Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Specification and Verification of Concurrent Programs through Refinements

Sandip Ray* · **Rob Sumners**

July 30, 2012

Abstract We present a framework for the specification and verification of reactive concurrent programs using general-purpose mechanical theorem proving. We define specifications for concurrent programs by formalizing a notion of refinements analogous to stuttering trace containment. The formalization supports the definition of intuitive specifications of the intended behavior of a program. We present a collection of proof rules that can be effectively orchestrated by a theorem prover to reason about complex programs using refinements. The proof rules systematically reduce the correctness proof for a concurrent program to the definition and proof of an invariant. We include automated support for discharging this invariant proof with a predicate abstraction tool that leverages the existing theorems proven about the components of the concurrent programs. The framework is integrated with the ACL2 theorem prover and we demonstrate its use in the verification of several concurrent programs in ACL2.

Keywords abstraction, fairness, reactive systems, theorem proving, trace containment

1 Introduction

Reactive concurrent programs consist of a number of interacting components (*e.g.*, processes, threads, etc.) that perform ongoing, non-terminating computations while receiving stimulus from an external environment. The complexity induced by the interactions of these components makes the implementations of concurrent programs particularly subtle and error-prone, and bugs are often difficult to detect and diagnose. With pervasive deployment of multicore and multiprocessor systems, robust methodologies for the verification of reactive concurrent programs are essential.

* This work was done when the author was at the University of Texas at Austin.

Sandip Ray
Strategic CAD Labs, Intel Corporation, Hillsboro OR 97124, USA.
E-mail: sandip.ray@intel.com

Rob Sumners
Advanced Micro Devices, Inc., Austin TX 78741, USA.
E-mail: robert.sumners@amd.com

A common approach for reasoning about a reactive system is to prove that its executions can be appropriately viewed as executions of a simpler system; the simpler system serves as the specification. There has been extensive work on defining a notion of correspondence as a mathematical theory that captures the intuitive relationship between an implementation and a specification while facilitating effective reasoning [1, 60, 4, 51, 34, 31, 50]. While the actual definition of correspondence varies, all share two features that Manolios [50] succinctly summarized as follows:

Stuttering Since the specification is defined at a more abstract level than the implementation, notions of correctness should allow for *stuttering*, where the implementation may take several steps before matching a single step of the specification [44].

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

In this paper, we build a formal verification framework for reasoning about reactive concurrent programs with the ACL2 theorem prover [35]. Our framework is composed of three ingredients, (1) a formal notion of correspondence between a program implementation and specification that supports stuttering and refinement, (2) a collection of formalized *reduction rules* orchestrated by the theorem prover to decompose a verification problem into manageable pieces, and (3) an integrated predicate abstraction procedure to automate proofs of invariants. We also formalize notions of fairness to facilitate reasoning about progress properties. The result is a powerful and extendible deductive framework inside the ACL2 theorem prover, that can be applied to reason effectively about a large class of reactive concurrent programs. We report our experiences on the use of the framework in verification of several concurrent programs in ACL2.

The notion of correspondence we formalize is loosely based on *stuttering trace containment* [1]: "For each (infinite) execution of the implementation there is an (infinite) execution of the specification that has the same observable behavior up to finite stuttering." Stuttering trace containment and related notions of stuttering simulation and bisimulation have, of course, been studied extensively in formal verification literature in the context of developing semantics for reactive systems [54, 62, 1]; see Section 8 for a review of related literature. However, there has been little work on formalizing the notion in a theorem prover with the goal to verify concrete program implementations.[1] Consequently, in refinement proofs of concrete implementations, such notions typically have been argued informally in the metatheory, for instance to manually justify proof rules that are then used by the theorem prover [51]. Such an approach, however, is not amenable to building a robust verification framework. Success in deductive reasoning of a complex program implementation entails careful orchestration of proof strategies. Every verification problem is different, and the framework must support sound extension of the repertoire of proof rules to decompose the problem at hand effectively. This in turn necessitates the notion of correspondence itself to be formalized in the theorem prover in a form suitable for mechanical derivation of proof rules. We show how our approach affords both the definition of intuitive specifications, sound extension of proof

[1] One notable exception is the Atelier B approach, which has been mechanized in the B theorem prover. See Section 8.

strategies, and tools to automate verification of concrete program implementations by incorporating domain insights.

In spite of the classic nature of this notion of correspondence, the formalization of stuttering trace containment in ACL2 is not trivial. The complications arise from the limited expressive power of ACL2: the logic is first-order and has limited support for reasoning about infinitary objects. One result of this paper is to show how to circumvent these limitations and usefully formalize the notion in the ACL2 logic. Using the formalization, we mechanically derive a collection of *reduction theorems* that can then be used as proof rules to decompose the verification of complex concurrent systems into a number of manageable proof obligations. In practice, the theorems reduce the proof of stuttering trace containment to the problem of proving the invariance of certain predicