM2) $\langle \forall u : s \rightarrow u : \langle \exists v : w \rightarrow v : uRv\rangle\rangle$

Definition 3 (Bisimulation [51])**.** B is a bisimulation relation of a transition system $\mathcal{M} = \langle S, \rightarrow, L\rangle$ *iff* $B \subseteq S \times S$ and for all $s, w \in S$ such that sRw, both of the following conditions hold.

(BISIM1) $L.s = L.w$

(BISIM2) $\langle \forall u : s \rightarrow u : \langle \exists v : w \rightarrow v : uBv\rangle\rangle$

(BISIM3) $\langle \forall v : w \rightarrow v : \langle \exists u : s \rightarrow u : uBv\rangle\rangle$

We say that s *simulates* w *iff* there is simulation relation R such that sRw holds. There exists a greatest simulation that is a preorder. Similarly, we say that s is bisimilar to w *iff* there is a bisimulation relation B such that sBw holds. There is a greatest bisimulation relation that is an equivalence relation. The definitions of simulation and bisimulation itself suggest a useful proof method for verification. It requires only local reasoning, *i.e.*, reasoning about states and their successors.

Lamport makes the case that a notion of correctness must directly account for *stuttering* [33]. Due to the difference in the level of abstraction used to specify the systems, it is often the case that the concrete system requires many steps to match one step of the abstract system. This is referred to as stuttering behavior. The notions of simulation and bisimulation are often too strong to directly analyze correctness of systems that exhibit such stuttering. Several variants of simulation and bisimulation have been studied in the literature [58]. We present here the notions of *stuttering simulation* and *stuttering bisimulation*. Stuttering simulation and bisimulation are defined using the notion of *match*, which we define below. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and B be a binary relation on S. Informally, a fullpath σ in $\mathcal{M}$ matches a fullpath δ in $\mathcal{M}$ if the fullpaths can be partitioned into non-empty, finite segments such that all states in a segment of σ are related by B to all states in the corresponding segment of δ.

Definition 4 (match [41])**.** Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system, σ, δ be fullpaths in $\mathcal{M}$. For $\pi, \xi \in \mathit{INC}$ and binary relation $B \subseteq S \times S$, we define

$$corr(B, \sigma, \pi, \delta, \xi) \equiv \langle \forall i \in \omega :: \langle \forall s \in {}^{\pi}\sigma^{i} \wedge w \in {}^{\xi}\delta^{i} :: sBw \rangle \rangle \textit{ and}$$

$$match(B, \sigma, \delta) \equiv \langle \exists \pi, \xi \in \mathit{INC} :: corr(B, \sigma, \pi, \delta, \xi) \rangle.$$

Definition 5 (Stuttering Simulation [41])**.** B is a stuttering simulation relation of a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ *iff* $B \subseteq S \times S$ and for all $s, w \in S$ such that sRw, both of the following conditions hold.

(STS1) $L.s = L.w$

(STS2) $\langle \forall \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B, \sigma, \delta) \rangle \rangle$

There is a greatest STS that is a preorder [41]. Though the notion of *matching fullpaths* is easy to understand as a correctness criterion, reasoning based on it would

require reasoning about nested quantifiers over fullpaths, hich often is difficult for automated verification tools. *Well-founded simulation* was proposed as an alternative characterization of STS [41] that requires only local reasoning, *i.e.*, reasoning about a state and its successors, and therefore amenable for automated verification.

Definition 6 (Well-founded Simulation (WFS) [41])**.** $B \subseteq S \times S$ is a well-founded simulation relation on transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

(WFS1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(WFS2) There exist functions, $rankt : S \times S \rightarrow W$, $rankl : S \times S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and

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

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

(b) $(uBw \wedge rankt(u, w) \prec rankt(s, w)) \vee$

(c) $\langle \exists v : w \rightarrow v : sBv \wedge rankl(v, s, u) < rankl(w, s, u) \rangle \rangle$

Theorem 1 ([41])**.** Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and $B \subseteq S \times S$. B is an STS on $\mathcal{M}$ *iff* B is WFS on $\mathcal{M}$.

Definition 7 (Stuttering Bisimulation [41])**.** B is a stuttering bisimulation on a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ *iff* both B and B^{-1} are STS's on $\mathcal{M}$.

There is a greatest STB that is an equivalence relation.

Definition 8 (Equivalence Stuttering Bisimulation [41])**.** B is an equivalence stuttering bisimulation (ESTB) on a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ *iff* B is an equivalence relation and an STS on $\mathcal{M}$.

In [41], it was shown that ESTB is completely characterized by well-founded equivalence bisimulation (WEB), an alternative characterization that requires only local reasoning.

Definition 9 (Well-founded Equivalence Bisimulation (WEB) [41])**.** $B \subseteq S \times S$ is a well-founded equivalence bisimulation relation on transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

(WEB1) B is an equivalence relation;

(WEB2) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$;

(WEB3) There exist functions, $\mathit{erankt} : S \rightarrow W$, $\mathit{erankl} : S \times S \rightarrow \omega$, such that $\langle W, \prec \rangle$ is well-founded and

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

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

(b) $(uBw \wedge \mathit{erankt}(u) \prec \mathit{erankt}(s)) \vee$

(c) $\langle \exists v : w \rightarrow v : sBv \wedge \mathit{erankl}(v, u) < \mathit{erankl}(w, u) \rangle \rangle$

Theorem 2 ([41])**.** Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and $B \subseteq S \times S$. B is an ESTB on $\mathcal{M}$ *iff* B is WEB on $\mathcal{M}$.

Notice that alternatively WEB could have been defined to be a WFS that is also an equivalence relation. However, this formalization is preferable since the witness ranking functions depend on fewer arguments than in WFS.

1.4 Summary

In this chapter, we presented notational conventions used in this dissertation. We gave an overview of simulation, bisimulation, stuttering simulation and stuttering bisimulation and associated proof-methods that are amenable for automated reasoning.

Chapter 2

Skipping Simulation

In this chapter, we first define the notion of skipping simulation. Then we study its algebraic properties and develop a compositional theory of skipping refinement. Finally, we develop sound and complete proof methods that are amenable for automated reasoning about skipping refinement.

2.1 Running Example

Consider an example of an event processing system system (EPS). An abstract high-level specification, AEPS, of an event processing system is defined as follows. Let E be a set of *events* and V be a set of *state variables*. A *state* of AEPS is a three-tuple $\langle t, Sch, St \rangle$, where t is a natural number denoting the current time; Sch is a set of pairs (e, t_e), where $e \in E$ is an event scheduled to be executed at time $t_e \geq t$; St is an assignment to state variables in V. The transition relation for the AEPS 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. Otherwise, we (nondeterministically) choose and execute an event of the form $(e, t) \in Sch$. The execution of an event may result in modifying St and also removing and adding a

finite number of new pairs (e', t') to Sch. We require that $t' > t$. Finally, execution involves removing the executed event (e, t) from Sch. This is a simple but a generic model of an event processing system. We place no restriction on the type of state variables or the type of events. Moreover, the ability to remove events can be used to specify systems with preemption [48]: an event scheduled to execute at some future time may be cancelled (and possibly rescheduled to be executed at a different time in future) as a result of execution of an event that preempts it.

Now consider, tEPS, an optimized implementation of AEPS. As before, a state is a three-tuple $\langle t, Sch, St\rangle$. However, unlike the abstract system which just increments time by 1 when there are no events scheduled at the current time, the optimized system finds the earliest time in future an event is scheduled to execute. The transition relation of tEPS 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 AEPS.

Consider an execution of AEPS and tEPS in Figure 2.1. (We only show the prefix of executions.) Suppose at $t = 0$, Sch be $\{(e_1, 0)\}$. The execution of event e_1 add a new pair (e_2, k) to Sch, where k is a positive integer. AEPS at $t = 0$, executes the event e_1, adds a new pair (e_2, k) to Sch, and updates t to 1. Since no events are scheduled to execute before $t = k$, the AEPS system repeatedly increments t by 1 until $t = k$. At $t = k$, it executes the event e_2. At time $t = 0$, tEPS executes e_1. The next event is scheduled to execute at time $t = k$; hence it updates in one step t to k. Next, in one step it executes the event e_2. Note that tEPS runs faster than AEPS by *skipping* over abstract states when no event is scheduled for execution at the current time. If $k > 1$, the step from s_2 to s_3 in tEPS neither corresponds to stuttering nor to a single step of the specification. Therefore, notions of refinement based on stuttering simulation and bisimulation are not directly applicable for reasoning about the correctness of tEPS. In this chapter, we define skipping refinement, a new no-

tion of refinement that directly supports reasoning about optimized reactive systems that can run faster than their high-level specification systems. We also develop a compositional theory of skipping refinement and provide sound and complete proof methods that require only local reasoning, thereby enabling mechanised verification of skipping refinement.

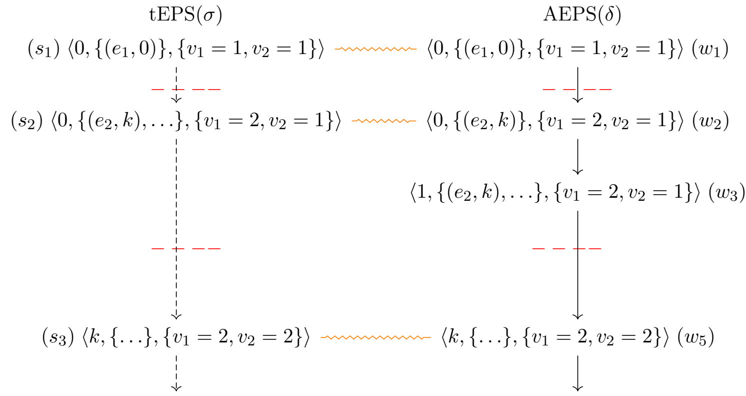

Figure 2.1: Event Processing System

2.2 Skipping Simulation

The definition of skipping simulation is based on the notion of *smatch*, a new notion of matching fullpaths. Informally, we say that a fullpath σ *smatches* a fullpath δ under the relation B if the fullpaths can be partitioned in to non-empty, finite segments such that all elements in a segment of σ are related to the first element in the corresponding segment of δ. Using the notion of *smatch*, skipping simulation is defined as follows: a relation B is a skipping simulation on a transition system $\mathcal{M} = \langle S, \rightarrow, L\rangle$, if for any $s, w \in S$ such that sBw, s and w are labeled identically and any fullpath starting at s can be smatched by some fullpath starting at w. Notice that we define skipping simulation using a single transition system, while our

ultimate goal is to define a notion of refinement that relates two transition system: an abstract transition system and a concrete transition system. We will see that this approach has some technical advantages. Moreover, it is easy to lift the notion defined on a single transition system to one that relates two transition systems by considering their disjoint union.

Definition 10 (smatch)**.** Let $\mathcal{M} = \langle S, \rightarrow, L\rangle$ be a transition system, σ, δ be fullpaths in $\mathcal{M}$. For $\pi, \xi \in \mathit{INC}$ and binary relation $B \subseteq S \times S$, we define

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

$$\mathit{smatch}(B, \sigma, \delta) \equiv \langle \exists \pi, \xi \in \mathit{INC} :: \mathit{scorr}(B, \sigma, \pi, \delta, \xi)\rangle.$$

In Figure 2.1, we illustrate the notion of smatching using our running example of an event processing system. In the figure, σ is a fullpath of tEPS and δ is a fullpath of AEPS. (We only show the prefix of the fullpaths.) The other parameter for matching is the relation B, which is just the identity function. In order to show that $\mathit{smatch}(B, \sigma, \delta)$ holds, we have to find $\pi, \xi \in \mathit{INC}$ 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 $\sim$. Since all elements of a σ partition are related to the first element of the corresponding δ partition, $\mathit{scorr}(B, \sigma, \pi, \delta, \xi)$ holds, therefore, $\mathit{smatch}(B, \sigma, \delta)$ holds.

Definition 11 (Skipping Simulation)**.** $B \subseteq S \times S$ is a skipping simulation on a transition system $\mathcal{M} = \langle S, \rightarrow, L\rangle$ *iff* for all s, w such that sBw, both of the following hold.

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

(SKS2) $\langle \forall \sigma : \mathit{fp}.\sigma.s : \langle \exists \delta : \mathit{fp}.\delta.w : \mathit{smatch}(B, \sigma, \delta)\rangle\rangle$

Theorem 3. Let $\mathcal{M}$ be a transition system. If B is an SKS on $\mathcal{M}$ then B is an STS on $\mathcal{M}$.

Proof: Follows directly from the definitions of SKS (Definition 11), *smatch* (Definition 10), STS(Definition 5), and *match*(Definition 4). □

2.2.1 Algebraic Properties

SKS enjoys several useful algebraic properties. In particular, it is closed under union and relational composition. The later property enables us to develop a theory of refinement that enables (vertical) modular reasoning of complex reactive systems.

Lemma 4. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and $\mathcal{C}$ be a set of SKS's on $\mathcal{M}$. Then $G = \langle \cup B : B \in \mathcal{C} : B \rangle$ is an SKS on $\mathcal{M}$.

Proof: Let $s, w \in S$ and sGw. We show that SKS1 and SKS2 hold for G. Since $G = \langle \cup B : B \in \mathcal{C} : B \rangle$, there is an SKS $B \in \mathcal{C}$ such that sBw. Since B is an SKS on $\mathcal{M}$, we have that $L.s = L.w$. Hence, SKS1 holds for G. Next SKS2 also holds for B, *i.e.*, for any fullpath σ starting at s, there is a fullpath δ starting at w such that $smatch(B, \sigma, \delta)$ holds. From the definition of *smatch*, there exists $\pi, \xi \in INC$ such that for all $i \in \omega$ and for all $s \in {}^{\pi}\sigma^{i}$, $sB\delta(\xi.i)$ holds. Since $B \subseteq G$, $sG\delta(\xi.i)$ holds. Hence $smatch(G, \sigma, \delta)$ holds, *i.e.*, SKS2 holds for G.

□

Corollary 5. For any transition system $\mathcal{M}$, there is a greatest SKS on $\mathcal{M}$.

Proof: Let $\mathcal{C}$ be the set of all SKS's on $\mathcal{M}$ and let $G = \langle \cup B : B \in \mathcal{C} : B \rangle$. By construction, G is the greatest and from Lemma 4, G is an SKS on $\mathcal{M}$.

□

The following lemma shows that SKS is not closed under intersection and negation.

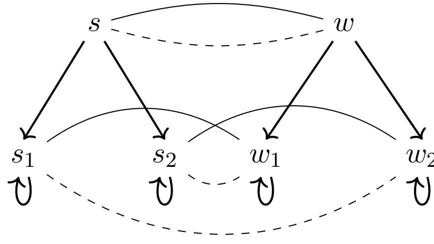

Figure 2.2: An example showing that SKS is not closed under intersection. Consider a TS with set of states $S = \{s, w, s_1, s_2, w_1, w_2\}$. The transition relation is denoted by solid arrows and all states are labeled identically. The first SKS relation B_1, denoted by solid lines, is $\{(s, w), (s_1, w_1), (s_2, w_2)\}$. The second SKS relation B_2, denoted by dashed lines is $\{(s, w), (s_1, w_2), (s_2, w_1)\}$. $B_1 \cap B_2$ is $\{(s, w)\}$ but does not include any related children of s and w.

Lemma 6. SKS are not closed under negation and intersection.

Proof: Consider a TS $\mathcal{M} = \langle S = \{a, b\}, \rightarrow = \{(a, a), (b, b)\}, L = \{(a, 1), (b, 2)\}\rangle$. The identity relation is an SKS, but its negation is not.

An example transition system to show that SKS is not closed under intersection appears in Figure 2.2.

The following lemma shows that skipping simulation is closed under relational composition.

Lemma 7. Let $\mathcal{M}$ be a transition system. If P and Q are SKS's on $\mathcal{M}$, then $R = P; Q$ is an SKS on $\mathcal{M}$.

Proof: To show that R is an SKS on $\mathcal{M} = \langle S, \rightarrow, L\rangle$, we show that for any $s, w \in S$ such that sRw, SKS1 and SKS2 hold. Let $s, w \in S$ and sRw. From the definition of R, there exists $x \in S$ such that sPx and xQw. Since P and Q are SKS's on $\mathcal{M}$, $L.s = L.x = L.w$, hence, SKS1 holds for R.

To prove that SKS2 holds for R, consider a fullpath σ starting at s. Since P and Q are SKSs on $\mathcal{M}$, there is a fullpath τ in $\mathcal{M}$ starting at x, a fullpath δ in $\mathcal{M}$ starting at w and $\alpha, \beta, \theta, \gamma \in INC$ such that $scorr(P, \sigma, \alpha, \tau, \beta)$ and $scorr(Q, \tau, \theta, \delta, \gamma)$ hold.

We use the fullpath δ as a witness and define $\pi, \xi \in INC$ such that $scorr(R, \sigma, \pi, \delta, \xi)$ holds.

We define a function, r, that given i, corresponding to the index of a partition of τ under β, returns the index of the partition of τ under θ in which the first element of τ's i^{th} partition under β resides.

$$r.i = j \text{ iff } \theta.j \leq \beta.i < \theta(j+1)$$

Note that r is indeed a function, as every element of τ resides in exactly one partition of θ. Also, since there is a correspondence between the partitions of α and β, (by $scorr(P, \sigma, \alpha, \tau, \beta)$), we can apply r to indices of partitions of σ under α to find where the first element of the corresponding β partition resides. Note that r is non-decreasing: $a < b \Rightarrow r.a \leq r.b$.

We define $\pi\alpha \in INC$, a strictly increasing sequence that will allow us to merge adjacent partitions in α as needed to define the strictly increasing sequence π on σ used to prove SKS2. Partitions in π will consist of one or more α partitions. Given i, corresponding to the index of a partition of σ under π, the function $\pi\alpha$ returns the index of the corresponding partition of σ under α.

$$\pi\alpha(0) = 0$$

$$\pi\alpha(i) = \text{ min } j \in \omega \text{ s.t. } |\{k : 0 < k \leq j \wedge r.k \neq r(k-1)\}| = i$$

Note that $\pi\alpha$ is an increasing function, *i.e.*, $a < b \Rightarrow \pi\alpha(a) < \pi\alpha(b)$. We now define π as follows.

$$\pi.i = \alpha(\pi\alpha.i)$$

There is an important relationship between r and $\pi\alpha$

$$r(\pi\alpha.i) = \cdots = r(\pi\alpha(i+1)-1)$$

That is, for all α partitions that are in the same π partition, the initial states of the corresponding β partitions are in the same θ partition.

We define ξ as follows.

$$\xi.i = \gamma(r(\pi\alpha.i))$$

We are now ready to prove SKS2. Let $s \in {}^{\pi}\sigma^{i}$. We show that $sR\delta(\xi.i)$. By the definition of π, we have

$$s \in {}^{\alpha}\sigma^{\pi\alpha.i} \vee \cdots \vee s \in {}^{\alpha}\sigma^{\pi\alpha(i+1)-1}$$

Hence,

$$sP\tau(\beta(\pi\alpha.i)) \vee \cdots \vee sP\tau(\beta(\pi\alpha(i+1)-1))$$

Note that by the definition of r (apply r to $\pi\alpha.i$):

$$\theta(r(\pi\alpha.i)) \leq \beta(\pi\alpha.i) < \theta(r(\pi\alpha.i)+1)$$

Hence,

$$\tau(\beta(\pi\alpha.i))Q\delta(\gamma(r(\pi\alpha.i))) \vee \cdots \vee \tau(\beta(\pi\alpha(i+1)-1))Q\delta(\gamma(r(\pi\alpha(i+1)-1)))$$

By the definition of ξ and the relationship between r and $\pi\alpha$ described above, we simplify the above formula as follows.

$$\tau(\beta(\pi\alpha.i))Q\delta(\xi.i) \vee \cdots \vee \tau(\beta(\pi\alpha(i+1)-1))Q\delta(\xi.i)$$

Therefore, by the definition of R, we have that $sR\delta(\xi.i)$ holds.

□

Theorem 8. The reflexive transitive closure of an SKS is an SKS.

Proof: Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a TS and B be an SKS on $\mathcal{M}$. The reflexive, transitive closure of B, written B^*, is $\langle \cup i \in \omega :: B^i \rangle$. First we show that for all $i \in \omega$, B^i is an SKS using induction on natural numbers. In the base case, B^0, the identity relation, is clearly an SKS. For $i \geq 0$, we have that $B^{i+1} = B; B^i$; from Lemma 7 and the induction hypothesis, we have that B^{i+1} is an SKS on $\mathcal{M}$. Finally, from Lemma 4, we have that $\langle \cup i \in \omega :: B^i \rangle$, *i.e.*, B^* is an SKS on $\mathcal{M}$.

□

Theorem 9. Given a TS $\mathcal{M}$, the greatest SKS on $\mathcal{M}$ is a preorder.

Proof: Let G be the greatest SKS on $\mathcal{M}$. From Theorem 8, G^* is an SKS. Hence $G^* \subseteq G$. Furthermore, since $G \subseteq G^*$, we have that $G = G^*$, *i.e.*, G is reflexive and transitive.

□

2.3 Skipping Refinement

In this section, we use the notion of skipping simulation, which is defined in terms of a *single* transition system to define the notion of skipping refinement, a notion that relates *two* transition systems: an *abstract* transition system and a *concrete* transition system. Informally, if a concrete system is a skipping refinement of an abstract system, then its observable behaviors are also behaviors of the abstract system, modulo skipping (which includes stuttering). The notion is parameterized by a *refinement map*, a function that maps concrete states to their corresponding abstract states. A refinement map along with a labeling function determines what

is observable at a concrete state. Using the algebraic properties of skipping simulation proved in the previous section, we show that skipping refinement can be used for the modular reasoning of complex reactive systems using a stepwise refinement methodology.

Note that we do not place any restriction on the state space size or the branching factor of the transition relations of the abstract and the concrete systems, and both can be of arbitrary infinite cardinalities. Thus, the theory of skipping refinement and sound and complete proof methods for reasoning about skipping refinement (developed in the Section 2.4) provide a reasoning framework that can be used to analyze a large class of reactive systems.

Definition 12 (Skipping Refinement)**.** Let $\mathcal{M}_{\text{A}} = \langle S_A, \xrightarrow{A}, L_A \rangle$ and $\mathcal{M}_{\text{C}} = \langle S_C, \xrightarrow{C}, L_C \rangle$ be transition systems and let $r : S_C \rightarrow\ S_A$ be a refinement map. We say $\mathcal{M}_C$ is a *skipping refinement* of $\mathcal{M}_A$ with respect to r, written $\mathcal{M}_C \lesssim_r \mathcal{M}_A$, if there exists a binary relation B 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 $s \in S_C$.

Notice that the above definition does not place any restriction on the choice of refinement map and depending on the systems under analysis one has great flexibility in its choice. In particular, the refinement map is not restricted to a simple *projection function* [4] that projects the observable component of a concrete state. In conjunction with the sound and complete proof methods presented in the next section, this provides a theory of refinement and a reasoning framework that is applicable to a large class of optimized reactive systems. The flexibility in the choice of refinement map is also useful in developing computationally efficient methods for

verification and testing [43, 26]. However, one should be prudent in the choice of refinement map; a complicated refinement map may bypass the verification problem. We discuss this aspect in more detail in Chapter 5.

Next, we use the property that skipping simulation is closed under relational composition to show that skipping refinement supports modular reasoning using a stepwise refinement approach. In order to verify that a low-level complex implementation $\mathcal{M}_C$ refines a simple high-level abstract specification $\mathcal{M}_A$ one proceeds as follows: starting with $\mathcal{M}_A$ define a sequence of intermediate lower level systems leading to the final complex implementation $\mathcal{M}_C$. At each step in the sequence, show that system at the current step is a refinement of the previous one. This approach is often more scalable than a monolithic approach. This is primarily because at each step, the verification effort is largely focused only on the difference between two systems under consideration. Note that this methodology is orthogonal to (horizontal) modular reasoning that infers correctness of a system from the correctness of its sub-components.

The following lemma is useful for lifting the notion of skipping simulation, which is defined on single transition system, to the notion of skipping refinement, which relates two transition system.

Lemma 10. Let S, S_1, S_2 be a set of states such that $S_1 \cap S_2 = \varnothing$ and $S_1 \cup S_2 \subseteq S$. Let B be an SKS on $\mathcal{M} = \langle S, \rightarrow, L \rangle$ such that any state in S_1 can only reach states in S_1, and any state in S_2 can only reach states in S_2, then $B' = \{(s_1, s_2) \mid s_1 \in S_1 \wedge s_2 \in S_2 \wedge s_1 B s_2\}$ is an SKS on $\mathcal{M}$.

Proof: Let $s_1 B' s_2$. We show that SKS1 and SKS2 holds for B'. From definition of B', we have that $s_1 \in S_1$, $s_2 \in S_2$, and $s_1 B s_2$. Since B is an SKS on $\mathcal{M}$, we have that $L.s_1 = L.s_2$; hence SKS1 holds for B'. Next let σ and δ be fullpaths in $\mathcal{M}$ starting at s_1 and s_2 respectively and $\pi, \xi \in INC$ such that $\langle \forall i \in \omega :: \langle \forall s \in {}^{\pi}\sigma^{i} ::$

$sB\delta(\xi.i)\rangle\rangle$ holds. Next from the assumptions that any state in S_1 can only reach states in S_1, and σ is a fullpath in $\mathcal{M}$ starting at $s_1 \in S_1$, all states in ${}^{\pi}\sigma^i$ are in S_1. Also, since any state in S_2 can only reach states in S_2, state $\delta(\xi.i) \in S_2$. Hence we have that $\langle \forall i \in \omega :: \langle \forall s \in {}^{\pi}\sigma^i :: sB'\delta(\xi.i)\rangle\rangle$, *i.e.*, SKS2 holds for B'.

$\square$

Theorem 11. Let $\mathcal{M}_1 = \langle S_1, \xrightarrow{1}, L_1\rangle$, $\mathcal{M}_2 = \langle S_2, \xrightarrow{2}, L_2\rangle$, and $\mathcal{M}_3 = \langle S_3, \xrightarrow{3}, L_3\rangle$ be transition systems, $p : S_1 \to S_2$ and $r : S_2 \to S_3$. If $\mathcal{M}_1 \lesssim_p \mathcal{M}_2$ and $\mathcal{M}_2 \lesssim_r \mathcal{M}_3$, then $\mathcal{M}_1 \lesssim_{p;r} \mathcal{M}_3$.

Proof: Since $\mathcal{M}_1 \lesssim_p \mathcal{M}_2$, we have an SKS, say A, such that $\langle \forall s \in S_1 :: sA(p.s)\rangle$. Furthermore, from Lemma 10, without loss of generality we can assume that $A \subseteq S_1 \times S_2$. Similarly, since $\mathcal{M}_2 \lesssim_r \mathcal{M}_3$, we have an SKS, say B, such that $\langle \forall s \in S_2 :: sB(r.s)\rangle$ and $B \subseteq S_2 \times S_3$. Define $C = A;B$. Then we have that $C \subseteq S_1 \times S_3$ and $\langle \forall s \in S_1 :: sCr(p.s)\rangle$. Also, from Theorem 8, C is an SKS on $\langle S_1 \uplus S_3, \xrightarrow{1} \uplus \xrightarrow{3}, \mathcal{L}\rangle$, where $\mathcal{L}.s = L_3(s)$ if $s \in S_3$ else $\mathcal{L}.s = L_3(r(p.s))$.

$\square$

Formally, to establish that a complex low-level implementation $\mathcal{M}_C$ refines a simple high-level abstract specification $\mathcal{M}_A$, one defines intermediate systems $\mathcal{M}_1$, $\ldots \mathcal{M}_n$, where $n \geq 1$ and establishes the following: $\mathcal{M}_C = \mathcal{M}_0 \lesssim_{r_0} \mathcal{M}_1 \lesssim_{r_1} \ldots \lesssim_{r_{n-1}} \mathcal{M}_n = \mathcal{M}_A$. Then from Theorem 11, we have that $\mathcal{M}_C \lesssim_r \mathcal{M}_A$, where $r = r_0; r_1; \ldots; r_{n-1}$. We illustrate the utility of this approach in Chapter 4 by proving correctness of two optimized event processing systems.

Theorem 12. Let $\mathcal{M} = \langle S, \to, L\rangle$ be a transition system. Let $\mathcal{M}' = \langle S', \to', L\rangle$ where $S' \subseteq S$, $\to' \subseteq S' \times S'$, $\to'$ is a left-total subset of $\to^+$, and $L' = L|_{S'}$. Then $\mathcal{M}' \lesssim_I \mathcal{M}$, where I is the identity function on S'.

Proof: Let B be I. Let σ be a fullpath starting at an $\mathcal{M}'$ state. To show that B is an SKS relation, the key observation is that since $\to' \subseteq \to^+$, there is a fullpath

starting from the correspodning $\mathcal{M}$ state, say δ, such that a step in σ corresponds to a finite, positive number of steps in δ. We choose such a fullpath δ as a witness and show σ and δ smatch under B. We consider the partitioning of σ such that a partition has only one state. Next we define the partitioning of δ. The ith partition of δ includes (1) the state, say s, in $\mathcal{M}$ corresponding to the state in the ith partition of σ, and (2) intermediate states in $\mathcal{M}$ required to reach from s to the state in $\mathcal{M}$ corresponding to the state in the $(i+1)^{th}$ partition of σ. It is easy to see that σ, δ and their partitions defined above satisfy *scorr* in Definition 10.

$\square$

Corollary 13. Let $\mathcal{M}_{\mathrm{C}} = \langle S_C, \xrightarrow{C}, L_C\rangle$ and $\mathcal{M}_{\mathrm{A}} = \langle S_A, \xrightarrow{A}, L_A\rangle$ be transition systems, $r : S_C \rightarrow S_A$ be a refinement map. Let $\mathcal{M}'_C = \langle S'_C, \xrightarrow{C}', L'_C\rangle$ where $S'_C \subseteq S_C$, $\xrightarrow{C}'$ is a left-total subset of $\xrightarrow{C}^+$, and $L'_C = L_C|_{S'_C}$. If $\mathcal{M}_C \lesssim_r \mathcal{M}_A$ then $\mathcal{M}'_C \lesssim_{r'} \mathcal{M}_A$, where r' is $r|_{S'_C}$.

Proof: From Lemma 12, $\mathcal{M}'_C \lesssim_I \mathcal{M}_C$, where I is the identity function on $S_{C'}$. Since $\mathcal{M}_C \lesssim_r \mathcal{M}_A$, from Theorem 11, we have that $\mathcal{M}'_C \lesssim_{I;r} \mathcal{M}_A = \mathcal{M}'_C \lesssim_{r'} \mathcal{M}_A$.

$\square$

We now illustrate the usefulness of the theory of skipping refinement using our running example of event processing systems. Consider MPEPS, an optimized EPS that uses a priority queue to find a non-empty set of events to execute next. As in Section 2.1, a state of MPEPS is a three tuple $\langle t, Sch, St\rangle$. The transition relation of MPEPS is defined as follows. Let t be the current time, and E_t be the set of events in *Sch* that are scheduled to execute at time t. If E_t is empty, then MPEPS uses the priority queue to find the minimum time $t' > t$ at which an event is scheduled for execution and updates the time to t'. Otherwise, MPEPS uses the priority queue to choose a non-empty subset of events in E_t and executes them. Note that we allow the priority queue in MPEPS to be deterministic or nondeterministic. For example,

the priority queue may deterministically select a single event in E_t to execute, or based on considerations such as resource utilization it may execute some subset of events in E_t in a single step. When reasoning about the correctness of MPEPS, one thing to notice is that there is a difference in the data structures used in the two systems: MPEPS uses a priority queue to effectively find the next set of events to execute in the scheduler, while AEPS uses a simple abstract set representation for the scheduler. Another thing to notice is that MPEPS can run "faster" than AEPS in two ways: it can increment time by more than 1 and it can execute more than one event in a single step. The theory of skipping refinement developed in this chapter enables us to separate out these concerns and apply a stepwise refinement approach to effectively analyze MPEPS.

First, we account for the difference in the data structures between MPEPS and AEPS. Towards, this we define an intermediate system MEPS that is identical to MPEPS except that the scheduler in MEPS is now represented as a set of event-time pairs. Under a refinement map, say p, that extracts the set of event-time pairs in the priority queue of MPEPS, a step in MPEPS can be matched by a step in MEPS. Hence, MPEPS $\lesssim_p$ MEPS. Next we account for the difference between MEPS and AEPS in the number of events the two systems may execute in a single step. Towards this, observe that the state space of MEPS and tEPS are equal and the transition relation of MEPS is a left-total subset of the transitive closure of the transition relation of tEPS. Hence, from Theorem 12, we infer that MPEPS is a skipping refinement of tEPS using the identity function, say I_1, as the refinement map, *i.e.*, MEPS $\lesssim_{I_1}$ tEPS. Next observe that the state space of tEPS and AEPS are equal and the transition relation of tEPS is left-total subset of the transition relation of AEPS. Hence, from Theorem 9, we infer that tEPS is a skipping refinement of AEPS using the identity function, say I_2, as the refinement map, *i.e.*, tEPS $\lesssim_{I_2}$ AEPS.

Finally, from the transitivity of skipping refinement (Theorem 11), we conclude that $MPEPS \lesssim_{p'} AEPS$, where $p' = p; I_1; I_2$.

2.4 Mechanised Reasoning

To prove that a transition system $\mathcal{M}_C$ is a skipping refinement of a transition system $\mathcal{M}_A$ using Definition 11, requires us to show that for any fullpath in $\mathcal{M}_C$, we can find a matching fullpath in $\mathcal{M}_A$. However, reasoning about nested quantifiers over infinite sequences is often problematic using automated tools. To redress the situation, we propose four alternative characterizations of skipping simulation that are amenable for mechanical reasoning.

2.4.1 Reduced Well-Founded Skipping Simulation

We first introduce reduced well-founded skipping simulation (RWFSK). The intuition is that for any pair of states s, w that are related and a state u such that $s \rightarrow u$, there are two cases to consider (Figure 2.4.1): (a) either the step from s to u is a stuttering step and u is related to w or (b) we can match the step from s to u with one or more steps from w.

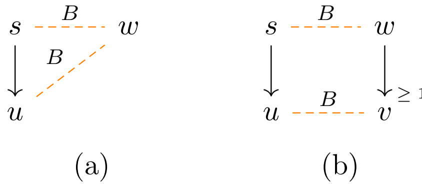

Figure 2.3: Reduced well-founded skipping simulation (a solid line with arrow indicates the transition relation, a dashed orange line indicates that states are related by B)

Definition 13 (Reduced Well-founded Skipping [25])**.** $B \subseteq S \times S$ is a reduced well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

(RWFSK1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(RWFSK2) There exists a function, $\mathit{rankt} : S \times S \to W$, such that $\langle W, \prec \rangle$ is well-founded and

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

(a) $(uBw \wedge \mathit{rankt}(u, w) \prec \mathit{rankt}(s, w)) \vee$

(b) $\langle \exists v : w \to^{+} v : uBv \rangle \rangle$

Observe that RWFSK2b accounts for both stuttering and skipping on the right. This is possible because skipping subsumes stuttering. Moreover, notice that the condition RWFSK2a is local, *i.e.*, it requires reasoning only about state and its successor. In contrast, the condition RWFSK2b is not local and in general may require unbounded reachability analysis, which can often be problematic for automated verification tools. Nevertheless, RWFSK is a useful proof method in case both stuttering and skipping on the right are bounded by a constant.

RWFSK characterizes skipping simulation, *i.e.*, it is a sound and complete proof method for reasoning about skipping simulation. We first prove the soundness, *i.e.*, RWFSK implies SKS.

Lemma 14 ([25]). Let $\mathcal{M}$ be a transition system. If B is an RWFSK on $\mathcal{M}$, then B is an SKS on $\mathcal{M}$.

Proof: To show that B is an SKS on $\mathcal{M} = \langle S, \to, L \rangle$, we show that for any $x, y \in S$ such that xBy, SKS1 and SKS2 hold. SKS1 follows directly from condition 1 of RWFSK.

Next we show that SKS2 holds. We start by recursively defining δ. In the process, we also define partitions π and ξ. For the base case, we let $\pi.0 = 0$, $\xi.0 = 0$ and $\delta.0 = y$. By assumption $\sigma(\pi.0)B\delta(\xi.0)$. For the recursive case, assume that we have

defined $\pi.0, \ldots, \pi.i$ as well as $\xi.0, \ldots, \xi.i$ and $\delta.0, \ldots, \delta(\xi.i)$. We also assume that $\sigma(\pi.i)B\delta(\xi.i)$. Let s be $\sigma(\pi.i)$; let u be $\sigma(\pi.i+1)$; let w be $\delta(\xi.i)$. We consider two cases.

First, say that RWFSK2b holds. Then, there is a v such that $w \rightarrow^{+} v$ and uBv. Let $\vec{v} = [v_0 = w, \ldots, v_m = v]$ be a finite path from w to v where $m \geq 1$. We define $\pi(i+1) = \pi.i + 1, \xi(i+1) = \xi.i + m$, ${}^{\xi}\delta^{i} = [v_0, \ldots, v_{m-1}]$ and $\delta(\xi(i+1)) = v$.

If the first case does not hold, *i.e.*, RWFSK2b does not hold, and RWFSK2a does hold. We define J to be the subset of the positive integers such that for every $j \in J$, the following holds.

$$\neg\langle \exists v : w \rightarrow^{+} v : (\sigma(\pi.i+j)Bv)\rangle \ \wedge \qquad (2.1)$$
$$\sigma(\pi.i+j)Bw \ \wedge \ \mathit{rankt}(\sigma(\pi.i+j), w) \prec \mathit{rankt}(\sigma(\pi.i+j-1), w)$$

The first thing to observe is that $1 \in J$ because $\sigma(\pi.i+1) = u$, RWFSK2b does not hold (so the first conjunct is true) and RWFSK2a does (so the second conjunct is true). The next thing to observe is that there exists a positive integer $n > 1$ such that $n \notin J$. Suppose not, then for all $n \geq 1, n \in J$. Now, consider the (infinite) suffix of σ starting at $\pi.i$. For every adjacent pair of states in this suffix, say $\sigma(\pi.i+k)$ and $\sigma(\pi.i+k+1)$ where $k \geq 0$, we have that $\sigma(\pi.i+k)Bw$ and that only RWFSK2a applies (*i.e.*, RWFSK2b does not apply). This gives us a contradiction because *rankt* is well-founded. We can now define n to be $\mathit{min}(\{l : l \in \omega \wedge l > 0 \wedge l \notin J\})$. Notice that only RWFSK2a holds between $\sigma(\pi.i+n-1)), \sigma(\pi.i+n)$ and w, hence $\sigma(\pi.i+n)Bw$ and $\mathit{rankt}(\sigma(\pi.i+n), w) \prec \mathit{rankt}(\sigma(\pi.i+n-1), w)$. Since Formula 2.1 does not hold for n, there is a v such that $w \rightarrow^{+} v \wedge \sigma(\pi.i+n)Bv$. Let $\vec{v} = [v_0 = w, \ldots, v_m = v]$ be a finite path from w to v where $m \geq 1$. We are now ready to extend our recursive definition as follows: $\pi(i+1) = \pi.i + n$, $\xi(i+1) = \xi.i + m$, and ${}^{\xi}\delta^{i} = [v_0, \ldots, v_{m-1}]$.

Now that we defined δ we can show that SKS2 holds. We start by unwinding definitions. The first step is to show that $fp.\delta.y$ holds, which is true by construction. Next, we show that $smatch(B, \sigma, \delta)$ by unwinding the definition of *smatch*. That involves showing that there exist π and ξ such that $scorr(B, \sigma, \pi, \delta, \xi)$ holds. The π and ξ we used to define δ can be used here. Finally, we unwind the definition of *corr*, which gives us a universally quantified formula over the natural numbers. This is handled by induction on the segment index; the proof is based on the recursive definitions given above.

□

Next we prove the completeness of RWFSK, *i.e.*, SKS implies RWFSK.

Lemma 15 ([25]). Let $\mathcal{M}$ be a transition system. If B is an SKS on $\mathcal{M}$, then B is an RWFSK on $\mathcal{M}$.

Let B be an SKS on $\mathcal{M}$. To show that B is an RWFSK on $\mathcal{M}$, we exhibit as witness the existence of a well-founded domain and a ranking function *rankt* that satisfy the conditions in RWFSK. Towards this, we first introduce a few definitions and lemmas.

Definition 14. Given a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$, the *computation tree* rooted at a state $s \in S$, denoted $ctree(\mathcal{M}, s)$, is obtained by "unfolding" $\mathcal{M}$ from s. Nodes of $ctree(\mathcal{M}, s)$ are finite sequences over S and $ctree(\mathcal{M}, s)$ is the smallest tree satisfying the following.

1. The root is $\langle s \rangle$.

2. If $\langle s, \ldots, w \rangle$ is a node and $w \rightarrow v$, then $\langle s, \ldots, w, v \rangle$ is a node whose parent is $\langle s, \ldots, w \rangle$.

Our next definition is used to construct the ranking function appearing in the definition of RWFSK.

Definition 15 (ranktCt)**.** Given a transition system $\mathcal{M} = \langle S, \rightarrow, L\rangle$, $s, w \in S$ and an SKS B on $\mathcal{M}$, $rank tCt(\mathcal{M}, B, s, w)$ is the empty tree if $\neg(sBw)$, otherwise $ranktCt(\mathcal{M}, B, s, w)$ is the largest subtree of $ctree(\mathcal{M}, s)$ rooted at s such that for any non-root node $\langle s, \ldots, x\rangle$ of the tree $ranktCt(\mathcal{M}, B, s, w)$, we have that xBw and $\langle \forall v : w \rightarrow^{+} v : \neg(xBv)\rangle$.

A basic property of our construction is the finiteness of paths.

Lemma 16. Let B be an SKS on $\mathcal{M}$. Every path of $ranktCt(\mathcal{M}, B, s, w)$ is finite.

Proof: The proof is by contradiction, so we start by assuming that there exists an infinite path, σ, in $ranktCt(\mathcal{M}, B, s, w)$. The path has to start at s. Let $\sigma(1) = u$ and let σ' be the suffix of σ starting at u. Since all states in σ' appear in $ranktCt(\mathcal{M}, B, s, w)$, by construction for any state x in σ', xBw and $\neg\langle \exists v : w \rightarrow^{+} v : xBv\rangle$. Since B is an SKS and uBw, there is a fullpath δ starting at w and $\pi, \xi \in INC$ such that $scorr(B, \sigma', \pi, \delta, \xi))$ holds. In particular, $\sigma(\pi.1)B\delta(\xi.1)$, and we have our contradiction because $\sigma(\pi.1)$ is in $ranktCt(\mathcal{M}, B, s, w)$ and can't be related by B to states reachable from w.

□

A node in a computation tree is a finite sequence of states. This induces a natural partial order *"is an initial segment of"* on nodes. A computation tree rooted at $s \in S$ tree has a root node $\langle s\rangle$ and is the maximum node. In the case of finite-path trees we can also refer to *minimal* nodes. Furthermore, we can use ordinal numbers to classify finite-path trees. Given Lemma 16, we define a function, *size*, that given a non-empty finite-path tree, say T, maps a node in the tree to an ordinal number. The ordinal assigned to a node $x \in T$ is defined as follows: if x is a leaf node in T, then $size(T, x) = 0$, else $size(T, x) = (\cup_{c \in children(T,x)} size(T, c)) + 1$, where $children(T, x)$ returns a subset of nodes that are immediate successors of x in T. We let the size of a computation tree to be the size of its root.

Note that we are using the standard set-theoretic representation for ordinal numbers, where an ordinal number is defined to be the set of ordinal numbers below it (*e.g.*, $2 = \{0, 1\}$), which also explains the notation union of ordinal numbers.

We define the size of a $\mathit{ranktCt}(\mathcal{M}, B, s, w)$ to be $\mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, s, w), \langle s \rangle)$. We use $\preceq$ to compare ordinal numbers (and therefore cardinal numbers as well).

Lemma 17 ([41]). If $|S| \preceq \kappa$, where $\omega \preceq \kappa$ then for all $s, w \in S$, $\mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, s, w))$ is an ordinal of cardinality $\preceq \kappa$.

Lemma 17 shows that we can use the cardinal $\mathit{max}(|S|^+, \omega)$ as the domain of our well-founded function in RWFSK2: either ω if the state space is finite, or $|S|^+$, the cardinal successor of the size of the state space otherwise.

Lemma 18. If $sBw, s \rightarrow u, \langle s, u \rangle \in \mathit{ranktCt}(\mathcal{M}, B, s, w)$ then $\mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, u, w)) \prec \mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, s, w))$.

Proof: Since $\langle s, u \rangle \in \mathit{ranktCt}(\mathcal{M}, B, s, w)$, from Lemma 16 and the definition of *size*, it follows that $\mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, u, w)) \prec \mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, s, w))$.

□

We are now ready to prove that SKS implies RWFSK.

Proof: RWFSK1 follows directly from SKS1. To show that RWFSK2 holds, let W be $\mathit{max}(|S|^+, \omega)$ and let $\mathit{rankt}(a, b)$ be $\mathit{size}(\mathit{ranktCt}(\mathcal{M}, B, a, b))$. Given $s, u, w \in S$ such that $s \rightarrow u$ and sBw, we show that either RWFSK2(a) or RWFSK2(b) holds.

There are two cases. First, suppose that $\langle \exists v : w \rightarrow^+ v : uBv \rangle$ holds, then RWFSK2(b) holds. If not, then $\langle \forall v : w \rightarrow^+ v : \neg(uBv) \rangle$, but B is an SKS so let σ be a fullpath starting at s and $\sigma.1 = u$. Then there is a fullpath δ such that $\mathit{fp}.\delta.w$ and $\mathit{smatch}(B, \sigma, \delta)$. Hence, there exists $\pi, \xi \in \mathit{INC}$ such that $\mathit{scorr}(B, \sigma, \pi, \delta, \xi)$. By the definition of *corr*, we have that $uB\delta(\xi.i)$ for some i, but i cannot be greater than 0 because then uBx for some x reachable from w, violating the assumptions

of the case we are considering. So, $i = 0$, *i.e.*, uBw. By Lemma 18, $rankt(u, w) = size(ranktCt(\mathcal{M}, B, u, w)) \prec size(ranktCt(\mathcal{M}, B, s, w)) = rankt(s, w)$.

□

2.4.2 Well-founded Skipping Simulation

We next introduce the notion of well-founded skipping simulation (WFSK). The intuition is that for any pair of states s, w that are related and a state u such that $s \rightarrow u$, there are four cases to consider (Figure 2.4.2): (a) either we can match the move from s to u in a single step, *i.e.*, there is a v such that $w \rightarrow v$ and u is related to v, or (b) a move from s to u is a stuttering step and u is related to w, or (c) w can make a move to v that is a stuttering step and v is related to s, or (d) a move from s to u skips one or more steps starting from w.

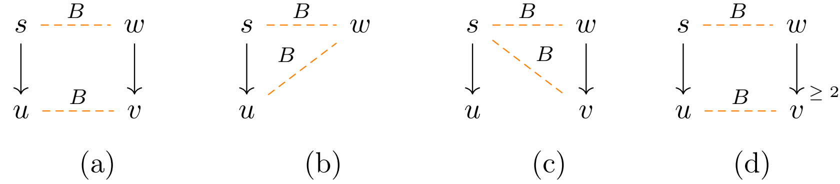

Figure 2.4: Well-founded skipping simulation (a solid line with arrow indicates the transition relation, a dashed orange line indicates that states are related by B)

Definition 16 (Well-founded Skipping (WFSK) [25]). $B \subseteq S \times S$ is a well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff :

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

(WFSK2) There exist functions, $rankt : S \times S \rightarrow W$, $rankl : S \times S \times S \rightarrow \omega$, such

that $\langle W, \prec \rangle$ is well-founded and

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

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

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

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

(d) $\langle \exists v : w \to^{\geq 2} v : uBv \rangle \rangle$

Observe that, reasoning about stuttering both on left (WFSK2b) and right (WFSK2c) is now local. Consider a scenario where we are verifying a system that has a bound—determined early early in the design—on the number of skipping steps possible but no such bound can be a priori determined for stuttering. If we use RWFSK, notice that RWSFK2b forces us to deal with stuttering and skipping steps in the same way, while WFSK enables us to distinguish the stuttering and skipping and locally deal with any amount of stuttering. However, the condition WFSK2d that accounts for skipping on the right still in general requires reachability analysis.

The following lemma asserts that WFSK and RWFSK are equivalent and therefore WFSK is also a sound and complete proof method to reason about skipping simulation.

Lemma 19 ([25]). Let $\mathcal{M}$ be a transition system. B is a WFSK on $\mathcal{M}$ *iff* B is an RWFSK on $\mathcal{M}$.

Proof: ($\Leftarrow$ direction): This direction is straightforward.
($\Rightarrow$ direction): Let $s, u, w \in S$, $s \to u$, and sBw. RWFSK1 follows directly from WFSK1.

Next we show that RWFSK2 holds. The key insight is that if we remove WFSK2c, then WFSK and RWFSK definitions are semantically equivalent. Therefore, it must be that WFSK2c is redundant. To see this, note that it allows finite stuttering on

the right (because *rankl* is well-founded), but finite stuttering is just a special case of more permissive skipping allowed by WFSK2a,d.

Let $s, u, w \in S$, $s \rightarrow u$, and sBw. If WFSK2a or WFSK2d holds then RWFSK2b holds. If WFSK2b holds, then RWFSK2a holds. So, what remains is to assume that WFSK2c holds and neither of WFSK2a, WFSK2b, or WFSK2d hold. From this we will derive a contradiction.

Let δ be a path starting at w, such that only WFSK2c holds between $s, u, \delta.i$. There are non-empty paths that satisfy this condition, *e.g.*, let $\delta = \langle w \rangle$. In addition, any such path must be finite. If not, then for any adjacent pair of states in δ, say $\delta.k$ and $\delta(k+1)$, $\mathit{rankl}(\delta(k+1), s, u) < \mathit{rankl}(\delta.k, s, u)$, which contradicts the well-foundedness of *rankl*. We also have that for every $k > 0$, $u \not\!B\ \delta.k$; otherwise WFSK2a or WFSK2d holds. Now, let δ be a maximal path satisfying the above condition, *i.e.*, every extension of δ violates the condition. Let x be the last state in δ. We know that sBx and only WFSK2c holds between s, u, x, so let y be a witness for WFSK2c, which means that sBy and one of WFSK2a,b, or d holds between s, u, y. WFSK2b can't hold because then we would have uBy (which would mean WFSK2a holds between s, u, x). So, one of WFSK2a,d has to hold, but that gives us a path from x to some state v such that uBv. The contradiction is that v is also reachable from w, so WFSK2a or WFSK2d held between s, u, w.

$\square$

2.4.3 Reduced Local Well-founded Skipping Simulation

Next we introduce the notion of reduced local well-founded skipping simulation (RLWFSK). Recall the event processing systems AEPS and tEPS described in Section(2.1). When no events are scheduled to execute at a give time, say t, tEPS increments time to the earliest time in future, say $k > t$, at which an event is sched-

uled to execute. Consider the scenario $k \geq t+1$. Since AEPS increments time by at most 1, in this scenario tEPS skips multiple states of AEPS. Moreover, execution of an event may add a new event to be executed at an arbitrary time in future. Therefore, one cannot a priori determine an upper-bound on k. Using WFSK to analyze correctness of such systems would require unbounded reachability analysis, a task often difficult for automated verification tools. In contrast, RLWFSK requires reasoning about states and their successors and can be used to effectively analyze systems that exhibit finite unbounded skipping.

Definition 17 (Reduced Local Well-founded Skipping (RLWFSK))**.** $B \subseteq S \times S$ is a local well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(RLWFSK1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(RLWFSK2) There exist functions, $\mathit{rankt} : S \times S \longrightarrow W$, $\mathit{rankls} : S \times S \longrightarrow \omega$ such that $\langle W, \prec \rangle$ is well founded, and, a binary relation $\mathcal{O} \subseteq S \times S$ such that

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

(a) $(uBw \wedge \mathit{rankt}(u, w) \prec \mathit{rankt}(s, w)) \vee$

(b) $\langle \exists v : w \rightarrow v : u\mathcal{O}v \rangle \rangle$

and

$\langle \forall x, y \in S : x\mathcal{O}y :$

(c) $xBy \vee$

(d) $\langle \exists z : y \rightarrow z : x\mathcal{O}z \wedge \mathit{rankls}(z, x) < \mathit{rankls}(y, x) \rangle \rangle$

As is the case with RWFSK and WFSK, RLWFSK also characterizes skipping simulation, *i.e.*, it is a sound and complete proof method for reasoning about skipping simulation. To prove the completeness, *i.e.*, SKS implies RLWFSK, we first prove that RWFSK implies RLWFSK. Then from Lemma 15, we infer that SKS implies RLWFSK. The soundness of RLWFSK follows from Theorem 22 proved later in the

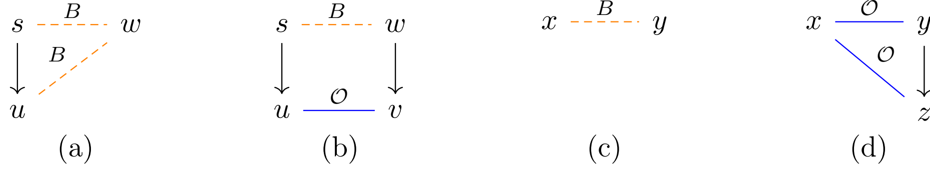

Figure 2.5: Reduced local well-founded skipping simulation (a solid line with arrow indicates the transition relation, a dashed orange line indicates that states are related by B and a solid blue line indicate the states are related by $\mathcal{O}$)

section.

Lemma 20. Let $\mathcal{M}$ be a transition system. If B is an RWFSK on $\mathcal{M}$, then B is an RLWFSK on $\mathcal{M}$.

Proof: Let B be an RWFSK on $\mathcal{M}$. RLWFSK1 follows directly from RWFSK1.

To show that RLWFSK2 holds, we use any *rankt* function that can be used to show that RWFSK2 holds. We define $\mathcal{O}$ as follows.

$$\mathcal{O} = \{(u, v) : \langle \exists z : v \rightarrow^{+} z : uBz \rangle\}$$

We define $rankls(u, v)$ to be the minimal length of a $\mathcal{M}$-segment that starts at v and ends at a state, say z, such that uBz, if such a segment exists and 0 otherwise.

Let $s, u, w \in S$, sBw and $s \rightarrow u$. If RWFSK2a holds between s, u, and w, then RLWFSK2a also holds. Next, suppose that RWFSK2a does not hold but RWFSK2b holds, *i.e.*, there is an $\mathcal{M}$-segment $\langle w, a, \ldots, v \rangle$ such that uBv; therefore, $u\mathcal{O}a$ and RLWFSK2b holds.

To finish the proof, we show that $\mathcal{O}$ and *rankls* satisfy the constraints imposed by the second conjunct in RLWFSK2. Let $x, y \in S$, $x\mathcal{O}y$ and $x \not B\ y$. From the definition of $\mathcal{O}$, we have that there is an $\mathcal{M}$-segment from y to a state related to x by B; let $\vec{y}$ be such a segment of minimal length. From the definition of *rankls*, we have $rankls(y, x) = |\vec{y}|$. Observe that y cannot be the last state of $\vec{y}$ and

$|\vec{y}| \geq 2$. This is because the last state in $\vec{y}$ must be related to x by B, but from the assumption we know that $x \not\!B\, y$. Let y' be a successor of y in $\vec{y}$. Clearly, $x\mathcal{O}y'$; therefore, $\mathit{rankls}(y', x) < |\vec{y}| - 1$, since the length of a minimal $\mathcal{M}$-segment from y' to a state related to x by B, must be less or equal to $|\vec{y}| - 1$.

□

Lemma 21. Let $\mathcal{M}$ be a transition system. If B is an SKS on $\mathcal{M}$, then B is an RLWFSK on $\mathcal{M}$.

Proof: Follows directly from Lemma 20 and Lemma 15.

□

2.4.4 Local well-founded Skipping Simulation

Reduced local well-founded skipping simulation introduced above requires only local reasoning and therefore is amenable for mechanical reasoning. However, note that RLWFSK (like RWFSK), does not differentiate between skipping and stuttering on the right. Such a differentiation can often be useful in practice. To redress this, we define local well-founded skipping simulation (LWFSK), a characterization of skipping simulation that separates reasoning about skipping from reasoning about stuttering on the right.

Definition 18 (Local Well-founded Skipping (LWFSK))**.** $B \subseteq S \times S$ is a local well-founded skipping relation on TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(LWFSK1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

(LWFSK2) There exist functions, $\mathit{rankt} : S \times S \longrightarrow W$, $\mathit{rankl} : S \times S \times S \longrightarrow \omega$, and $\mathit{rankls} : S \times S \longrightarrow \omega$ such that $\langle W, \prec \rangle$ is well founded, and, a binary relation

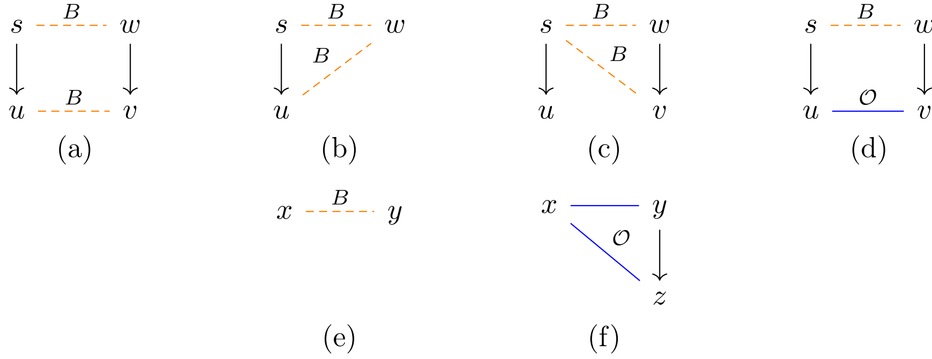

Figure 2.6: Local well-founded skipping simulation (a solid line with arrow indicates the transition relation, a dashed orange line indicates that states are related by B and a solid blue line indicate the states are related by $\mathcal{O}$)

$\mathcal{O} \subseteq S \times S$ such that

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

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

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

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

(d) $\langle \exists v : w \rightarrow v : u\mathcal{O}v \rangle \rangle$

and

$\langle \forall x, y \in S : x\mathcal{O}y :$

(e) $xBy \vee$

(f) $\langle \exists z : y \rightarrow z : x\mathcal{O}z \wedge \mathit{rankls}(z, x) < \mathit{rankls}(y, x) \rangle \rangle$

As was the case with RLWFSK, to prove that a relation is an LWFSK, reasoning about single steps of the transition system suffices. However, LWFSK2c accounts for stuttering on the right, and LWFSK2d along with LWFSK2e and LWFSK2f account for skipping on the right. Also observe that states related by $\mathcal{O}$ are not required to be labeled identically.

We conclude by showing that the four notions, RWFSK, WFSK, RLWFSK, and LWFSK, introduced in this section are equivalent and completely characterize skipping simulation.

Theorem 22. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and $B \subseteq S \times S$. The following statements are equivalent:

(i) B is an SKS on $\mathcal{M}$;

(ii) B is an RLWFSK on $\mathcal{M}$;

(iii) B is an LWFSK on $\mathcal{M}$;

(iv) B is a WFSK on $\mathcal{M}$;

(v) B is an RWFSK on $\mathcal{M}$;

Proof: That (ii) implies (iii) follows from the simple observation that RLWFSK2 implies LWFSK2. That (iv) implies (v) follows from Lemma 19, that (v) implies (i) follows from Lemma 14, and that (i) implies (ii) follows from Lemma 21. To complete the proof, we prove that (iii) implies (iv) in Lemma 23,

Lemma 23. Let $\mathcal{M}$ be a transition system. If B is an LWFSK on $\mathcal{M}$, then B is a WFSK on $\mathcal{M}$.

Proof: Let B be an LWFSK on $\mathcal{M}$. WFSK1 follows directly from LWFSK1.

Let *rankt*, *rankl*, and *rankls* be functions, and $\mathcal{O}$ be a binary relation such that LWFSK2 holds. To show that WFSK2 holds, we use the same *rankt* and *rankl* functions and let $s, u, w \in S$ and $s \rightarrow u$ and sBw. LWFSK2a, LWFSK2b and LWFSK2c are equivalent to WFSK2a, WFSK2b and WFSK2c, respectively, so we show that if only LWFSK2d holds, then WFSK2d holds. Since LWFSK2d holds, there is a successor v of w such that $u\mathcal{O}v$. Since $u\mathcal{O}v$ holds, either LWFSK2e or

LWFSK2f must hold between u and v. However, since LWFSK2a does not hold, LWFSK2e cannot hold and LWFSK2f must hold, *i.e.*, there exists a successor v' of v such that $u\mathcal{O}v' \wedge rankls(v', u) < rankls(v, u)$. So, we need a path of at least 2 steps from w to satisfy the universally quantified constraint on $\mathcal{O}$. Let us consider an arbitrary path, δ, such that $\delta.0 = w$, $\delta.1 = v$, $\delta.2 = v'$, $u\mathcal{O}\delta.i$, LWFSK2e does not hold between u and $\delta.i$ for $i \geq 1$, and $rankls(\delta.(i+1), u) < rankls(\delta.i, u)$. Notice that any such path must be finite because *rankls* is well founded. Hence, δ is a finite path and there exists a $k \geq 2$ such that LWFSK2e holds between u and $\delta.k$. Therefore, WFSK2d holds, *i.e.*, there is a state in δ reachable from w in two or more steps which is related to u by B.

□

Theorem 22 shows that under no restrictions on systems under consideration, RLWFSK (Definition 17) and LWFSK (Definition 18) are complete proof methods and require only local reasoning. Thus, to prove that a concrete system is a skipping refinement of an abstract system, one can always prove it using local reasoning about states and their successors and do not require global reasoning about infinite paths. We discuss this in more detail in Chapter 5.

2.5 Summary

In this chapter, we introduced a new notion of skipping simulation. We showed that skipping simulation enjoys several useful algebraic properties and developed a compositional theory of skipping refinement that aligns with a stepwise refinement verification methodology [61, 5]. Finally, we developed four alternative characterizations of skipping simulation that are amenable for mechanised reasoning using formal-methods tools.

In Chapter 4, we present four case studies that highlight the applicability of

the proof methods: a JVM-inspired stack machine, a simple memory controller, a scalar to vector compiler transformation, and an optimized event processing system. Our experimental results demonstrate that current model-checking and automated theorem proving tools have difficulty analyzing these systems using existing notions of correctness, but they can effectively analyze the systems using skipping refinement.

Chapter 3

Reconciling Simulation

In this chapter, we first define the notion of reconciling simulation. Then we study its algebraic properties and develop a theory of reconciling refinement. Finally, we develop sound and complete proof methods that are amenable for automated reasoning about reconciling refinement.

3.1 Running Example (Continued)

Consider event processing systems AEPS and MPEPS described in Chapter 2. If no event is scheduled to execute at the current time, say t, AEPS increments time to $t+1$ while MPEPS updates time to the earliest time in future at which an event is scheduled to execute. Otherwise, AEPS picks an event while MPEPS picks a non-empty subset of events from the set of events scheduled to execute at t. AEPS and MPEPS do not stutter. Earlier, we had argued that MPEPS is a skipping refinement of AEPS. But that there is a sense in which AEPS refines MPEPS. Suppose AEPS and MPEPS are in related states and E_t be the set of events scheduled for execution at current time t. At each step, AEPS nondeterministically chooses events to execute from E_t in some order. Then MPEPS can also choose to execute events in

the same order though it may execute more than one event in a single step. Hence, a single AEPS step neither corresponds to stuttering nor corresponds to skipping. Therefore, skipping refinement (and stuttering refinement) is not an appropriate notion of refinement to directly analyze the correctness of AEPS. But the two systems eventually *reconcile* after executing all events in E_t. In this chapter, we define reconciling refinement, a new notion of refinement that can be used to directly analyze correctness of such reactive systems. Reconciling refinement is strictly weaker than skipping refinement, and extends the domain of applicability of the refinement-based approach to verification of a larger class of reactive systems. We develop the theory of reconciling refinement using an approach that is analogous to one used to develop the theory of skipping refinement (Chapter 2). We first define the notion of reconciling simulation using a single transition system and study its algebraic properties. Then we use the notion of reconciling simulation and a refinement map to define the notion of reconciling refinement, a notion that relates *two* transition systems: an abstract transition system and a concrete transition system. Finally, we develop sound and complete proof methods for reasoning about reconciling refinement that require only local reasoning, and are amenable for mechanised verification.

3.2 Reconciling Simulation

The notion of reconciling simulation is based on *rmatch*, a new notion of matching fullpaths. Informally, we say a fullpath σ rmatches a fullpath δ under the relation B if the fullpaths can be partitioned in to non-empty, finite segments such that the first element in a segment of σ is related to the first element in the corresponding segment of δ. Using the notion of rmatch, reconciling simulation is defined as follows: a relation B is a reconciling simulation on a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$, if for any $s, w \in S$ such that sBw, s and w are labeled identically and any fullpath starting

at s can be rmatched by some fullpath starting at w.

Definition 19 (rmatch)**.** Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system, σ, δ be fullpaths in $\mathcal{M}$. For $\pi, \xi \in INC$ and binary relation $B \subseteq S \times S$, we define

$$rcorr(B, \sigma, \pi, \delta, \xi) \equiv \langle \forall i \in \omega :: \sigma(\pi.i) B \delta(\xi.i) \rangle \ \ and$$

$$rmatch(B, \sigma, \delta) \equiv \langle \exists \pi, \xi \in INC :: rcorr(B, \sigma, \pi, \delta, \xi) \rangle.$$

Definition 20 (Reconciling Simulation)**.** $B \subseteq S \times S$ is a reconciling simulation (RES) on a TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ *iff* for all s, w such that sBw, both of the following hold.

(RES1) $L.s = L.w$

(RES2) $\langle \forall \sigma \colon fp.\sigma.s \colon \langle \exists \delta \colon fp.\delta.w \colon rmatch(B, \sigma, \delta) \rangle \rangle$

Notice that the definition reconciling simulation differs from skipping simulation only in the notion of matching fullpaths used to define them. Both require that matching fullpaths σ and δ be partitioned into finite non-empty segments. But they differ in what states in a segment of σ relate to the first state in the corresponding segment of δ: while SKS requires that all states in a segment of σ are related to the first state in the corresponding segment of δ, RES only requires that first state in a segment of σ is related to the first state in the corresponding segment of δ. Clearly, RES is strictly weaker than SKS.

Theorem 24. If B is an SKS on $\mathcal{M}$, then B is an RES on $\mathcal{M}$.

Proof: Follows directly from the definitions of SKS(Definition 11), *smatch*(Definition 10), RES(Definition 20), and *rmatch*(Definition 19) .

□

3.3 Algebraic Properties

We next study the algebraic properties of reconciling simulation. RES is closed under arbitrary union. However, it is not closed under relational composition.

Lemma 25. Let $\mathcal{M}$ be a transition system and $\mathcal{C}$ be a set of RES's on $\mathcal{M}$, then $G = \langle \cup B \colon B \in \mathcal{C} \colon B \rangle$ is an RES on $\mathcal{M}$.

Proof: Let $s, w \in S$ and sGw. We show that RES1 and RES2 holds for G. Since $G = \langle \cup B : B \in \mathcal{C} : B \rangle$, there is an RES $B \in \mathcal{C}$ on $\mathcal{M}$ such that sBw. Since B is an RES on $\mathcal{M}$, we have that $L.s = L.w$. Hence, RES1 holds for G. Next RES2 also holds for B, *i.e.*, for any fullpath σ starting at s, there is a fullpath δ starting at w such that $rmatch(B, \sigma, \delta)$ holds. From Definition 19 of *rmatch*, there exists $\pi, \xi \in INC$ such that $\langle \forall i \in \omega :: \sigma(\pi.i) \; B \; \delta(\xi.i) \rangle$. Since $B \subseteq G$, we have that $\langle \forall i \in \omega :: \sigma(\pi.i) \; G \; \delta(\xi.i) \rangle$. Hence, from Definition 19, $rmatch(G, \sigma, \delta)$ holds, *i.e.*, RES2 holds for G.

□

Corollary 26. For any transition system $\mathcal{M}$, there is a greatest RES on $\mathcal{M}$.

Proof: Let $\mathcal{C}$ be the set of all RES's on $\mathcal{M}$ and $G = \langle \cup B : B \in \mathcal{C} : B \rangle$. By construction, G is the greatest and from Lemma 25, G is an RES on $\mathcal{M}$.

□

The following lemma shows that RES is not closed under intersection and negation, as was the case with stuttering simulation [40] and skipping simulation 2.2.

Lemma 27. RESs are not closed under negation or intersection.

Proof: The example TSs used in the proof in Lemma 6 provide counterexamples.

□

However, unlike stuttering simulations and skipping simulations, RESs are not closed under relational composition.

Lemma 28. RESs are not closed under relational composition.

Proof: A counterexample appears in Figure 3.1.

□

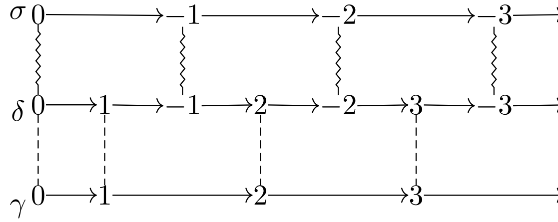

Figure 3.1: An example showing that RESs are not closed under relational composition. Consider a TS $\mathcal{M}$ that consists of three fullpaths σ, δ, and γ. The transition relation for the TS is represented by $\rightarrow$. The labeling function is defined as follows: $L(\sigma.0) = L(\delta.0) = L(\gamma.0) = 0$. And for all $i \geq 1$, $L(\sigma.i) = -i$, $L(\delta.i) = (i-1)/2+1$ if i is odd and $L(\delta.i) = -i/2$ if i is even, and $L(\gamma.i) = i$. The two reconciling simulations, B and B', are denoted by squiggly and dashed lines. We have that $(\sigma.0)B(\delta.0)$ and $(\delta.0)B'(\gamma.0)$. However, $(\sigma.0)B;B'(\gamma.0)$ does not hold because they have no rmatching successors. In fact there is no RES that relates $\sigma.0$ with $\gamma.0$.

3.4 Reconciling refinement

In this section, we use the notion of reconciling simulation to define the notion of reconciling refinement, a notion that relates *two* transition systems: an *abstract* transition system and a *concrete* transition system. The notion is parameterized by a refinement map, a function that maps a concrete state to an abstract state. Informally, if a concrete system, say $\mathcal{C}$, is a reconciling refinement of an abstract system,

say $\mathcal{A}$, under a refinement map r, then observable behaviors of $\mathcal{C}$ are observable behaviors of $\mathcal{A}$ up to reconciliation (reconciliation subsumes skipping and stuttering). Recall that a refinement map along with the labeling function determines what is observable at a concrete state.

Note that we do not place any restriction on the state space sizes and the branching factor of the transition relation of the abstract and the concrete systems, and both can be of arbitrary infinite cardinalities. Thus the theory of reconciling refinement, like the theory of skipping refinement developed in Chapter 2, and sound and complete proof methods for reasoning about reconciling refinement (developed in the Section 3.5) provide a reasoning framework that can be used to analyze a large class of reactive systems.

Definition 21 (Reconciling Refinement)**.** 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 $r\colon S_C \rightarrow\ S_A$ be a refinement map. We say $\mathcal{M}_C$ is a *reconciling refinement* of $\mathcal{M}_A$ with respect to r, written $\mathcal{M}_C \trianglelefteq_r \mathcal{M}_A$, if there exists a binary relation B such that all of the following hold.

1. $\langle \forall s \in S_C :: sB(r.s)\rangle$ *and*

2. B is an RES 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 $s \in S_C$.

Observe that our definition of reconciling refinement is quite general, *e.g.*, the state space and the branching factor of the transition relation of the systems can be of arbitrary infinite cardinality and there are no restrictions on the choice of refinement map. In the next section, we develop sound and complete proof methods that are amenable for mechanised reasoning. This provides a general theory of refinement and a reasoning framework that is applicable for analyzing a large class of optimized reactive systems. However, in general reconciling refinement is not compositional

and therefore, unlike skipping refinement, reconciling refinement does not align with the stepwise refinement verification methodology.

3.5 Mechanised reasoning

We turn our attention to the mechanical verification of correctness of systems using reconciling refinement. If we use Definition 21 to prove that transition system $\mathcal{M}_C$ is a reconciling refinement of transition system $\mathcal{M}_A$ with respect to a refinement map r, we must show that for any fullpath in $\mathcal{M}_C$ there is an rmatching fullpath in $\mathcal{M}_A$. However, reasoning about formulas with nested quantifiers over infinite sequences using formal-methods tools tends to be difficult, *e.g.*, SMT solvers and model checkers either do not allow or do not fully support quantifiers and the manual effort required to prove such theorems with interactive theorem provers can be quite high. To redress this situation, we introduce alternative characterizations of reconciling simulation that are amenable for mechanised reasoning.

3.5.1 Reduced Well-founded Reconciling Simulation

As a first step, we introduce the notion of *reduced well-founded reconciling simulation* (RWRS). It localizes reasoning about reconciliation on the left using an additional binary relation over states and a rank function. However, reasoning about reconciliation on the right still requires reachability analysis and in general is not local. Nevertheless, RWRS is a useful proof method in scenarios where number of steps required to reconcile with a state on the right can be bounded by a constant.

Definition 22 (Reduced Well-founded Reconciling Simulation (RWRS))**.** $B \subseteq S \times S$ is a reduced well-founded reconciling relation on a TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(RWRS1) $\langle \forall s, w \in S : sBw : L.s = L.w \rangle$

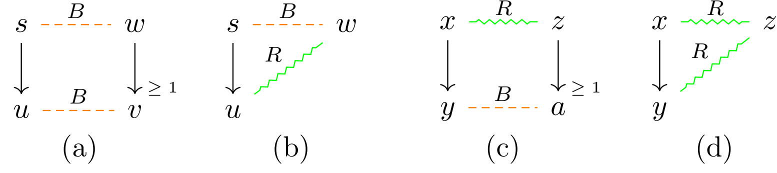

Figure 3.2: Reduced well-founded reconciling simulation (a solid line with an arrow indicates the transition relation, a dashed orange line indicates that states are related by B and a squiggly green line indicates that states are related by R).

(RWRS2) There exist a function $\mathit{rankts}\colon S \times S \longrightarrow W$ such that $\langle W, \prec \rangle$ is well founded and a binary relation $R \subseteq S \times S$ such that

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

(a) $\langle \exists v\colon w \rightarrow^{+} v\colon uBv \rangle \vee$

(b) $uRw \rangle$

and

$\langle \forall x, y, z \in S : xRz \wedge x \rightarrow y :$

(c) $\langle \exists a : z \rightarrow^{+} a : yBa \rangle \vee$

(d) $yRz \wedge \mathit{rankts}(y, z) \prec \mathit{rankts}(x, z) \rangle$

RWRS characterizes reconciling simulation, *i.e.*, it is a sound and complete proof method for reconciling simulation. The soundness of RWRS, *i.e.*, RWRS implies RES, is proved later in Theorem 36. Here we prove the completeness, *i.e.*, RES implies RWRS. Towards this, we first introduce some definitions and lemmas.

Definition 23 (ranktsCt)**.** Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and B be an RES on $\mathcal{M}$. Given $x, z \in S$, $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ is the empty tree if $\neg\langle \exists s : s \rightarrow^{+} x : sBz \rangle$, otherwise $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ is the largest subtree of $\mathit{ctree}(\mathcal{M}, x)$ such that for any node $\langle x, \ldots, y \rangle$ in $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$, we have that $\langle \forall a : z \rightarrow^{+} a : \neg(yBa) \rangle$.

Remark 1. Observe that the computation tree $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ does not depend

on the choice of $s \in S$ that can reach x and is related by B to z.

A basic property of our construction is the finiteness of paths in *ranktsCt*.

Lemma 29. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system, B be an RES on $\mathcal{M}$, and $x, z \in S$. Every path of $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ is finite.

Proof: We consider two cases. First, let $\neg\langle \exists s : s \rightarrow^{+} x : sBz \rangle$. Then $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ is empty and the conclusion trivially holds. Second, let $s \in S$ such that $s \rightarrow^{+} x$ and sBz. The proof for this case is by contradiction. We start by assuming that there exists an infinite path σ in $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$. From any such path we can construct a fullpath, say σ, that starts at s and $\sigma.i = x$ for some positive integer i. Since sBz, and B is an RES on $\mathcal{M}$, from Definition 20, there is a fullpath δ starting at z such that $\mathit{rmatch}(B, \sigma, \delta)$ holds. From Definition 19, there is a $k > i$ and a state in δ that is related by B to $\sigma.k$. However, by assumption and from Definition 23 of *ranktsCt*, we have for all $j \geq i$, $\langle x = \sigma.i, \ldots, \sigma.j \rangle$ is a node in $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ and z cannot reach a state that is related by B to $\sigma.j$. In particular, $\langle x = \sigma.i, \ldots, \sigma.k \rangle$ is a node in $\mathit{ranktsCt}(\mathcal{M}, B, x, z)$ and z cannot reach a state that is related by B to $\sigma.k$. We have our contradiction.

□

Given Lemma 29, we define a function *size*, that given a $\mathit{ctree}(\mathcal{M}, s)$, say T, all of whose paths are finite, assigns an ordinal to T. The ordinal assigned to a node $x \in T$ is defined as follows: if x is a leaf node in T then $\mathit{size}(T, x) = 0$, else $\mathit{size}(T, x) = (\bigcup_{c \in \mathit{children}(T,x)} \mathit{size}(T, c)) + 1$, where $\mathit{children}(T, x)$ returns a subset of nodes that are immediate successor of x in T. The *size* of a computation tree is the *size* of its root.

Lemma 30 ([41]). If $|S| \preceq \kappa$, where $\omega \preceq \kappa$ then for all $s, w \in S$, $\mathit{size}(\mathit{ranktsCt}(\mathcal{M}, s, w))$ is an ordinal of cardinality $\preceq \kappa$.

We use the cardinal $max(|S|^+, \omega)$, where $|S|^+$ is the cardinal successor of the size of the state space, as the well-founded domain for *rankts* in Definition 22. Next we define the binary relation F_R such that the second universal quantifier in RWRS2 of Definition 22 holds.

Definition 24 (F_R)**.** Let $\mathcal{M} = \langle S, \rightarrow, L\rangle$ be a transition system and B be an RES on $\mathcal{M}$. We define $F_R(\mathcal{M}, B)$ as the set of pairs $(x, z) \in S \times S$ such that $ranktsCt(\mathcal{M}, B, x, z)$ is non-empty.

Lemma 31. Let $\mathcal{M} = \langle S, \rightarrow, L\rangle$ be a transition system, B be an RES on $\mathcal{M}$, $x, y, z \in S$, $x \rightarrow y$, and $(x, z) \in F_R(\mathcal{M}, B)$. If $\langle \forall a : z \rightarrow^+ a : \neg(yBa)\rangle$ then $(y, z) \in F_R(\mathcal{M}, B)$ and $size(ranktsCt(\mathcal{M}, y, z)) \prec size(ranktsCt(\mathcal{M}, x, z))$.

Proof: To show that $(y, z) \in F_R(\mathcal{M}, B)$, we show that $ranktsCt(\mathcal{M}, B, y, z)$ is non-empty. Towards this observe that there is a $s \in S$ such that $s \rightarrow^+ y \wedge sBz$. This is because by assumption $x \rightarrow y$ and from Definition 24 and Definition 23, $ranktsCt(\mathcal{M}, B, x, z)$ is non-empty and $\langle \exists s : s \rightarrow^+ x \wedge sBz\rangle$. Also, $\langle x, y\rangle \in ranktsCt(\mathcal{M}, B, x, z)$. This is because, by assumption, $x \rightarrow y$ and $\langle \forall a : z \rightarrow^+ a : \neg(yBa)\rangle$. Hence, $ranktsCt(\mathcal{M}, B, y, z)$ is non-empty. Also, since $y \in children(x)$ in the tree $ranktsCt(\mathcal{M}, B, x, z)$, from the definition of *size* above, we infer that $size(ranktsCt(\mathcal{M}, y, z)) \prec size(ranktsCt(\mathcal{M}, x, z))$.

□

We are now ready to prove the completeness of RWRS.

Theorem 32 (Completeness)**.** Let $\mathcal{M}$ be a transition system. If B is an RES on $\mathcal{M}$, then B is an RWRS on $\mathcal{M}$.

Proof: Let B be an RES on $\mathcal{M}$. RWRS1 follows directly from RES1. To show that RWRS2 holds, let W be $max(|S|^+, \omega)$. Let $a, b \in S$, $rankts(a, b)$ be $size(ranktsCt(\mathcal{M}, a, b))$, and $R = F_R(\mathcal{M}, B)$ be as defined in Definition 24. From

Lemma 31, we have that R and the rank function *rankts* satisfy the second universal quantifier in RWRS2 of Definition 22.

Next let $s, u, w \in S$, $s \rightarrow u$, and sBw. We consider two cases. First, suppose that $\langle \exists v : w \rightarrow^{+} v : uBv \rangle$ holds, then RWRS2a holds. If not, then $\langle \forall v : w \rightarrow^{+} v : \neg(uBv) \rangle$. Hence, $rankts Ct(\mathcal{M}, B, u, w)$ is non-empty. This is because B is an RES on $\mathcal{M}$, $s \rightarrow u$, and sBw. Finally, from the definition of F_R we have that $(u, w) \in F_R$ and uRw holds, *i.e.*, RWRS2b holds.

□

3.5.2 Well-founded Reconciling Simulation

Next, we introduce the notion of *well-founded reconciling simulation* that requires only local reasoning, *i.e.*, reasoning about states and their successors. Unlike RWRS, it can be used to effectively analyze systems that may take finite but unbounded number of steps to reconcile on the left.

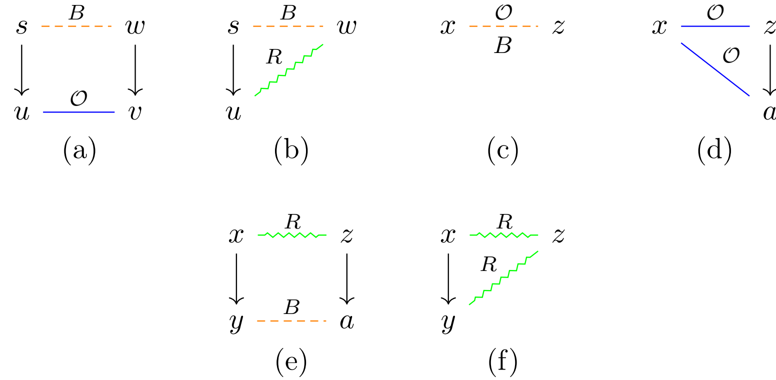

Figure 3.3: Well-founded reconciling simulation (a solid line with an arrow indicates the transition relation, a dashed orange line indicates that states are related by B, a solid blue line indicate the states are related by $\mathcal{O}$, and a squiggly green line indicates that states are related by R).

Definition 25 (Well-founded Reconciling Simulation (WRS))**.** $B \subseteq S \times S$ is a well-founded reconciling relation on a TS $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(WRS1) $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$

(WRS2) There exist functions, $\mathit{rankls} \colon S \times S \longrightarrow \omega$, and $\mathit{rankts} \colon S \times S \longrightarrow W$ such that $\langle W, \prec \rangle$ is well founded, and binary relations $R, \mathcal{O} \subseteq S \times S$ such that

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

(a) $\langle \exists v \colon w \rightarrow v \colon u\mathcal{O}v \rangle \vee$

(b) $uRw \rangle$

and

$\langle \forall x, z \in S : x\mathcal{O}z :$

(c) $xBz \vee$

(d) $\langle \exists a : z \rightarrow a : x\mathcal{O}a \wedge \mathit{rankls}(a, x) < \mathit{rankls}(z, x) \rangle$

and

$\langle \forall x, y, z \in S : xRz \wedge x \rightarrow y :$

(e) $\langle \exists a \colon z \rightarrow a \colon y\mathcal{O}a \rangle \vee$

(f) $yRz \wedge \mathit{rankts}(y, z) \prec \mathit{rankts}(x, z) \rangle$

Intuitively, the condition RWRS2a in Definition 22, that required reachability analysis, is replaced by conditions WRS2a, WRS2c and WRS2d in Definition 25. Similarly, the condition RWRS2c in Definition 22, that required reachability analysis, is replaced by conditions WRS2e, WRS2c and WRS2d in Definition 25. To show that a binary relation is a WRS, we have to provide as witness binary relations R and $\mathcal{O}$, and rank functions *rankts* and *rankls*. $\mathcal{O}$ and *rankls* enable local reasoning about reconciliation on the right and R and *rankts* as was the case with RWRS enable local reasoning about reconciliation on the left. In fact, RWRS and WRS

are equivalent. Here we show that RWRS implies WRS. The other direction, WRS implies RWRS, is inferred from Theorem 36 proved later in the section.

Theorem 33. Let $\mathcal{M}$ be a transition system. If B is an RWRS on $\mathcal{M}$, then B is a WRS on $\mathcal{M}$.

Proof: Let B be an RWRS on $\mathcal{M}$. WRS1 follows directly from RWRS1. Next we have to show that RWRS2 implies WRS2. Let $\langle W, \prec \rangle$ be a well-founded domain, $rankts : S \times S \to W$ and $R \subseteq S \times S$ such that RWRS2 holds. To show that WRS2 holds, we use the same rank function $rankts$ and the binary relation R as in RWRS2. The binary relation $\mathcal{O} \subseteq S \times S$ is defined as follows: $\mathcal{O} = \{(u, v) : \langle \exists v' : v \to^* v' : uBv' \rangle\}$. The rank function $rankls : S \times S \to \omega$ is defined as follows: let $u, v \in S$ then $rankls(u, v)$ is the minimal length of an $\mathcal{M}$-segment that starts at v and ends in a state that is related to u by B, if such a segment exists and 0 otherwise.

Let $s, u, w \in S$, $s \to u$, and sBw. If RWRS2b holds then WRS2b holds. Next suppose that RWRS2b does not hold and RWRS2a holds, *i.e.*, $\langle \exists v : w \to^+ v : uBv \rangle$. Let $\langle w, v_1, \ldots, v_k = v \rangle$, where $k \geq 1$, be such an $\mathcal{M}$-segment. Hence, from construction of $\mathcal{O}$ we have that $u\mathcal{O}v_1$, and WRS2a holds.

Next let $x, y, z \in S$, xRz, and $x \to y$. Since R satisfies the second conjunct in RWRS2, one of RWRS2c or RWRS2d must hold. If RWRS2d holds then WRS2f holds. Next suppose RWRS2d does not hold and RWRS2c holds, *i.e.*, $\langle \exists a : z \to^+ a : yBa \rangle$. Let $\langle z, a_1, \ldots, a_k = a \rangle$, where $k \geq 1$, be such an $\mathcal{M}$-segment. Hence. from the definintion of $\mathcal{O}$, we have that $y\mathcal{O}a_1$, *i.e.*, WRS2e holds.

To finish the proof we show that $\mathcal{O}$ and rank function $rankls$, as defined above, satisfy the constraints imposed by the second conjunct in WRS2. Let $x, y \in S$, $x\mathcal{O}y$, and WRS2c does not hold, *i.e.*, $\neg(xBy)$. From the definition of $\mathcal{O}$, we have that there is an $\mathcal{M}$-segment from y to a state related to x by B; let $\vec{y}$ be such a segment with

the minimal length. From the definition of *rankls*, we have that $\mathit{rankls}(y, x) = |\overrightarrow{y}|$. Since the last state in $\overrightarrow{y}$ must be related to x by B, and from the assumption $\neg(xBy)$, we have that y cannot be the last state of $\overrightarrow{y}$ and $|\overrightarrow{y}| > 1$. Let y' be the successor of y in $\overrightarrow{y}$. Then from construction of $\mathcal{O}$, we have that $x\mathcal{O}y'$. Observe that the length of the minimal $\mathcal{M}$-segment from y' to a state that is related to x by B must be less or equal to $|\overrightarrow{y}| - 1$. Hence $\mathit{rankls}(y', x) = |\overrightarrow{y}| - 1 < \mathit{rankls}(y, x)$.

□

3.5.3 Well-founded Reconciling Simulation with Explicit Stuttering

Next we introduce the notion of *well-founded reconciling simulation with explicit stuttering* (WRSS). Like WRS, it only requires reasoning about a state and its successors. However, unlike WRS, it distinguishes between stuttering and reconciling.

Figure 3.4 illustrates the conditions in WRSS. The intuition is, for any pair of states s, w which are related by B, a state u such that $s \rightarrow u$, there are five cases to consider (a) either we can rmatch the move from s to u with a single step from w, or (b) there is stuttering on the left, or (c) there is stuttering on the right, or (d) there is reconciling on the right, or (e) there is reconciling on the left. Additionally conditions (f) and (g) together ensure that reconciling on the right is finite, and conditions (h) and (i) ensure that reconciling on the left is finite.

Definition 26 (Well-founded Reconciling With Explicit Stuttering (WRSS))**.** $B \subseteq S \times S$ is a well-founded reconciling relation with explicit stuttering on a transition system $\mathcal{M} = \langle S, \rightarrow, L \rangle$ iff:

(WRSS1) $\langle \forall s, w \in S \colon sBw \colon L.s = L.w \rangle$

(WRSS2) There exist functions, $\mathit{rankt} \colon S \times S \longrightarrow W$, $\mathit{rankl} \colon S \times S \times S \longrightarrow \omega$, and $\mathit{rankls} \colon S \times S \longrightarrow \omega$, and $\mathit{rankts} \colon S \times S \longrightarrow W$ such that $\langle W, \prec \rangle$ is well founded,

and binary relations $R, \mathcal{O} \subseteq S \times S$ such that

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

(a) $\langle \exists v \colon w \rightarrow v \colon uBv \rangle \vee$

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

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

(d) $\langle \exists v \colon w \rightarrow v \colon u\mathcal{O}v \rangle \vee$

(e) $uRw \rangle$

and

$\langle \forall x, z \in S : x\mathcal{O}z :$

(f) $xBz \vee$

(g) $\langle \exists a : z \rightarrow a : x\mathcal{O}a \wedge \mathit{rankls}(a, x) < \mathit{rankls}(z, x) \rangle \rangle$

and

$\langle \forall x, y, z \in S : xRz \wedge x \rightarrow y :$

(h) $\langle \exists a \colon z \rightarrow a \colon y\mathcal{O}a \rangle \rangle \vee$

(i) $yRz \wedge \mathit{rankts}(y, z) \prec \mathit{rankts}(x, z) \rangle$

We next prove that WRSS implies RES.

Theorem 34. Let $\mathcal{M}$ be a transition system. If B is a WRSS on $\mathcal{M}$ then B is an RES on $\mathcal{M}$.

Proof: Let B be a WRSS on $\mathcal{M} = \langle S, \rightarrow, L \rangle$, $x, y \in S$, and xBy. To show that B is an RES on $\mathcal{M}$, we show that RES1 and RES2 hold. RES1 follows directly from WRSS1.

Next we show that RES2 holds, *i.e.*, for any fullpath σ starting at x in $\mathcal{M}$, there is a fullpath δ starting at y in $\mathcal{M}$ such that $\mathit{rmatch}(B, \sigma, \delta)$ holds. We start by recursively defining $\pi, \xi \in \mathit{INC}$, and a fullpath δ in $\mathcal{M}$. For the base case,

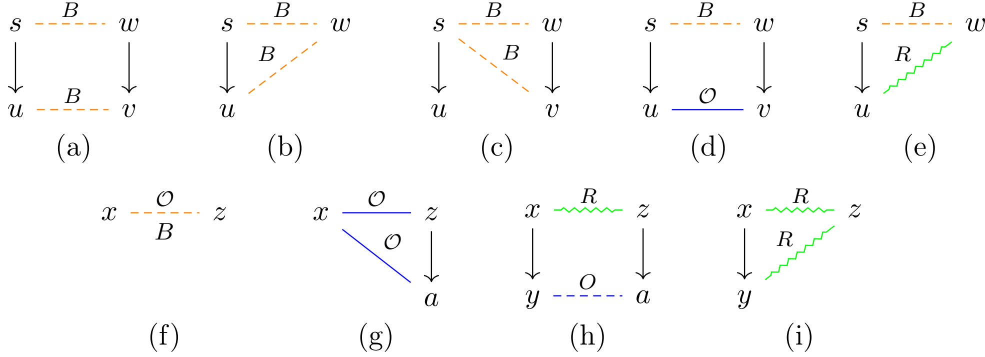

Figure 3.4: Well-founded reconciling simulation with explicit stuttering (a solid line with an arrow indicates the transition relation, a dashed orange line indicates that states are related by B, a solid blue line indicate the states are related by $\mathcal{O}$, and a squiggly green line indicates that states are related by R).

let $\pi.0 = 0, \xi.0 = 0$, and $\delta.0 = y$. From assumption, we have that $\sigma(\pi.0)B\delta(\xi.0)$. For the recursive case, we assume that $\pi.0, \ldots, \pi.i$, $\xi.0, \ldots, \xi.i$, and $\delta.0, \ldots, \delta(\xi.i)$ are defined, and that $\sigma(\pi.i)B\delta(\xi.i)$. Let $s = \sigma(\pi.i)$, $u = \sigma(\pi.i{+}1)$, and $w = \delta(\xi.i)$. Since WRSS2 holds there exists a well-founded domain $\langle W, \prec \rangle$, rank functions *rankt*, *rankl*, *rankls*, *rankts*, and binary relation $R, \mathcal{O}$ satisfying the three conjuncts in WRSS2. Before we proceed, we prove the following helpful lemma.

Lemma 35. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system, $x, z \in S$, $\mathcal{O}, B \subseteq S \times S$, and $\mathit{rankls} : S \times S \rightarrow \omega$. Let $\mathcal{O}, B$, and *rankls* satisfy the second universal quantifier in WRSS2. If $x\mathcal{O}z$ then there is a non-empty finite $\mathcal{M}$-segment starting from z to a state that is related to x by B.

Proof: Let $x\mathcal{O}z$. First suppose xBz holds. Then $\langle z \rangle$ is an $\mathcal{M}$-segment such that xBz. Next, suppose $\neg(xBz)$ and $\langle \exists a : z \rightarrow a : x\mathcal{O}a \wedge \mathit{rankls}(a, x) < \mathit{rankls}(z, x) \rangle$. Let δ be an $\mathcal{M}$-path starting at z such that only WRSS2g holds, *i.e.*, for all $i \geq 0$, $x\mathcal{O}\delta.i$ and $\mathit{rankls}(\delta(i+1), x) < \mathit{rankls}(\delta.i, x)$ and $\neg(xB\delta.i)$. Observe that δ must be a finite path in $\mathcal{M}$. Otherwise, it contradicts the well-foundedness of *rankls*. Let a be the last state of δ. Also since $x\mathcal{O}a$ and $\neg\langle \exists a' : a \rightarrow a' : x\mathcal{O}a' \wedge \mathit{rankls}(a', x) <$

$rankls(a, x)\rangle$, it must be the case that xBa holds.

□

We now return to the proof of Theorem 34. We consider five cases.

1. Suppose WRSS2a holds, *i.e.*, there is a successor v of w such that uBv holds. Let u and v mark the beginning of the next segment in σ and δ respectively. We define $\pi(i+1) = \pi.i + 1$, $\xi(i+1) = \xi.i + 1$, and ${}^{\xi}\delta^{i} = \langle w \rangle$.

2. Suppose WRSS2a does not hold and WRSS2d holds, *i.e.*, there is a successor v of w such that $u\mathcal{O}v$. From the second universal quantifier in WRSS2, and Lemma 35, we have that there is an $\mathcal{M}$-segment $\langle v_1 = v, \ldots, v_m \rangle$, where $m \geq 1$ such that uBv_m. Let u and v_m mark the beginning of the next segment in σ and δ respectively. We define $\pi(i+1) = \pi.i + 1$, $\xi(i+1) = \xi.i + m$, ${}^{\xi}\delta^{i} = \langle w, \ldots, v_{m-1} \rangle$, and $\delta(\xi(i+1)) = v_m$.

3. Suppose WRSS2a and WRSS2d do not hold and WRSS2e holds, *i.e.*, $\neg\langle \exists v : w \to v : uBv \rangle$, $\neg\langle \exists v : w \to v : u\mathcal{O}v \rangle$ and uRw. Let $\sigma(\pi.i{+}2) = u'$. We consider two cases. First, suppose that WRSS2h holds between u, u', and w, *i.e.*, there is a successor v of w such that $u'\mathcal{O}v$. From the second universal quantifier in WRSS2 and Lemma 35, we have that there is an $\mathcal{M}$-segment $\langle v_1 = v, \ldots, v_m \rangle$, where $m \geq 1$ such that $u'Bv_m$ holds. Let u' and v_m mark the beginning of the next segment in σ and δ respectively. We define $\pi(i+1) = \pi.i + 2$, $\xi(i+1) = \xi.i + m$, ${}^{\xi}\delta^{i} = \langle w, \ldots, v_{m-1} \rangle$ and, $\delta(\xi(i+1)) = v_m$. Second, suppose that WRSS2h does not hold and WRSS2i holds between u, u', and w. We define J to be the subset of the positive integers ≥ 2 such that for every $j \in J$, the following holds.

$$\langle \forall v : w \to v : \neg(\sigma(\pi.i + j)\mathcal{O}v)\rangle \;\wedge \tag{3.1}$$
$$\langle \sigma(\pi.i + j)Rw \;\; \wedge \;\; \mathit{rankts}(\sigma(\pi.i + j), w) \prec \mathit{rankts}(\sigma(\pi.i + j - 1), w)\rangle$$

First observe that $2 \in J$ because $\sigma(\pi.i + 2) = u'$, WRSS2h does not hold by assumption (so the first conjunct is true) and WRSS2i does (so the second conjunct is true). Next observe that there is an integer $n > 2$ such that $n \notin J$. Suppose not, then for all $n \geq 2, n \in J$. Now, consider the (infinite) suffix of σ starting at $\sigma(\pi.i + 1)$. For every adjacent pair of states in this suffix, say $\sigma(\pi.i + k)$ and $\sigma(\pi.i + k + 1)$ where $k \geq 1$, we have that only WRSS2i holds (*i.e.*, WRSS2h does not hold). This gives us a contradiction because *rankts* is well-founded. We now define $n = \mathit{min}(\{l \in \omega : l > 2 \wedge l \notin J\})$. Notice that only WRSS2i holds between $\sigma(\pi.i + n - 1)), \sigma(\pi.i + n)$ and w, hence $\sigma(\pi.i + n)Rw$ and $\mathit{rankts}(\sigma(\pi.i + n), w) \prec \mathit{rankts}(\sigma(\pi.i + n - 1), w)$. Since Formula 3.1 does not hold for n, one of the conjuncts has to be false, and above we determined that it must be the first one, *i.e.*, there is a successor v of w such that $\sigma(\pi.i+n)\mathcal{O}v$. From the third universal quantifier in WRSS2 and Lemma 35, we have that there is an $\mathcal{M}$-segment $\langle v_1 = v, \ldots, v_m\rangle$, where $m \geq 1$, such that $\sigma(\pi.i+n)Bv_m$. We are now ready to extend our recursive definitions as follows: $\pi(i + 1) = \pi.i + n$, $\xi(i + 1) = \xi.i + m$, and ${}^{\xi}\delta^{i} = \langle w, \ldots, v_{m-1}\rangle$.

4. Suppose only WRSS2b holds between $s = \sigma(\pi.i), u = \sigma(\pi.i+1)$ and w. Let K be the set of positive integers such that for every $k \in K$ only WRSS2b holds between $\sigma(\pi.i + k - 1), \sigma(\pi.i + k)$, and w. First observe that $1 \in K$. This is because by assumption $\sigma(\pi.i)Bw$, $\sigma(\pi.i + 1)Bw$, and $\mathit{rankt}(\sigma(\pi.i + 1), w) \prec \mathit{rankt}(\sigma(\pi.i), w)$. Next observe that there exists an integer $n > 1$ such that $n \notin K$. Suppose not, then for all $n \geq 1$, $n \in K$. Consider the infinite suffix of σ starting at $\sigma(\pi.i)$ and the next state $\sigma(\pi.i + 1)$. For every adjacent pair of

states in this suffix, say $\sigma(\pi.i+l-1)$ and $\sigma(\pi.i+l)$ where $l \geq 1$, only WRSS2b holds. This gives us a contradiction because *rankt* is well-founded. We now define $n = min(\{l \in \omega : l > 1 \wedge l \notin K\})$. Since $\sigma(\pi.i+n-1)Bw$ and WRSS2b does not hold between $\sigma(\pi.i+n-1)$, $\sigma(\pi.i+n)$, and w, it must be that one of WRSS2a, WRSS2c, WRSS2d, or WRSS2e must between $\sigma(\pi.i+n-1)$, $\sigma(\pi.i+n)$, and w.

If WRSS2c holds, *i.e.*, there exists a v such that $w \rightarrow v$, $\sigma(\pi.i+n-1)Bv$, and $rankls(v, \sigma(\pi.i+n-1), \sigma(\pi.i+n)) < rankls(w, \sigma(\pi.i+n-1), \sigma(\pi.i+n))$holds. In this case, we extend our recursive definitions as follows: let $\pi(i+1) = \pi.i+n-1$, $\xi(i+1) = \xi.i+1$ and ${}^{\xi}\delta^i = \langle w \rangle$.

If WRSS2c does not hold and WRSS2a holds, then there is a v such that $w \rightarrow v$ and $\sigma(\pi.i+n)Bv$. In this case we extend our recursive definitions as follows: let $\pi(i+1) = \pi.i+n$, $\xi(i+1) = \xi.i+1$ and ${}^{\xi}\delta^i = \langle w \rangle$

Next if WRSSS2c and WRSS2a do not hold and WRSS2d holds, then from the second universal quantifier in WRSS2 and Lemma 35,there is an $\mathcal{M}$-segment $\langle w = v_0, \ldots, v_m \rangle$, where $m \geq 1$, such that $\sigma(\pi.i+n)Bv_m$. We extend our recursive definitions as follows: $\pi(i+1) = \pi.i+n$, $\xi(i+1) = \xi.i+m$ and ${}^{\xi}\delta^i = \langle w, \ldots, v_{m-1} \rangle$.

Finally, if WRSS2a, WRSS2c, and WRSS2d do not hold and WRSS2e holds, then as in the third case above, there exists $p, m \geq 1$ such that we can extend our recursive definition as follows: $\pi(i+1) = \pi.i+n+p$, $\xi(i+1) = \xi.i+m$, and ${}^{\xi}\delta^i = \langle w, \ldots, v_{m-1} \rangle$.

5. Finally, we consider the case when only WRSS2c holds between s, u and w, *i.e.*, there is a successor v of w such that sBv and $rankl(v, s, u) < rankl(w, s, u)$. Let γ be an $\mathcal{M}$-path starting at w such that only WRSS2c holds between

s, u and a state in γ. There is a non-empty path that satisfy this condition, *e.g.*, let $\gamma = \langle w \rangle$. Notice that any such $\mathcal{M}$-path must be finite. If not, then for any adjacent pair of states in γ, say $\gamma.k$ and $\gamma(k+1)$, where $k \in \omega$, $rankl(\gamma(k+1), s, u) < rankl(\gamma.k, s, u)$, which contradicts the well-foundedness of $rankl$. Let $\vec{v} = \langle v_0 = w, \ldots, v_m \rangle$, where $m \geq 1$, be a maximal $\mathcal{M}$-segment satisfying the above condition. Since sBv_m holds, one of WRSS2a, WRSS2b, WRSS2d, or WRSS2e must hold between s, u, and v_m.

Suppose WRSS2a holds, then, as in the first case above, we can extend our recursive definitions as follows: $\pi(i+1) = \pi.i + 1$, $\xi(i+1) = \xi.i + m + 1$, and ${}^{\xi}\delta^{i} = \langle w, \ldots, v_m \rangle$.

Suppose WRSS2a does not hold and WRSS2b holds between s, u, v_m, *i.e.*, uBv_m and $rankt(u, v_m) \prec rankt(s, v_m)$. In this case we extend our recursive definitions as follows: $\pi(i+1) = \pi.i + 1$, $\xi(i+1) = \xi.i + m$, and ${}^{\xi}\delta^{i} = \langle v_0 = w, \ldots, v_{m-1} \rangle$.

Suppose WRSS2a does not hold, and WRSS2d holds. Then, as in second case above, we can extend our recursive definitions as follows: there exists $n \geq 1$ such that $\pi(i+1) = \pi.i + 1$, $\xi(i+1) = \xi.i + m + n$ and ${}^{\xi}\delta^{i} = \langle w, \ldots, v_{m+n-1} \rangle$.

Suppose WRSS2a and WRSS2d do not hold and WRSS2e hold. Then, as in the third case above, we can extend our recursive definitions as follows: there exits $p > 2$ and $n \geq 1$ such that $\pi(i+1) = \pi.i + p$, $\xi(i+1) = \xi.i + m + n$, and ${}^{\xi}\delta^{i} = \langle w, \ldots, v_{m+n-1} \rangle$.

Now we show that RES2 holds. We start by unwinding definitions. The first step is to show that $fp.\delta.y$ holds, which is true by construction. Next, we show that $rmatch(B, \sigma, \delta)$ by unwinding the definition of $rmatch$. That involves showing that there exist π and ξ such that $rcorr(B, \sigma, \pi, \delta, \xi)$ holds. The π and ξ we used to

define δ can be used here. Finally, we unwind the definition of *rcorr*, which gives us a universally quantified formula over the natural numbers. This is handled by induction on the segment index; the proof is based on the recursive definitions given above.

□

The following theorem states that all the three characterizations of reconciling simulation introduced above are equivalent.

Theorem 36. Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system and $B \subseteq S \times S$. The following are equivalent

(i) B is an RES on $\mathcal{M}$;

(ii) B is an RWRS on $\mathcal{M}$;

(iii) B is a WRS on $\mathcal{M}$;

(iv) B is WRSS on $\mathcal{M}$

Proof: (i) implies (ii) follows from Theorem 32, (ii) implies (iii) follows from Theorem 33, and (iv) implies (i) in Theorem 34. To conclude the proof, we show that (iii) implies (iv).

WRSS1 follows directly from WRS1. To prove that WRSS2 holds, we use the same rank functions *rankts* and *rankls* and binary relations R and $\mathcal{O}$. If WRS2a or WRS2b holds then WRSS2d or WRSS2e holds. Clearly, $\mathcal{O}$ and R satisfy the second and the third conjunct in WRSS2.

□

3.6 Summary

In this chapter, we introduced a new notion of reconciling simulation, studied its algebraic properties, and based on it developed a theory of reconciling refinement. Unlike skipping refinement, reconciling refinement is not compositional. We developed three alternative characterizations of reconciling simulation that are amenable for mechanised reasoning using formal-methods tools.

Well-founded reconciling simulation (Definition 25) and well-founded reconciling simulation with explicit stuttering (Definition 26) are complete proof methods that require only local reasoning, *i.e.*, reasoning about states and their successors. Thus, if a concrete system is reconciling refinement of an abstract system under a refinement map, one can always prove it using local reasoing about states and their successors and do not require global reasoning about infinite paths. Furthermore, reconciling simulation is a strictly weaker notion of correctness than skipping simulation (and stuttering simulation). Together, these results significantly extend the domain of applicability of refinement-based methodology to verification of a large class of reactive systems.

The approach we followed towards developing a sound and complete proof methods for mechanised reasoning of reconciling simulation is similar to the approach for skipping simulation in Chapter 2. We first defined an intermediate characterization, RWRS (Definition 22), that localizes reasoning on the left by introducing an auxiliary binary relation on states (conditions RWRS2b-2d), while reasoning about reconciling on the left is accounted for by transitive closure (condition RWRS2a) and in general requires reachability analysis. Then, similar to RLWFSK (Definition 17), we defined WRS (Definition 25) that localizes reasoning about reconciling on the right by introducing another auxiliary binary relation on states (conditions WRS2a, WRS2c-2d). Finally, similar to LWFSK (Definition 18), we define WRSS that ex-

plicitly accounts for stuttering. In fact, one can look at LWFSK as WRSS modulo conditions WRSS2e, WRSS2h-2i, conditions that account for reconciling on the left, a scenario that does not occur when reasoning about skipping simulation.

Finally, observe that one can envision defining other equivalent characterizations. For example, we can define a notion that distinguishes between stuttering and reconciling on the left (as in condition WRSS2b and WRSS2e) and use transitive closure (as in RWRS2a) to combine stuttering and skipping the right. Such a characterization is justified as it can be useful when skipping on the right is bounded (and small). The three characterizations of reconciling simulation that we choose to study are interesting in the sense that they highlight the considerations when designing a sound and complete proof method that involves only local reasoning.

Chapter 4

Case Studies

In this chapter, we present case studies that enable us to explore the conceptual aspects of the theory of skipping refinement and the applicability of associated proof methods. When verifying that an implementation system refines a specification system two primary choices are: (1) selecting an appropriate notion of refinement, and (2) selecting a proof-method that can be used to mechanically prove it. We highlight these choices in analysing correctness of four optimized reactive systems (1) a vectorizing compiler transformation (2) a JVM-inspired stack machine, (3) an optimized memory controller, and (4) an asynchronous event-processing system. (1), (2), and (3) are examples of optimized reactive systems that exhibit bounded skipping while (4) is an example of an optimized reactive system that exhibits unbounded (but finite) skipping. To facilitate understanding and focus on evaluating the theory of refinement, we only model certain aspects of the systems. All case studies are performed using ACL2s, an interactive theorem prover [13]. In particular, for case study (4), we present in detail, a formalization of the specification system and the implementation system and the proof of correctness in ACL2s. Using finite state models of the systems in case studies (2) and (3), we also evaluate the impact of

these choices on the specification cost and the effectiveness of proof methods when using model-checkers. We compare two notions of correctness: skipping refinement and input-output equivalence. The finite-state models of the systems are compiled to sequential AIGs. We will see that when correctness is based on input-output equivalence, model-checkers quickly start timing out as the complexity (size) of the systems increases. In contrast, when skipping refinement is used, much larger systems can be automatically verified. The proof scripts are publicly available [1].

4.1 Superword Level Parallelism with SIMD instructions

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 analyse programs for *superword level parallelism* and when possible replace multiple scalar instructions with a compact SIMD instruction that operates on multiple data in parallel [35]. In this case study, we illustrate the applicability of skipping refinement to verify the correctness of such a compiler transformation using ACL2s. The source language consists of scalar instructions and the target language consists of both scalar and vector instructions. We model the transformation as a function that takes as input a program in the source language and outputs a program in the target language. Instead of proving that the compiler is correct, we use the translation validation approach [6] and prove the equivalence of the source program and the generated target program.

We make some simplifying assumptions for modeling purpose: the state of the source and target programs (modeled as transition systems) is a three-tuple consisting of a sequence of instructions, a program counter, and a store. We also assume that a SIMD instruction operates on *two* sets of data operands simultaneously and

$$\begin{matrix} a & = & b & + & c \\ d & = & e & + & f \end{matrix} \rightarrow \boxed{\begin{matrix} a \\ d \end{matrix}} = \boxed{\begin{matrix} b \\ e \end{matrix}} +_{SIMD} \boxed{\begin{matrix} c \\ f \end{matrix}}$$

$$\begin{matrix} u & = & v & \times & w \\ x & = & y & \times & z \end{matrix} \rightarrow \boxed{\begin{matrix} u \\ x \end{matrix}} = \boxed{\begin{matrix} v \\ y \end{matrix}} \times_{SIMD} \boxed{\begin{matrix} w \\ z \end{matrix}}$$

Figure 4.1: An example of superword parallelism optimization

that the transformation identifies parallelism at the basic block level. Therefore, we do not model any control flow instructions. Note that we do *not* reorder instructions in the source program. Figure 4.1 shows how two add and two multiply scalar instructions are transformed into corresponding SIMD instructions.

The syntax and operational semantics of source and target programs are given in Figure 4.2. Adding element e to the beginning or end of list (or array) l is denoted by $e :: l$ and l::e, respectively. Each transition consists of a state:condition pair above a line, followed by the next state below the line. If a state matches the state in a transition and satisfies the conditions associated with it, then the state can transition to the state below the line. Every condition implicitly contains the condition $pc < n$. We denote that $x, \ldots, y$ are variables with values $v_x, \ldots, v_y$ in *store* by $\{\langle x, v_x\rangle, \ldots, \langle y, v_y\rangle\} \subseteq store$.$[\![(sop\ v_x\ v_y)]\!]$ denotes the result of the scalar operation *sop* and $[\![(vop\ \langle v_a\ v_b\rangle\langle v_d\ v_e\rangle)]\!]$ denotes the result of the vector operation *vop*. Finally, we use $store|_{x:=v_x,\ldots,y:=v_y}$ to denote that variables $x, \ldots, y$ are updated (or added) to *store* with values $v_x, \ldots, v_y$.

What is an appropriate notion of refinement to analyse the correctness of the compiler transformation? Since the target program can execute multiple scalar instructions in a single step, the notion of stuttering simulation is too strong to relate the source and the target program produced by the compiler, no matter what refinement map we use. To see this, note that the target program might run exactly twice as fast as the source program and during each step the scalar program might be modifying the memory. Since both the programs do not stutter, in order to use

$loc := \{x, y, z, a, b, c, \ldots\}$ (Variables)

$sop := add \mid sub \mid mul \mid and \mid or \mid nop$ (Scalar Ops)

$vop := vadd \mid vsub \mid vmul \mid vand \mid vor \mid vnop$ (Vector Ops)

$sinst := sop\langle z\ x\ y\rangle$ (Scalar Inst)

$vinst := vop\langle c\ a\ b\rangle\langle f\ d\ e\rangle$ (Vector Inst)

$sprg := [] \mid sinst :: sprg$ (Scalar Program)

$vprg := [] \mid (sinst \mid vinst) :: vprg$ (Vector Program)

$store := [] \mid \langle x, v_x\rangle :: store$ (Registers)

Scalar Machine ($\xrightarrow{A}$)

$$\frac{\langle sprg, pc, store\rangle,\ \{\langle x, v_x\rangle, \langle y, v_y\rangle\} \subseteq store,\quad sprg[pc] = sop\langle z\ x\ y\rangle,\ v_z = [\![(sop\ v_x\ v_y)]\!]}{\langle sprg, pc + 1, store|_{z:=v_z}\rangle}$$

Vector Machine ($\xrightarrow{C}$)

$$\frac{\langle vprg, pc, store\rangle,\ \{\langle x, v_x\rangle, \langle y, v_y\rangle\} \subseteq store,\quad vprg[pc] = sop\langle z\ x\ y\rangle,\ v_z = [\![(sop\ v_x\ v_y)]\!]}{\langle vprg, pc + 1, store|_{z:=v_z}\rangle}$$

$$\frac{\langle vprg, pc, store\rangle,\ vprg[pc] = vop\langle c\ a\ b\rangle\langle f\ d\ e\rangle,\quad \{\langle a, v_a\rangle, \langle b, v_b\rangle, \langle d, v_d\rangle, \langle e, v_e\rangle\} \subseteq store,\quad \langle v_c, v_f\rangle = [\![(vop\ \langle v_a\ v_b\rangle\langle v_d\ v_e\rangle)]\!]}{\langle vprg, pc + 1, store|_{c:=v_c, f:=v_f}\rangle}$$

Figure 4.2: Syntax and Semantics of Scalar and Vector Program

stuttering refinement the length of run of the source and target program must be equal, which is not in our case. On the other hand, skipping refinement directly accounts for such differences in behavior between a specification and an optimized implementation. Therefore, we use skipping refinement to analyse the correctness of the transformation.

Let $\mathcal{M}_A$ and $\mathcal{M}_C$ be transition systems of the source and target programs, respectively. The set of states S_A and S_C and the transition relation $\xrightarrow{A}$ and $\xrightarrow{C}$ are as described in Figure 4.2, and labeling function L_A and L_C are just the identity functions. We begin by defining the refinement map. Let *sprg* be the source pro-

gram and *vprg* be the compiled target program. The refinement map $R : S_C \rightarrow S_A$ is defined as follows.

$$R(\langle vprg, pc, store\rangle) = \langle sprg, pcT(pc), store\rangle$$

In the above definition, pcT is a function that maps values of the target program counter to the corresponding values of the source program counter; and *sprg* is the scalar program obtained by *scalarizing* each of the vector instructions in *vprg*. Note that scalarizing a vector program is in some sense the inverse of the compiler transformation. However, it is significantly simpler because the complexity of the compiler transformation typically lies in the analysis phase that determines if the transformation is even feasible and not in the transformation phase. Also, note that $pcT(pc)$ can alternatively be determined using a history variable.

To show that $\mathcal{M}_C$ is a skipping refinement of $\mathcal{M}_A$ under the refinement map R we must show as witness a binary relation that satisfies the two conditions in Definition 12: (1) it agrees with the refinement map R, and (2) it is a skipping simulation on the disjoint union of $\mathcal{M}_C$ and $\mathcal{M}_A$.

Let $B = \{\langle s, R.s\rangle \mid s \in S_C\}$. It is easy to see that the first condition holds for B. To show that B is a skipping simulation relation we choose RWFSK (Definition 13). This is because in this case, it is simple to determine j, an upper-bound on skipping: observe that a step of the target program corresponds to at most 2 steps of the source program; hence $j = 3$ suffices. This reduces condition RWFSK2b, that in general requires reachability analysis, to bounded reachability. Moreover, $\mathcal{M}_A$ and $\mathcal{M}_C$ do not stutter, therefore we ignore RWFSK2a and do not define *rankt*. We can further simplify our proof obligation by using our knowledge about the machines. Since the scalar machine is deterministic we can eliminate the existential quantifier

in RWFSK2b. Finally, our proof obligation is: for all $s \in S_C$ such that $s \xrightarrow{C} u$:

$$R.s \xrightarrow{A}^{<3} R.u \tag{4.1}$$

Furthermore, since the semantics of the vector machine is deterministic, u is a function of s, so we can remove u from the above formula, if we wish. Also, we can expand out $\xrightarrow{A}^{<3}$ to obtain a formula using only $\xrightarrow{A}$ instead.

For this case study we used ACL2s to model and prove the correctness of the compiler transformation. A state of the scalar machine and the vector machine is described using the Defdata framework in ACL2s [13, 12] and their operational semantics is formalized in ACL2s using standard methods. The sizes of the program and store are unbounded. After the data definitions and function definitions modeling the transition systems for the two systems were in place, we proved additional lemmas to discharge the proof obligation 4.1. The additional lemmas can be roughly categorized into two categories. First, the set of lemmas related to the data structures used to model the operational semantics of scalar and vector machines. Second, the set of lemmas that establish a relationship between the current instruction (as pointed by the program counter) in a state of the vector machine, say s, to the current instruction in $R.s$, the (scalar) state obtained by applying the refinement map. In total, we introduced 15 data definitions and 21 function definitions to model the two machines and the refinement map. We have in total 44 lemmas. The entire proof takes about 3 minutes on a machine with 2.2 Ghz Intel Core i7 with 16 GB main memory. The models and the proof script are publicly available [1].

4.2 JVM-inspired Stack Machine

In this case study we verify a stack machine inspired by the Java Virtual Machine (JVM). Java processors have been proposed as an alternative to just-in-time compilers to improve the performance of Java programs. Java processors such as JME [22] fetch bytecodes from an instruction memory and store them in an instruction buffer. The bytecodes in the buffer are analysed to perform instruction-level optimizations *e.g.*, instruction folding. In this case study, we verify BSTK, a simple hardware implementation of a part of the JVM. BSTK is an incomplete and inaccurate model of JVM that only models an instruction memory, an instruction buffer and a stack. Only a small subset of JVM instructions are supported ($push, pop, top, nop$). However, even such a simple model is sufficient to exhibit benefits of using skipping refinement and its associated proof methods for mechanical reasoning in comparison to existing notions of correctness.

A state of BSTK is a four tuple $\langle imem, pc, ibuf, stk\rangle$, where *imem* is the instruction memory, *pc* is the program counter, and *stk* is the stack. The capacity of the instruction buffer is a constant positive integer. BSTK fetches the instruction from the *imem* at location *pc*, and if the fetched instruction is not *top* and the instruction buffer is not full, it enqueues the instruction to the end of the buffer *ibuf* and increments the *pc* by 1. Otherwise, BSTK executes all buffered instructions in the order they were enqueued in a single step, thereby draining the buffer, and obtaining a new stack. To verify the correctness of BSTK, we define an abstract high-level specification STK that serves as a specification. A state of STK is a three tuple $\langle imem, pc, stk\rangle$. STK fetches an instruction from the *imem* at location *pc*, executes it, increments the *pc* by 1 and possibly modifies the *stk*.

The syntax and operational semantics are shown in Figure 4.3 using the conventions described earlier in Section 4.1. If none of the transition rules match (which

happens when $pc = n$), then STK and BSTK stutters (this is not shown as a transition rule in Figure 4.3).

What is an appropriate notion of refinement to analyse the correctness of BSTK? Observe that if the buffer is full or the instruction fetched is `top`, BSTK executes all buffered instructions in `ibuf` in a single step. This neither corresponds to a stuttering step nor can be matched by a single step of STK. Therefore, stuttering refinement cannot be directly used to prove the correctness of BSTK. On the other hand, skipping refinement directly accounts for such differences in behaviors of STK and BSTK and therefore is appropriate to prove the correctness of BSTK. We remark that reconciling refinement can also be used to prove the correctness of BSTK. However, skipping refinement is preferable since it is a stronger notion than reconciling refinement. Alternatively, one could construct a *new* specification whose transition relation is the transitive closure of STK, *i.e.*, we adapt the high-level specification to *align* with optimizations introduced in the implementation. However, this is highly undesirable since a specification is a part of the trusted computing base, and should be as simple as possible. Moreover, reasoning about transitive closure of a transition relation often requires finding a closed form characterization of the set of reachable states and increases the challenges during mechanical reasoning.

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 the transition systems of STK and BSTK machines respectively. The set of states S_A and S_C and the transition relation $\xrightarrow{A}$ and $\xrightarrow{C}$ are as described in Figure 4.3, and labeling function L_A and L_C are just the identity functions. We show that $\mathcal{M}_C$ is a skipping refinement of $\mathcal{M}_A$ under the refinement map $R : S_C \rightarrow S_A$, where R is defined as follows:

$$R(\langle imem, pc, ibuf, stk \rangle) = \langle imem, abs(pc - n), stk \rangle$$

where n is the number of instructions in $ibuf$ and abs is a function that takes as

$stk := \varnothing \;|\; \langle v \rangle :: stk$ (Stack)

$inst := \langle push\ v \rangle \mid \langle pop \rangle \mid \langle top \rangle \mid \langle nop \rangle$ (Instruction)

$imem := [inst_1, \ldots, inst_n]$ (Program)

$pc := 0 \mid 1 \mid \cdots \mid n$ (Program Counter)

$ibuf := \varnothing \;|\; ibuf :: \langle inst \rangle$ (Instruction Buffer)

$S_A := \langle imem, pc, stk \rangle$ (STK State)

$S_C := \langle imem, pc, ibuf, stk \rangle$ (BSTK State)

STK ($\xrightarrow{A}$) where $s =$ capacity of stk, $t = |stk|$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle push\ v \rangle, t < s}{\langle imem, pc + 1, v :: stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle push\ v \rangle, t = s}{\langle imem, pc + 1, stk \rangle}$$

$$\frac{\langle imem, pc, [] \rangle : imem[pc] = \langle pop \rangle}{\langle imem, pc + 1, [] \rangle}$$

$$\frac{\langle imem, pc, v :: stk \rangle : imem[pc] = \langle pop \rangle}{\langle imem, pc + 1, stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle top \rangle}{\langle imem, pc + 1, stk \rangle}$$

$$\frac{\langle imem, pc, stk \rangle : imem[pc] = \langle nop \rangle}{\langle imem, pc + 1, stk \rangle}$$

BSTK ($\xrightarrow{C}$) where $k =$ capacity of $ibuf$, $m = |ibuf|$

$$\frac{\langle imem, pc, ibuf, stk \rangle : m < k,\ imem[pc] \neq top}{\langle imem, pc + 1, ibuf :: imem[pc], stk \rangle}$$

$$\frac{\langle imem, pc, ibuf, stk \rangle : imem[pc] = top, \quad \langle ibuf, 0, stk \rangle \xrightarrow{A}{}^{m} \langle ibuf, m, stk' \rangle}{\langle imem, pc + 1, [], stk' \rangle}$$

$$\frac{\langle imem, pc, ibuf, stk \rangle : m = k, \quad \langle ibuf, 0, stk \rangle \xrightarrow{A}{}^{m} \langle ibuf, m, stk' \rangle}{\langle imem, pc + 1, [imem[pc]], stk' \rangle}$$

Figure 4.3: Syntax and Semantics of Stack and Buffered Stack Machine

input an integer and returns its absolute value.

From Definition 12, we must show as witness a binary relation that satisfies the two conditions: (1) it agrees with the refinement map R and (2) it is a skipping simulation on the disjoint union of $\mathcal{M}_C$ and $\mathcal{M}_A$.

Let $B = \{\langle s, R.s \rangle \mid s \in S_C\}$. It is easy to see that the first condition above holds for B. Next we have to choose a proof-method to effectively show that B is an SKS relation. First observe that one step of BSTK corresponds to the largest number of STK steps when the instruction buffer is full or the current instruction is *top*. In this case, the BSTK machine executes all instructions in the instruction buffer and if the current instruction is *top* it executes it as well. Hence, the largest number of steps that BSTK can skip is bounded by $k + 2$, where k is the capacity of the instruction buffer. Thus the conditions RWFSK2b (Definition 13) and WFSK2d (Definition 16), that in general require reachability analysis, can be reduced to bounded reachability. It is not enough to know an upper bound on skipping; the choice of proof method depends on its value. If the upper bound is small, unrolling the transition relation is a viable option and therefore it is feasible to use either RWFSK or WFSK to show that B is an SKS relation. However, with increasing value of the upper bound, unrolling quickly becomes impractical for mechanical reasoning. In such cases it is preferable to use RLWFSK and LWFSK proof methods since they only require local reasoning, *i.e.*, reasoning about states and their successors and therefore amenable for mechanical reasoning. In this case study, we illustrate the case when the upper bound is small and choose RWFSK to show that BSTK is a skipping refinement of STK.

We use our knowledge of the systems to simplify our proof obligations. First, observe that STK and BSTK are deterministic machines, therefore for RWFSK2a we only need to define *rankt*, from the set of states to non-negative integers. Next, as

discussed above, skipping is bounded by $k + 2$, therefore we reduce the reachability analysis in RWFSK2b to bounded reachability analysis. Thus, our proof obligations are

For all $s \in S_C$ and $s \xrightarrow{C} u$:

$$\begin{aligned} &\textit{(RWFSK2b)}\ \langle R.s = R.u \wedge \textit{rankt}(u) < \textit{rankt}(s)\rangle \vee \\ &\textit{(RWFSK2d)}\ \langle R.s \xrightarrow{A}^{<(k+2)} R.u\rangle \end{aligned} \tag{4.2}$$

Note that since BSTK is a deterministic machine, u is a function of s, so we can remove u from the above formula, if we wish. Also k is a (small) constant and we can expand $\xrightarrow{A}^{k+2}$ using only $\xrightarrow{A}$.

We now experimentally evaluate how effective is the analysis with RWFSK is when using existing verification tools. Our goals are to evaluate the specification costs and determine the impact that the use of skipping refinement and the choice of proof method has on verification tools in terms of capacity and verification times. Towards this, we analyse the correctness of BSTK using ACL2s and state-of-the-art model-checkers. For the later, we analyse finite-state models only.

We formalized the operational semantics of BSTK and STK in ACL2s using standard methods. The sizes of the *imem* and *stk* are unbounded. Once the definitions were in place, proving skipping refinement with ACL2s was straightforward. We also evaluated the amenability of RWFSK when using only symbolic simulation and no additional lemmas in ACL2s[1]. We model BSTK with instruction buffer capacity of 2, 3, and 4. Note that the instruction memory (`imem`) and the stack (`stk`) component of the state for BSTK and STK machines are still unbounded. The experiments were run on a 2.2 GHz Intel Core i7 with 16 GB of memory. For the BSTK with

[1]By "additional lemma" we mean lemmas that are specific to the systems under consideration. We still use several theorems in ACL2s, *e.g.*, theorems introduced by defdata about the data structures used to model the systems.

instruction buffer capacity of 2 instructions, it took $\sim$ 12 minutes to complete the proof and for a BSTK with instruction buffer capacity of 3 instructions, it took $\sim$ 2 hours. For BSTK with instruction buffer capacity of 4 instructions the proof did not finish in over 3 hours.

stk depth	stk word	ibuf	imem	inputs	latches(io)	gates(io)	gates(sks)
8	8	8	16	227	552	24460	120598
8	8	16	64	707	1597	51019	307185
8	8	32	128	1347	3040	114380	859186
8	8	32	256	2627	5602	134112	897243
8	8	32	512	5186	10724	173557	935298
16	16	16	32	836	1981	255683	863083
16	16	16	64	1412	3135	264472	990759
16	16	32	128	2564	5730	539129	2080698
16	16	32	256	4868	10340	574221	2118753
16	16	32	512	9476	19558	644385	2152063

Table 4.1: Configurations of BSTK and STK machines and size of AIGs for input-output equivalence(io) and skipping refinement(sks).

Next, we evaluate the amenability our approach when using off-the-shelf model-checkers to analyse the correctness of finite state models of BSTK. Towards this we create a benchmark suite parameterized by the size of *imem*, *ibuf*, and *stk*. We compare the running times of model-checkers based on two notions of correctness: skipping refinement and input-output equivalence. We choose input-output equivalence since that is the most straightforward notion of correctness: if the specification and the implementation systems state in equivalent initial states and are given the same input then if both systems terminate, the final states of the two systems are equivalent. Notice that skipping refinement is a stronger notion of correctness than input-output equivalence, *e.g.*, the later holds even if the concrete system diverges and the abstract system does not diverge. In contrast, skipping refinement distinguishes systems that differ in their divergence behavior. Moreover, earlier we had argued why existing notions of refinement cannot be used to analyse the correctness

of BSTK.

We describe the transition systems of STK and BSTK, and encode the correctness conditions based on input-output equivalence and RWFSK using the BAT specification language and tool [46]. We extend the BAT back-end to compile these problems to sequential AIG's and use finite-state model-checkers to perform the verification. Table 4.1 shows the size of the AIG's generated in terms of number of state elements and gates. In Figure 4.4, we plot the running times (in seconds) for four model-checkers: SUPER_PROVE (SP), TIP, IIMC and BLIMC [2] on a machine with an Intel Xeon X5677 with 16 cores running at 3.4GHz and 96GB main memory. We chose SUPER_PROVE and IIMC as they are the top 2 model-checkers in the single safety property track [2]. TIP and BLIMC are chosen to cover temporal decomposition and bounded model-checking based tools. The timeout limit for model-checker runs is set to 900 seconds. While SUPER_PROVE and IIMC are multi-threaded, BLIMC and TIP use only single core of the machine. The x-axis represents the running time of model-checkers using input-output equivalence and y-axis represents the running time using skipping refinement. A point with x = 900 s indicates that the model-checker timed out for input-output equivalence while y = 900 s indicates that the model-checker timed out for skipping refinement. Results show that model-checkers timeout for most of the configurations when using input-output equivalence while all model-checkers except TIP can solve all the configurations using skipping refinement. Notice that the number of latches for the skipping refinement tables are 0. We believe, based on our experience with these problems, that if we encoded the problem as a sequential AIG, then performance would tend to improve. Given that performance was already so much better than the performance on the input-output equivalence problems, we did not try to further simplify the skipping proof (but notice that we did use sequential AIGs for the input-output equivalence proofs because

the combinatorial AIGs would have been too large due to unrolling).

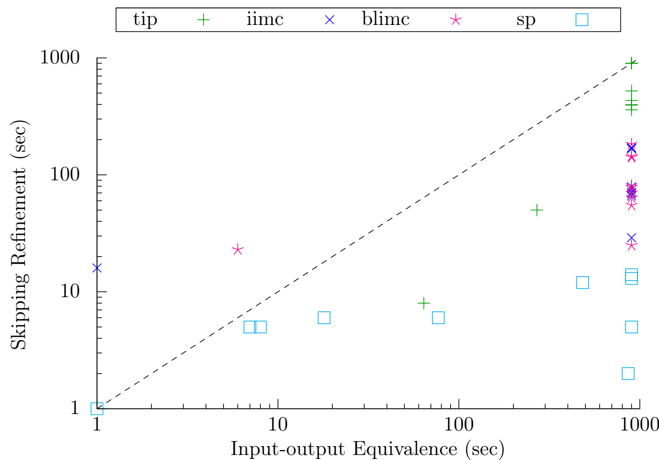

Figure 4.4: Running time (log scale) of model-checkers for stack machine

4.3 Memory Controller

Modern microprocessors operate at a higher clock frequency than their main memories. Thus 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. In this case study, we model such a memory controller, OptMEMC. OptMEMC fetches a memory request from location *pt* in a queue of CPU requests, *reqs*. It enqueues the fetched request in the request buffer, *rbuf* 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 (the capacity of *rbuf* is k, a fixed positive integer), then before enqueuing the request into *rbuf*, OptMEMC first analyses the request buffer for consecutive write requests to the same address in the memory (*mem*). 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 in a single step. Requests in the buffer are executed in the order they were enqueued.

To reason about the correctness of OptMEMC using refinement, we define a simple, high-level specification system, MEMC, that serves as a specification. It fetches memory requests from the CPU one at a time and immediately executes the *read* or *write* request. The syntax and the semantics of MEMC and OPTMEMC are given in Figure 4.5, using the conventions described in Section 4.1.

Like BSTK in Section 4.2, existing notions of refinement cannot be used to directly use to prove the correctness of OptMEMC. This is because when the fetched request is *read* or the request buffer is full, OptMEMC executes all requests in the request buffer, excluding the redundant ones, in a single step. This neither corresponds to a stuttering step not a single step of MEMC. Therefore, stuttering refinement cannot be directly used to prove the correctness of BSTK. On the other hand, skipping refinement directly accounts behaviors introduced when an optimized implementation system runs faster than the specification system.

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 for MEMC and OptMEMC respectively. The set of states S_A and S_C and the transition relation $\xrightarrow{A}$ and $\xrightarrow{C}$ are as described in Figure 4.5, and labeling function L_A and L_C are just the identity functions. We show that $\mathcal{M}_C$ is a skipping refinement of $\mathcal{M}_A$ under the refinement map $R : S_C \to S_A$, where

$$R(\langle\langle reqs, pt, rbuf, mem\rangle\rangle) = \langle\langle reqs, abs(pt - n), mem\rangle\rangle$$

where n is the number of requests in *rbuf* and *abs* is a function that takes as input an integer and returns its absolute value.

Like in Section 4.2, to show that $\mathcal{M}_C$ is a skipping refinement map of $\mathcal{M}_S$, we define a binary relation and show that it satisfies the conditions in Definition 12

$$mem := [v_1, \ldots,] \qquad \text{(Memory)}$$

$$req := \langle write\ \ addr\ \ v\rangle \mid \langle read\ \ addr\rangle$$
$$\mid \langle refresh\rangle,\ addr < n \qquad \text{(Request)}$$

$$reqs := [req_1, \ldots] \qquad \text{(Requests)}$$

$$rbuf := [req_1, \ldots] \qquad \text{(Request Buffer)}$$

$$S_A := \langle reqs, pt, mem\rangle \qquad \text{(MEMC State)}$$

$$S_C := \langle\langle reqs, pt, rbuf, mem\rangle\rangle \qquad \text{(OptMEMC State)}$$

MEMC ($\xrightarrow{A}$)

$$\frac{\langle reqs, pt, mem\rangle :\ \ reqs[pt] = \langle write\ \ addr\ \ v\rangle}{\langle reqs, pt+1, mem[addr] \leftarrow v\rangle}$$

$$\frac{\langle reqs, pt, mem\rangle :\ \ reqs[pt] = \langle read\ \ addr\rangle}{\langle reqs, pt+1, mem\rangle}$$

$$\frac{\langle reqs, pt, mem\rangle :\ \ reqs[pt] = \langle refresh\rangle}{\langle reqs, pt+1, mem\rangle}$$

OptMEMC ($\xrightarrow{C}$)
Let $|rbuf| = j$

$$\frac{\langle reqs, pt, rbuf, mem\rangle :\quad j < k, req \neq top}{\langle reqs, pt, rbuf :: reqs[pt], mem\rangle}$$

$$\frac{\langle reqs, pt, rbuf, mem\rangle,\quad reqs[pt] = \langle read\ \ addr\rangle : \quad \langle rbuf, 0, mem\rangle \xrightarrow{A}{}^{j} \langle rbuf, j, mem'\rangle}{\langle reqs, pt, nil, mem'\rangle}$$

$$\frac{\langle reqs, pt, rbuf, mem\rangle :\quad j = k, \quad \langle rbuf, 0, mem\rangle \xrightarrow{A}{}^{j} \langle rbuf, k, mem'\rangle}{\langle reqs, pt, [reqs[pt]], mem'\rangle}$$

Figure 4.5: Syntax and Semantics of MEMC and OptMEMC

using RWFSK 13. In fact, since MEMC and OptMEMC are deterministic machines and MEMC does not stutter, it is easy to see that we can simplify our proof obligation to Equation 4.2. We describe the transition systems of MEMC and OptMEMC and the correctness conditions using BAT and compile the problem to a sequential AIG. Table 4.2 shows the size of the AIG's generated in terms of the number of state elements and gates. The running times for model checkers are shown in Figure 4.6. While model-checkers timeout for many configurations (points with x = 900s in Figure 4.6) when using input-output equivalence, all except TIP solve all the configurations when using skipping refinement. Furthermore, the running times with skipping refinement show improvement of several orders of magnitude. Also, the performance of SUPER_PROVE when using skipping refinement is much more robust with respect to the size of the system than when using input-output equivalence.

msize	mword	rbuf	reqs	input	latches(io)	gates(io)	gates(sks)
8	8	8	32	480	1082	26429	119075
8	8	8	64	896	1916	32817	117454
8	8	16	32	480	1187	48610	293098
8	8	16	64	896	2021	54998	302956
16	16	32	64	1664	4054	534746	2057352
16	16	32	128	3072	6872	556142	2097336
16	16	32	256	5888	12506	598914	2135409
16	16	32	512	11520	23772	684438	2173481

Table 4.2: Configurations of OPTMEMC and MEMC machines and size of AIGs for input-output equivalence(io) and skipping refinement(sks).

4.4 Event Processing System

Asynchronous event-based programming is a widely accepted methodology to design responsive and efficient user interfaces (UI) [37, 9]. In this methodology, a programmer designates the main UI thread that only performs short-running tasks

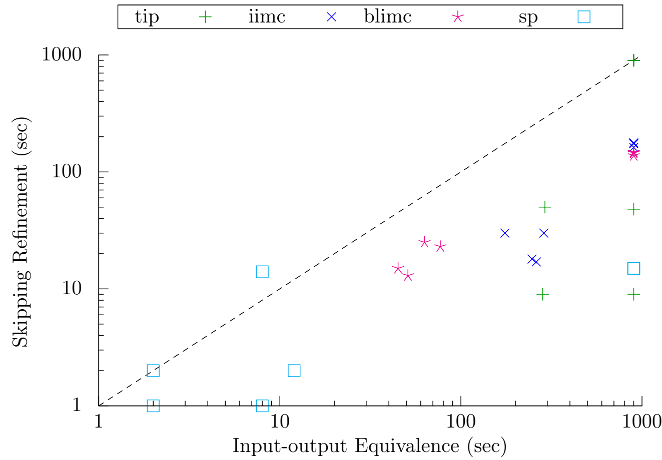

Figure 4.6: Running time (log scale) of model-checkers for memory controller

in response to a user event, while more computationally demanding tasks are performed *asynchronously*. The programming methodology is asynchronous in the sense the tasks may be scheduled for execution at some time in the future. This is in contrast with a *synchronous* programming methodology where the task corresponding to a user event must be executed immediately and the main UI thread block until the task is completed. In the asynchronous programming methodology, the time at which a task is scheduled to execute may depend on its priority and the computational resources available on the execution platform.

In this case study we study correctness of PEPS, a simple asynchronous event-based processing system. PEPS uses a priority queue to find the next event to execute and events interact with each other using a shared memory. The execution of an event may result in adding to the scheduler a finite number of new events scheduled for later execution as well as removing existing events from the scheduler. We make no assumptions on the type of events and place no limits on the number of events in the scheduler.

We first briefly describe the ACL2s constructs used in the formalization of the

transition systems and the correctness condition. For a more detailed exposition of the constructs we refer the reader to the tool documentation [27, 13]. We use the Defdata framework [12] to specify the data definition. It maintains both a predicative characterization and an enumerative characterization of the data type. The later is particularly useful in counterexample generation. The framework also generates a useful theory for reasoning about the data definitions and increases the amount of automation in ACL2s. For example, it generates type predicate, construction function, and accessor functions for the record data type and a suitable theory to reason about it. We use `defunc` to introduce a function definition along with its input-contract (pre-condition) and output-contract (post-condition). When a function definition is admitted in ACL2s using `defunc`, it is guaranteed that it terminates, that it satisfies its input-output contract, and that for every function call, say f, inside the body of the function being defined, all the arguments of f satisfy the input contracts of f. `defunbc` is a special case of `defunc` and is used to describe functions that return boolean values. In our experience, the later was particularly useful in automatically discharging type like proof obligations. ACL2 supports first-order quantifiers, `exists` and `forall`, by way of the `defun-sk` construct. We use `defun-sk` with `exists` construct to model the transition relation of AEPS. We use `encapsulate` to introduce a new function symbol, its signature and a (consistent) set of axioms about it.

We begin by defining the operational semantics of the AEPS, a simple abstract event-processing system that serves as a specification for analysing the correctness of PEPS (Section 2.1). An AEPS state is a three-tuple $\langle \texttt{tm}, \texttt{tevs}, \texttt{mem} \rangle$, where `tm` is a natural number denoting current time, `tevs` is a set of timed-event pairs denoting the scheduler, and `mem` is a collection of variable-integer pairs denoting the shared memory. We require that all events in `tevs` are scheduled to execute at a time $\geq tm$. An

```
(defdata time nat)

;; An event is a string (any type with a total ordering suffices)
(defdata event string)

;; A timed-event is a record consisting of a time and an event.
(defdata timed-event
  (record (tm . time)
          (ev . event)))

;; A set of timed-events
(defdata lo-te (listof timed-event))

;; A memory is a collection of a global variable (var) and integer pair
(defdata memory (alistof var integer))

(defdata aeps-state
  (record (tm . time)
          (tevs . lo-te)
          (mem . memory)))
```

AEPS state, `aeps-state`, is described using the Defdata framework [12]. The framework generates type predicates (`aeps-statep`), constructor function (`aeps-state`), and accessor functions for state components (`aeps-state-tm`, `aeps-state-tevs`, and `aeps-state-mem`).

We say two AEPS-states `w` and `v` are equivalent, `aeps-state-equal`, *iff* if they are equal or their current time components are equal, and their scheduler components and their memory components are set equivalent.

The transition relation of AEPS, `aeps-transp`, is defined as follows: if no event is scheduled to be executed at the current time `tm`, then `tm` is incremented by 1. Otherwise, AEPS nondeterministically picks an event in the scheduler that is scheduled to execute at `tm` and executes it. The execution of an event may remove existing timed-events (`remove-tevs`) in the scheduler and add a finite number of new timed-events (`append`) in the scheduler. We require that any new event added to scheduler

```
(defunbc aeps-state-equal (s w)
  :input-contract t
  (or (equal s w)
      (and (aeps-statep s)
           (aeps-statep w)
           (equal (aeps-state-tm s)
                  (aeps-state-tm w))
           (set-equiv (aeps-state-mem s)
                      (aeps-state-mem w))
           (set-equiv (aeps-state-tevs s)
           (aeps-state-tevs w)))))

;; aeps-state-equal is an equivalence relation
(defequiv aeps-state-equal)
```

must be scheduled to execute at time > `tm`. Execution of an event may also update the memory `mem`. Finally, execution involves removing the picked event from `otevs`.

We use `defun-sk` with `exists` quantifier to model the nondeterminism in AEPS. The function `events-at-tm` returns the set of all events in the scheduler that are scheduled to execution at time `tm`. The function `remove-tev` removes all occurrences of a timed-event `tev` from the scheduler. `remove-tevs` removes all occurrences of timed-events in a list of timed-events from the scheduler. `step-events-add` is an uninterpreted function constrained to return the set of timed-events to add as a result of an event execution. We use `encapsulate` to introduce the function signature and then constrain it to return a set of timed-events that are scheduled for later execution. Moreover, we require that when given set-equivalent memories `mem` and `mem-equiv`, `step-events-add` returns the same set of timed-events for an event at a given time.

Similarly, we introduce uninterpreted functions `step-events-rm` that return the set of timed-events to remove from the scheduler and `step-memory` that returns the updated memory.

We next describe the transition system of PEPS. A PEPS state, `peps-state`, is

```
(defunbc aeps-transp (w v)
  "transition relation of AEPS"
  :input-contract (and (aeps-statep w) (aeps-statep v))
  (let ((w-tm (aeps-state-tm w))
        (w-tevs (aeps-state-tevs w))
        (w-mem (aeps-state-mem w)))
    (if (not (events-at-tm w-tevs w-tm))
        (aeps-state-equal v (aeps-state (1+ w-tm) w-tevs w-mem))
      (aeps-ev-transp w v))))

(defun-sk aeps-ev-transp (w v)
  (exists tev
    (let ((tm (aeps-state-tm w))
          (tevs (aeps-state-tevs w))
          (mem (aeps-state-mem w)))
      (and (timed-eventp tev)
           (equal (timed-event-tm tev) tm)
           (member-equal tev tevs)
           (let* ((ev (timed-event-ev tev))
                  (add-tevs (step-events-add ev tm mem))
                  (rm-tevs (step-events-rm ev tm mem))
                  (new-mem (step-memory ev tm mem))
                  (new-tevs (remove-tevs rm-tevs (remove-tev tev tevs)))
                  (new-tevs (append new-tevs add-tevs)))
             (aeps-state-equal v (aeps-state tm new-tevs new-mem))))))))
```

a three-tuple $\langle \texttt{tm}, \texttt{otevs}, \texttt{mem} \rangle$ where `tm` is a natural number denoting current time, `otevs` is a set of timed-event pairs denoting the scheduler that is ordered with respect to a total order `te-<` on timed-event pairs, and `mem` is a collection of variable-integer pairs denoting the shared memory. We remark that the particular choice of the total order on timed-event pairs is irrelevant to the proof of correctness of PEPS.

The transition function of PEPS is defined as follows: if there are no events in `otevs`, then PEPS just increments the current time by 1. Else it picks the first timed-event pair, say $\langle e, t \rangle$ in `otevs`, executes it, and updates the time to t. The execution of an event may result in adding new timed-events to the scheduler (`step-events-add`), removing existing timed-events from the scheduler (`step-events-rm`), and update

```
(encapsulate
 (((step-events-add * * *) => *)
  ((step-events-rm * * *) => *)
  ((step-memory  * * *) => *))

; constraints on step-events-add
 (defthm step-events-add-contract
   (implies (and (eventp ev)
                 (timep tm)
                 (memoryp mem))
            (and (lo-tep (step-events-add ev tm mem))
                 (valid-lo-tevs (step-events-add ev tm mem)
                                (1+ tm)))))

 (defthmd step-events-add-congruence
   (implies (and (memoryp mem)
                 (memoryp mem-equiv)
                 (eventp ev)
                 (timep tm)
                 (set-equiv mem mem-equiv))
            (set-equiv (step-events-add ev tm mem)
                       (step-events-add ev tm mem-equiv))))
 ...
 )
```

to the memory (`step-memory`). Finally, the executed timed-event is removed from the scheduler. Labeling function is just the identity function.

We show that PEPS refines AEPS using a stepwise refinement approach: first we define an intermediate system HPEPS obtained by augmenting PEPS with history information and show that PEPS is a skipping refinement of HPEPS. Second, we show that HPEPS is a skipping refinement of AEPS. Finally, we appeal to Theorem 11 to infer that PEPS refines AEPS. Alternatively, we could have directly proved that PEPS is a skipping refinement of AEPS. This is possible because the proof-methods (Section 2.4) to reason about SKS are complete, *i.e.*, if PEPS is a skipping refinement of AEPS under a refinement map then we can always use any

```
(defdata peps-state
  (record (tm . time)
          (otevs . o-lo-te)
          (mem . memory)))

(defunbc o-lo-tep (l)
  "An OptAEPS schedule recognizer"
  :input-contract t
  (and (lo-tep l)
       (ordered-lo-tep l)))

(defunbc ordered-lo-tep (l)
  "Check if a list of timed-events, l, is ordered"
  :input-contract (lo-tep l)
  (cond ((endp l) t)
        ((endp (cdr l)) t)
        (t (and (te-< (car l) (second l))
                (ordered-lo-tep (cdr l))))))


(defunbc peps-state-equal (s w)
  :input-contract t
  (or (equal s w)
      (and (peps-statep s)
           (peps-statep w)
           (equal (peps-state-tm s)
                  (peps-state-tm w))
           (equal (peps-state-otevs s)
                  (peps-state-otevs w))
           (set-equiv (peps-state-mem s)
                      (peps-state-mem w)))))

(defequiv peps-state-equal)
```

of the four proof-methods to locally reason about it. However, we will see later that the history information in HPEPS is helpful in defining the constructs required to prove skipping refinement and enables us to separate some of the concerns in the proof. Moreover, we will see that it is relatively easy to prove that PEPS refines HPEPS.

```
(defunc peps-transf (s)
  "transition function for PEPS"
  :input-contract (peps-statep s)
  :output-contract (peps-statep (peps-transf s))
  (let ((tm (peps-state-tm s))
        (otevs (peps-state-otevs s))
        (mem (peps-state-mem s)))
    (if (endp otevs)
        (peps-state (1+ tm) otevs mem)
      (let* ((tev (car otevs))
             (ev (timed-event-ev tev))
             (et (timed-event-tm tev))
             (add-tevs (step-events-add ev et mem))
             (rm-tevs (step-events-rm ev et mem))
             (new-mem (step-memory ev et mem))
             (new-otevs (cdr otevs))
             (new-otevs (remove-tevs rm-tevs new-otevs))
             (new-otevs (insert-otevs add-tevs new-otevs))
             (new-tm et))
        (peps-state new-tm new-otevs new-mem)))))
```

An HPEPS state, `hpeps-state`, is a four-tuple $\langle tm, otevs, mem, h\rangle$, where `tm`, `otevs`, `mem` are respectively the current time, an ordered set of timed events and a collection of variable-integer pairs, and `h` is the history information. The history information `h` consists of a boolean variable `valid`, time `tm`, and an ordered set of timed-event pairs `otevs` and the memory `mem`. Informally, `h` records the state preceding the current state. Finally, labeling function is just the identity function.

```
(defdata hpeps-state
  (record (tm . time)
          (otevs . o-lo-te)
          (mem . memory)
          (h . history)))
(defdata history
  (record (valid . boolean)
          (tm . time)
          (otevs . o-lo-te)
          (mem . memory)))
```

The transition function HPEPS is same as the transition function `peps-transf` of PEPS except that HPEPS also records the history in `h`.

```
(defunc hpeps-transf (s)
  "transition function for HPEPS"
  :input-contract (hpeps-statep s)
  :output-contract (hpeps-statep (hpeps-transf s))
  (let* ((tm (hpeps-state-tm s))
         (otevs (hpeps-state-otevs s))
         (mem (hpeps-state-mem s))
         (hist (history t tm otevs mem)))
    (if (endp otevs)
        (hpeps-state (1+ tm) otevs mem hist)
      (let* ((tev (car otevs))
             (ev (timed-event-ev tev))
             (et (timed-event-tm tev))
             (add-tevs (step-events-add ev et mem))
             (rm-tevs (step-events-rm ev et mem))
             (new-mem (step-memory ev et mem))
             (new-otevs (cdr otevs))
             (new-otevs (remove-tevs rm-tevs new-otevs))
             (new-otevs (insert-otevs add-tevs new-otevs))
             (new-tm (timed-event-tm tev)))
        (hpeps-state new-tm new-otevs new-mem hist)))))
```

We show that PEPS is a skipping refinement of HPEPS under a refinement map `P`, a function that maps a PEPS state to an HPEPS state. Intuitively, this implies that augmenting PEPS with the history information does not change its observable behavior. From Definition 12, we show as witness a binary relation A that satisfies two conditions: first it agrees with the the refinement map P and second that A is an SKS on the disjoint union of PEPS and HPEPS. Let $A = \{\langle s, P.s \mid s \textit{ is a PEPS state}\rangle\}$. It is straightforward to show that the A satisfies the first condition. Next, we use RWFSK to show that A is an SKS relation. We choose RWFSK over the other three proof-methods (Section 2.4) based on the observation that PEPS does not skip with respect to HPEPS. Hence, we avoid defining additional

constructs that account for skipping in the other three proof methods. We can further simplify RWFSK2 based on following two observations: (1) PEPS does not stutter; therefore we can ignore RWFSK2a, and (2) HPEPS is a deterministic system; therefore we can drop the existential quantifier in RWFSK2b.

```
(defunc P (s)
  "A refinement map from an PEPS state to a HPEPS state"
  :input-contract (peps-statep s)
  :output-contract (hpeps-statep (P s))
  (let ((tm (peps-state-tm s))
        (otevs (peps-state-otevs s))
        (mem (peps-state-mem s)))
    (hpeps-state tm otevs mem (history nil 0 nil nil))))

(defthm A-is-a-simulation
  (implies (A s w)
           (A (peps-transf s)
              (hpeps-transf w))))
```

Next we show that HPEPS is a skipping refinement of AEPS under the refinement map R, a function that simply projects an HPEPS state to an AEPS state.

```
(defunc R (s)
  "A refinement map from an HPEPS state to a PEPS state"
  :input-contract (hpeps-statep s)
  :output-contract (aeps-statep (R s))
  (aeps-state (hpeps-state-tm s) (hpeps-state-otevs s) (hpeps-state-mem s)))
```

To show that HPEPS is a skipping refinement of AEPS under the refinement map R, from Definition 12, we must show as witness a binary relation B that satisfies two conditions: first, it agrees with the the refinement map R and second, B is an SKS on the disjoint union of HPEPS and AEPS. Let $B = \{(s, R.s) : s \textit{ is an HPEPS state}\}$. It is straightforward to show that B satisfies the first condition. To show that B is an SKS on the disjoint union of HPEPS and AEPS, we have a choice of four proof-methods (Section 2.4). Which is an appropriate one to analyse correctness of

HPEPS? Towards this observe that when no events are scheduled to execute at the current time `tm`, HPEPS skips over states of AEPS: while AEPS increments `tm` by 1, HPEPS updates `tm` to the earliest time in future when an event is scheduled for execution. Moreover, we cannot a priori determine an upper bound on skipping. For example, the execution of an event may add a new event that is scheduled to execute at an arbitrary time in the future. Without an upper bound on skipping, conditions RWFSK2b (Definition 13) and WFSK2d (Definition 16) requires reachability analysis which is not amenable for automated reasoning. In contrast, RLWFSK and LWFSK require only local reasoning. Finally, AEPS does not stutter; hence the distinction between stuttering and skipping provided by LWFSK is not helpful. Therefore, we choose RLWFSK to show that B is an SKS relation on the disjoint union of PEPS and AEPS.

RLWFSK1 holds trivially. To prove that RLWFSK2 holds we define a binary relation $\mathcal{O}$ and a rank function *rankls* and show that they satisfy the two universally quantified formulas in RLWFSK2. Moreover, since HPEPS does not stutter we ignore RLWFSK2a, and that is why we do not define *rankt*. Finally, our proof obligations are as follows:

Note that the proof obligations involves only states and their successors, *i.e.*, the reasoning is local. This is in contrast to reasoning about SKS using WFSK and RWFSK, which requires reachability analysis due to unbounded skipping exhibited by PEPS. Informally, we define $\mathcal{O}$ as follows: a HPEPS state x, and an AEPS state y are related by $\mathcal{O}$ *iff* one of the following two conditions hold: (1) x and y are related by B, or (2) a predecessor of x, obtained from the history `h` in x, has the current time component less than or equal to the current time component in y, has a non-empty scheduler that is set-equivalent to the scheduler in y, and has a memory that is set-equivalent to the memory in y. The function *rankls* is defined

```
(defthm B-is-an-RLWFSK
  (implies (B s w)
           (rlwfsk2b (hpeps-transf s) w)))

(defun-sk rlwfsk2b (u w)
  (exists v
    (and (aeps-transp w v)
         (𝒪 u v))))

(defthm 𝒪-is-good
  (implies (and (𝒪 x y)
                (not (B x y)))
           (rlwfsk2f x y)))

(defun-sk rlwfsk2f (x y)
  (exists z
    (and (aeps-transp y z)
         (𝒪 x z)
         (< (rankls z x) (rankls y x)))))
```

as follows: given a HPEPS state x, and an AEPS state y, *rankls* is equal to the (absolute) difference between time in x and time in y plus the number of events in the scheduler in y scheduled to execute at `t` in x. The witness states in the above proof obligations were easy to determine based on whether or not there is an event scheduled for execution at the current time.

After the data definitions and function definitions modeling the transition systems for the three systems were in place, additional lemmas were proved to discharge the above proof obligations. They can be roughly categorized into three categories. First, the set of lemmas to prove the input-output contracts of the functions. Second, a set of lemmas to show that operations (`remove-tevs, append, insert-otevs`) on the schedulers in AEPS, PEPS, and HPEPS preserve the invariant that all timed-events are scheduled to execute at a time greater or equal to the current time. Third, a set of lemmas to show that removing two equivalent sets of timed-events from a scheduler results in *equivalent* schedulers. Similarly, a set of lemmas to show

that inserting two equivalent sets of timed-events to a scheduler results in equivalent schedulers. Recall that two AEPS schedulers are equivalent if they are set equivalent and two schedulers in PEPS and HPEPS are equivalent if they are equal. Finally, a set of lemmas were proven to show that two ordered sets are equivalent *iff* they are set equal. These lemmas play an important role in relating a state in AEPS to a state in HPEPS and therefore proving that B is an SKS relation. Observe that the above lemmas establish a relationship between data structures in the implementation system and the specification system. The behavioral difference between the two systems is accounted for by the notion of skipping refinement. This separation significantly eases understanding as well as mechanical reasoning about the correctness of reactive systems. In total, we introduced 9 data definitions and 28 functions to model the three systems. We have 8 top-level proof obligations and 144 supporting lemmas. The entire proof takes about 11 minutes on a machine with 2.2 GHz Intel Core i7 with 16GB main memory. Note that the supporting lemmas are in addition to the set of lemmas automatically generated by the Defdata framework and those included in ACL2s for reasoning about primitive data types. The models and the proof script are publicly available [1].

Finally, we remark that PEPS can be seen as an instance of MPEPS system introduced in 2.3: consider an implementation of the priority queue in MPEPS that deterministically selects *one* event from the set of events scheduled to execute at the current time. Therefore, PEPS is a skipping refinement of AEPS can also be inferred from the proof that MPEPS is a skipping refinement of AEPS.

4.5 Conclusion

In this chapter, we experimentally validated the applicability of skipping refinement and associated proof-methods to analyse the correctness of four optimized reactive

systems. We highlighted several considerations in selecting an appropriate notion of refinement and a proof-method to analyse it. We show that skipping refinement enables us to directly reason about the correctness of such optimized reactive systems and is amenable for reasoning using both deductive and automated verification methodologies. In particular, we showed that without using skipping refinement, proving the correctness of relatively simple configurations is beyond the current model-checking technology. But when using skipping refinement and the associated local proof-methods, existing model-checkers scale to significantly larger configurations.

Chapter 5

Related Work

5.1 Notions of equivalences and refinement

Pnueli [52] first proposed the use of temporal logics to specify and reason about the correctness of reactive systems. This approach has stood the test of time and automated verification methodology based on *model checking* have been very successful in analyzing the correctness of several reactive systems. The correctness of a system is expressed as a set of formulas in a temporal logic and model checking algorithms determine if the system *satisfies* the formulas. An alternative approach based on behavioral equivalences was first studied by Milner. He proposed the notion of *simulation* to compare (possibly) non-terminating and deterministic systems [47]. It was used as the basis for a method to prove the correctness of a concrete data representation with respect to it abstract version [24]. However, concurrent systems cannot be analyzed using this notion. This is because when comparing concurrent systems, we must also account for nondeterminism introduced as a result of interleaving of its intermediate atomic operations. Therefore, a concurrent system, terminating or non-terminating, is often modeled as a reactive system that maintains an ongo-

ing interaction with its environment. With this in mind, Hennesey and Milner [23] proposed the notion of *strong bisimulation* to compare the semantics of concurrent systems. The notion strongly preserves the branching structure, including the non-determinism exhibited by them in their intermediate states. Strong bisimulation is an appropriate notion when all actions of a concurrent system are observable. The definition of strong bisimulation also suggests an effective local proof-method. Two states of a system are *bisimilar* if there exists a bisimulation relation between them. David Park [51] studied bisimilarity using theory of fixpoints. He showed that bisimilarity can be characterized as the greatest fixpoint of a monotone function on the complete lattice of binary relations on states. Note that strong bisimulation implies trace equivalence. Since the former only requires local reasoning, it is often used to prove trace equivalence.

Strong bisimulation is often too strong a notion of correctness when relating systems with unobservable events. *Weak bisimulation*, proposed in [23], abstracts over unobservable events *i.e.*, it identifies systems that differ only in unobservable events. It is strictly weaker than strong bisimulation. Both strong and weak bisimulation have favorable algebraic properties; they are closed under arbitrary union and relational composition and there exists the largest strong (weak) bisimulation relation. However weak bisimulation, unlike strong bisimulation, does not preserve the branching structure in presence of τ-steps. That is, it does not distinguish two systems that may differ in potential nondeterminism of intermediate states reachable from related states through unobservable steps.

To remedy this, *branching bisimulation* was presented in [59]. Like weak bisimulation, it abstracts over unobservable steps of a system but preserves its branching structure. Branching bisimulation is also closed under arbitrary union. However, the relational composition of two branching bisimulations is not a branching bisimula-

tion [7]. Nevertheless, the largest branching bisimulation relation, hence branching bisimilarity, is an equivalence relation [16, 7]. Branching bisimulation can be also characterized as a maximal fixpoint of a monotonic function on the complete lattice of binary relations on states.

When considering unobservable τ-steps, an important factor to determine an appropriate notion of correctness is its sensitivity to divergent behaviors: does the notion differentiate between a system with an infinite sequence of unobservable events from one that does not? Branching bisimulation abstracts away divergent behaviors and identifies a system with divergent behaviors to a system that does not. *Divergence sensitive branching bisimulation* [16] and *branching bisimulation with explicit divergence* [59] are finer than branching bisimulation and distinguish a system with divergent behaviors from one that does not. It is also closed under arbitrary union. In [38, 18], authors also present a fixpoint characterization of branching bisimulation with explicit divergence.

Stuttering bisimulation [14] also abstracts over unobservable steps. It differs from weak and branching bisimulation: it is defined on a state-labeled transition system and only identifies systems that exhibit finite stuttering. Furthermore, stuttering bisimulation is closed under arbitrary union and relational composition. In [16] it was shown that two states that are divergence sensitive branching bisimilar satisfy the same class of CTL*\X properties[1]. This in conjunction with the result that two states that are stuttering equivalent[2] also satisfy the same class of CTL*\X properties [14], indicates a strong relationship between divergence sensitive branching

[1]Notice that semantics of CTL*\X are defined on a state-labeled transition system while divergence sensitive branching bisimilarity is defined on an edge-labeled transition system. Therefore they cannot be compared directly. In [16], authors define a transformation from an edge-labeled transition system to state-labeled transition system and use it show that two states that are divergence sensitive branching bisimilar satisfy the same class of CTL*\X properties

[2]Two states, say s, w are stuttering equivalent if there exists a stuttering bisimulation relation B such that sBw.

bisimulation and stuttering equivalence [16].

The divergence sensitive variants of branching bisimulation require that for any two related states, say s and w, there is an infinite τ-run starting from s iff there is an infinite τ-run starting at w. Such reasoning about infinite τ-runs is often difficult to automate. In [38, 18], authors propose *inductive branching bisimulation* (IBB), an alternative characterization of branching bisimulation with explicit divergence that is expected to be useful for verification. IBB forbids infinite stuttering by demanding that the related states also satisfy an *inductive τ-matching closure property*. However, it still requires reasoning about unbounded τ-runs, and therefore is not amenable to verification using automated reasoning tools. *Well-founded bisimulation* (WFB), an alternative characterization of stuttering bisimulation, is proposed in [49, 42]. The characterization is based on a rank function whose codomain is a well-founded set. In contrast to IBB, WFB requires only local reasoning about states and their successors. This is highly desirable when mechanically reasoning about reactive systems; it significantly reduces proof effort and makes it more amenable to automated reasoning tools. In [42], the author also introduce the notion of stuttering simulation as well as a sound and complete proof method to analyze it. Stuttering simulation is closed under arbitrary union and relational composition. Skipping simulation (Chapter 2) is a weaker notion of correctness than stuttering simulation. Skipping simulation is the first (as far as we know) behavioral notion of correctness that directly supports reasoning about optimized reactive systems that may execute faster than their specifications. Skipping simulation is also closed under arbitrary union and relational composition. The latter property implies that skipping simulation supports modular reasoning using a stepwise refinement approach. We also develop sound and complete proof-methods that require only local reasoning, and therefore more amenable for automated reasoning. Reconciling simulation is a notion

weaker than skipping simulation and further extends the domain of applicability of refinement-based approach to reasoning about the correctness of a larger class of reactive systems. Reconcile simulation also admits sound and complete proof-methods that require only local reasoning about states and their successors. Like stuttering and skipping simulation, reconcile simulation is closed under arbitrary union. However, it is not closed under relational composition.

We finally note that characterization of skipping and reconciling simulation (like stuttering simulation and bisimulation) in terms of matching fullpaths [49, 42], are preferable as statements of correctness. It is simpler to understand and disentangles a statement of correctness from the intricacies associated with mechanical reasoning. This is highly desirable since the notion of correctness is part of the trust base in a verification methodology.

Local reasoning and refinement maps Refinement map, a *function* from a state of the concrete system to a state of the abstract system, is a key ingredient in a refinement-based verification methodology. The refinement map along with the notion of refinement and the abstract system constitute the trust base. A fundamental question about this methodology is the following: given a notion of correctness and a refinement map, if a concrete system *refines* an abstract system, under what conditions can we prove it using only local reasoning, *i.e.*, reasoning about states and their successors? This is an interesting question because it elucidates the domain of applicability of the notion of refinement to effectively analyze correctness of reactive systems. Abadi and Lamport [34] studied this problem in the linear-time framework where a behavior of a system is described as a sequence (possibly infinite) of states and *refines* is defined as the trace containment upto stuttering. They showed that if systems under consideration are from a restricted class (see their paper for details), then one can add appropriate *history* and *prophecy* variables to the concrete system,

define an appropriate projection function from a state of the augmented $\mathcal{C}$ to a state of $\mathcal{A}$ and reduce reasoning about infinite behaviors to reasoning only about single state transitions. Lynch [39] also studied several notions of correctness like forward simulation, backward simulation, and under what restrictions on the types of systems does trace inclusion can be proved using only local reasoning. Manolios [41, 42] studied the problem in the branching-time framework where the behavior of a system is defined as a computation tree. It was shown (Theorem 1) given any refinement map, say r, if a concrete system is a stuttering refinement of an abstract system under r, one can always prove it using only local reasoning. We also developed the theory of skipping refinement and reconciling refinement in the branching-time framework From the soundness and completeness results (Theorem 22 in Chapter 2 and Theorem 36 in Chapter 3), we infer that one can always reason locally about skipping refinement and reconciling refinement. This is analogous to the result for stuttering refinement. It differs from the result of Abadi and Lamport in that our results hold without any restrictions on the types of systems and without the need of augmenting the concrete system with history or prophecy variables. Moreover, the result holds even for systems with branching factor (nondeterminism) of arbitrary cardinality. Also, notice that we place no restriction on the refinement map; it can be an arbitrary function from states of the concrete system to a state of the abstract system. This flexibility in the choice of refinement map has been fruitfully exploited in the past to design efficient refinement-based verification methodology [43]. However, recall that refinement map forms part of the trust base and one must be prudent in its choice; by choosing a complicated refinement map, one can bypass the verification problem. We refer the reader to [41] for a more in-depth and insightful analysis of the differences between notions of correctness based on linear-time framework and branching-time framework.

5.2 Applications

5.2.1 Processor Verification

Several variants of correctness for verification of superscalar processors are proposed in the literature [3]. These variants can be broadly classified on the basis of whether (1) they support deterministic or nondeterministic abstract systems (2) they support deterministic or nondeterministic concrete systems, and (3) the kinds of refinement map allowed by the notion. In contrast, the theory of stuttering refinement [40], skipping refinement and reconciling refinement provides a general framework for both deterministic and nondeterministic systems and any choice of a refinement map; in all cases one proves the same theorem. We believe that a uniform notion of correctness significantly ease the verification effort and increases the trust in the verification methodology.

5.2.2 Software

The Common Criteria [11] are a standard for software verification and is used, among others, by several government agencies to specify software assurance requirements. It consists of seven levels of assurance, and details of what exactly is being certified, and for which application area. The highest Common Criteria evaluation level for a software artifact mandates a formal specification of requirements, its functional specification, and the high-level design. The low-level design and the correspondence between low-level design and the concrete system can be done informally. To completely analyze a software artifact formally, one must establish a formal correspondence between the low-level design and the concrete system in a way that assurances established at the higher level hold for the concrete system as well. Refinement-based approaches have been successful in the past to achieve this goal for several realis-

tic software artifacts [60, 8, 29]. Our work on skipping refinement and reconciling refinement extends the domain of the applicability of refinement-based approach to formally verify a larger class of reactive systems.

Compiler correctness In [44], it was shown how to prove the correctness of assembly programs running on a pipelined machine by decomposing such proofs in two parts. First, the assembly code is proven correct when running on an idealized processor that directly implements the instruction set architecture. Second, the pipelined machine is proven to be a *WEB-refinement* of the abstract machine. However, it was noted in [45] that one cannot prove WEB-refinement for pipelined machines that can retire multiple instructions in a single cycle (*e.g.*, superscalar processors); infinite executions of such a pipelined machine and its ISA will not match.

Modern compilers are highly complex software systems and among other things, an appropriate notion of correctness and the efficiency of associated proof methods play an important role in developing a successful verification methodology for them. There is a large body of work in the area of compiler verification [15, 36, 56]. What is different about verification of a compiler is that it requires us to prove full functional correctness: given a source program, the observable behaviors of the target program generated by the compiler must be a subset of observable behaviors of the source program.

Several back-end compiler transformations are proven correct in Compcert [36]. The overall proof strategy is to establish a *forward* simulation between the source and the target program. Since the semantics of the target program is deterministic, forward simulation implies trace inclusion, *i.e.*, behaviors of target program are a subset of behaviors of the source program.

Translation validation [54] is an alternative approach to verify the correctness of compiler transformations. In this approach, one constructs a validation tool, which

after every run of the compiler formally checks that the target program produced is a correct translation of the source program. However, notice that the assurance provided by the approach is only applicable to the particular source and target program pair. Nevertheless, this approach is especially attractive when formally verification of a full-fledged optimizing compiler is not feasible. The work on translation validation is restricted to programs which generate deterministic transition systems [6].

In [50], authors use an approach, similar to translation validation, to analyze correctness of several compiler transformations. An optimization procedure is augmented with an auxiliary *witness generator*. For every run of the compiler transformation, the generator first constructs a binary relation, called *witness relation*, between states of the source and the target program generated by the transformation. Then, a validation tool checks if the relation meets the conditions imposed by stuttering refinement. The key difference between this approach and general translation validation approach is that the former assumes that the analysis phase of an optimization is visible to the witness generator; hence, it can make use of the auxiliary invariants derived during the analysis in constructing an appropriate witness relation. Moreover, in contrast to simulation refinement that is commonly used in previous work on translation validation, authors use stuttering refinement as a notion of correctness. Stuttering refinement, unlike simulation refinement, is complete for the case when the target program produced by the compiler is shorter than the source program. However, stuttering refinement is only applicable if the instructions replaced in the source program do not change the observable component of state. Consider the control flow graph compression transformation in the paper: the witness relation states that a state t in the target program and a state s in the source program are related if s and t agree on all variables except the program counter and either the program counter at s and t are equal or the program counter

of s lies on the linear chain of *skip* statements starting from the program counter of t. Notice that the eliminated statements are *skip* statement – a statement that does not modify the value of any other variable. Now consider the following source program: $x = x + 3; x = x + 5$ and an optimizing transformation that produces the target program $x = x + 8$. The target program is shorter than the source program but stuttering refinement cannot be used to analyze correctness of this transformation because both statements in the source program change observable component of the state. Skipping refinement is an appropriate notion of correctness for reasoning about such transformations. In [25], we showed that a vectorizing transformation that replaces a sequence of scalar instructions with a SIMD instruction can be analyzed using skipping refinement.

Remark 2. Note that externally visible events in Compcert (and CompcertTSO [55]) are a procedure call and return and OS I/O statements like *printf*. The steps $x = x + 3; x = x + 5$ in the source program and the step $x = x + 8$ in the target program encapsulated between a procedure call and its return, are considered internal (unobservable) actions. Therefore such transformations can still be verified using forward simulation in Compcert. Similarly, in CompcertTSO, an optimization that eliminates a fence instruction induces additional nondeterminism since memory write operations in the store buffer can be unbuffered at any point in the time until the next fence instruction. However, unbuffering a memory write operations from a store buffer is not an observable event. Hence this optimization can also be analyzed with *backward simulation*.

Operating system kernels An operating system is at the core of several computer systems. There are several projects targeting verification of operating systems. We review two such projects, seL4 [30, 29] and Certikos [20, 21], from the viewpoint of the notion of correctness used and the associated proof method, and refer the reader

to [28] for a more comprehensive survey.

seL4 is a high-performance microkernel written in C. The functional correctness of seL4 is established by proving that it *refines* an abstract specification of the microkernel written in Isabelle/HOL, where the notion of refinement is based on *data refinement* [17]. This is achieved in two steps: first, an *intermediate specification* is obtained by (automatically) translating a prototype implementation of microkernel in (a subset of) Haskel to Isabelle/HOL. It is shown that the intermediate specification refines the abstract specification. Second, the C implementation is parsed into Isabelle/HOL and is shown to refine the intermediate specification. The transitivity property of data-refinement implies that the C implementation refines the abstract specification. The proof method used to show data refinement is based on *forward simulation* [17] which requires matching a step in the concrete system to a step in the abstract system. Use of this proof method dictates that the atomicity of a step at all levels of abstraction must be same. This is undesirable; since a lower-level implementation often describes the computation in more detail and hence in comparison to a high-level abstract specification take more steps to perform the same task.

Certikos [20, 21] is a formally verified OS kernel. The verification methodology takes advantage of the layered architecture design of an OS kernel. Let $\mathcal{M}$ be an implementation built over a layer interface L_b, let L_t be the layer interface of $\mathcal{M}$ and P be a client program running on L_t. Then $\mathcal{M}$ is analogous to a program transformer, that transforms a program P running on L_t to $\mathcal{M}(P)$ running on L_b. With this viewpoint, the correctness of an implementation $\mathcal{M}$ is stated as follows: behavior of $\mathcal{M}(P)$ running on top of L_t refines that of P running on top of L_b. Like Compcert, the notion of correctness is based on trace inclusion and the proof method used is based on forward simulation [39]. However, notice that forward simulation

does not preserve liveness properties and preserving such properties was one of the motivation to use *deep specification* over *shallow specification* (using Hoare triples).

Conclusions and Future Work

In this dissertation, we showed that refinement-based methodology can be used to effectively analyze correctness of optimized reactive systems. We introduced two new notions of correctness, skipping simulation and reconciling simulation and developed a theory of refinement based on them. We studied their algebraic properties and developed several sound and complete proof-methods that can be used to effectively reason about them. The new notions of refinement and associated proof-methods significantly extend the domain of applicability of the refinement-based approach to a large class of optimized reactive systems. Our work can be extended across multiple dimensions.

A logical characterization of SKS and RES An alternative approach to specify the correctness of reactive systems is based on temporal and modal logics such as Mu-calculus, LTL, and CTL^* [52, 19]. In this approach, the correctness of a reactive system is specified using a set of temporal logic properties: a reactive system is correct if its computations *satisfy* the set of formulas. It is natural to ask the following question: what properties are preserved by a behavioral notion of correctness. For example, it was shown that stuttering bisimulation preserves $CTL^*\backslash X$ in [10] and stuttering simulation preserves $ACTL^*\backslash X$ [41]. A similar logical characterization of SKS and RES will shed more light on the relationship between the two systems that

are related by a new notion.

A spectrum of notions of correctness Let $\mathcal{M} = \langle S, \rightarrow, L \rangle$ be a transition system, σ and δ be fullpaths in $\mathcal{M}$ and $B \subseteq S \times S$. Notions of matching studied in the dissertation (Definition 4, Definition 10, Definition 19) partition σ and δ in finite non-empty segments but differ in the size of the partitions and the relationship between states in a segment in σ and states in the corresponding segment in δ. The difference can be captured using the following three parameters: (1) number of states in a segment: all partitions in a fullpath (σ or δ) are of length 1 or ≥ 1; (2) labeling of states in a segment: all states in a segment of a fullpath (σ or δ) are labeled identically or not, and (3) what states in the corresponding segments of σ and δ are required to be related by B: $\{$*all*, *first*$\}$ states in a segment in σ are related by B to $\{$*all*, *first*$\}$ states in the corresponding segment in δ. It is easy to see that the notions of matching fullpaths and therefore the notions of correctness (Definition 2, Definition 5, Definition 11, Definition 20) studied in this dissertation can be expressed using these three parameters. In addition, the parameterization suggests several new notions of correctness. For example, consider the notion of correctness that allows skipping in the concrete system but only allows stuttering in the abstract system. It can be defined using the following notion of match: fullpaths σ and δ match under B *iff* the fullpaths can be partitioned in to non-empty finite segments such that the *first state* in a segment in σ is related to all states in the corresponding segment in δ. It would be interesting to present a unified exposition of all the notions of refinement and study the associated proof-methods.

We expect such a classification to be also useful in practice. When using a refinement-based approach to verification, the specification system and the notion of correctness constitute the trust base. Hence, it is essential that the specification is as-simple-as-possible to understand and the notion of correctness as-strong-as-

possible. Therefore, it is useful to have access to a collection of different notions of correctness. Thereby we avoid a situation where one is required to convolute the specification because the weakest notion of refinement available is unduly restrictive and cannot be used to directly reason about the correctness of the implementation. Recall that reconciling simulation is not compositional (Lemma 28). Hence, a natural question is the following: what is the weakest notion in the collection that admits compositional reasoning. A set of notions of correctness, their algebraic properties, associated proof-methods, and a classification such as above will be extremely helpful in practice to systematically choose an appropriate verification methodology.

Refinement-based testing Formal verification techniques provide guarantees about the 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. The current methodology that is prevalent in practice is based on specifying a set of properties, compile them into runtime checks and validate them during simulation. However, it is difficult to determine if the set of properties and tests under consideration is complete. An alternative approach is based on refinement: given an abstract system that serves as a specification for the implementations, compile the refinement conjecture into a runtime check that is validated during simulation. We anticipate that the local proof-methods developed in this thesis for formal analysis shall also result in efficient runtime checkers.

Bibliography

[1] Experimental artifacts http://www.ccs.neu.edu/home/jmitesh/dissertation.

[2] Results of hardware model checking competition, 2013 (http://fmv.jku.at/hwmcc13/hwmcc13.pdf).

[3] M. Aagaard, B. Cook, N. Day, and R. Jones. A framework for microprocessor correctness statements. *Correct Hardware Design and Verification Methods*, 2001.

[4] M. Abadi and L. Lamport. The existence of refinement mappings. In *Theoretical Computer Science*, 1991.

[5] R.-J. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, 1990.

[6] C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. TVOC: A translation validator for optimizing compilers. In *CAV*, 2005.

[7] T. Basten. Branching bisimilarity is an equivalence indeed! *Inf. Process. Lett.*, 1996.

[8] W. R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5:519–530, 1989.

[9] A. Bouajjani, M. Emmi, C. Enea, B. K. Ozkan, and S. Tasiran. Verifying robustness of event-driven asynchronous programs against concurrency. In *ESOP*, 2017.

[10] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. In *Theoretical Computer Science, 1988*, 1988.

[11] CC. Common criteria for information technology security evaluation CC v3.1, 2017.

[12] H. R. Chamarthi, P. C. Dillinger, and P. Manolios. Data definitions in the ACL2 sedan. In *ACL2*, 2014.

[13] H. R. Chamarthi, P. C. Dillinger, P. Manolios, and D. Vroon. The ACL2 sedan theorem proving system. In *TACAS*, 2011.

[14] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC*, 1986.

[15] M. A. Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 2003.

[16] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 1995.

[17] W.-P. de Roever. Data refinement: model-oriented proof methods and their comparison. 2003.

[18] R. W. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[19] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, 1990.

[20] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: a certified kernel for secure cloud computing. In *APSys*, 2011.

[21] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *POPL*, 2015.

[22] D. S. Hardin. Real-time objects on the bare metal: an efficient hardware realization of the java tm virtual machine. In *ISORC, 2001*, 2001.

[23] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *ICALP*, 1980.

[24] C. A. R. Hoare. Proof of correctness of data representation. In *Language Hierarchies and Interfaces*, 1975.

[25] M. Jain and P. Manolios. Skipping refinement. In *CAV*, 2015.

[26] M. Jain and P. Manolios. An efficient runtime validation framework based on the theory of refinement. *CoRR*, abs/1703.05317, 2017.

[27] M. Kaufmann and J. S. Moore. ACL2 homepage. 2006.

[28] G. Klein. Operating system verification-an overview. 2009.

[29] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 2014.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *SOSP*, 2009.

[31] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10:325–342, 1984.

[32] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143, 1977.

[33] L. Lamport. What good is temporal logic. *Information processing*, 1993.

[34] L. Lamport and M. Abadi. The existence of refinement mappings. *Theoretical Computer Science*, 1991.

[35] S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI, 2000*, 2000.

[36] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 2008.

[37] Y. Lin, S. Okur, and D. Dig. Study and refactoring of android asynchronous programming. 2015.

[38] X. Liu, T. Yu, and W. Zhang. Analyzing divergence in bisimulation semantics. In *POPL*, 2017.

[39] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Information and Computation*, 1995.

[40] P. Manolios. Correctness of pipelined machines. In *FMCAD*, 2000.

[41] P. Manolios. *Mechanical verification of reactive systems*. PhD thesis, University of Texas, 2001.

[42] P. Manolios. A compositional theory of refinement for branching time. In *CHARME*, 2003.

[43] P. Manolios and S. K. Srinivasan. A computationally effecient method based on commitment refinement maps for verifying pipelined machines. In *MEMOCODE*, 2005.

[44] P. Manolios and S. K. Srinivasan. A framework for verifying bit-level pipelined machines based on automated deduction and decision procedures. 2006.

[45] P. Manolios and S. K. Srinivasan. Automatic verification of safety and liveness for pipelined machines using web refinement. In *TODAES*, 2008.

[46] P. Manolios, S. K. Srinivasan, and D. Vroon. Bat: The bit-level analysis tool. *CAV*, 2007.

[47] R. Milner. An algebraic definition of simulation between programs. *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, 1971.

[48] J. Misra. Distributed discrete-event simulation. *ACM Computing Survey*, 1986.

[49] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1997.

[50] K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. *Static Analysis Symposium*, 2013.

[51] D. Park. Concurrency and automata on infinite sequences. *Theoretical computer science*, 1981.

[52] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, 1977.

[53] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. *Automata, Languages and Programming*, 1985.

[54] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *TACAS*, 1998.

[55] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. *J. ACM*, 2013.

[56] J.-B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *ACM Sigplan Notices*, number 1, 2010.

[57] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR*, 1990.

[58] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In *IFIP Congress*, 1989.

[59] R. J. Van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 1996.

[60] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the ucla unix security kernel (extended abstract). In *SOSP*, 1979.

[61] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 1971.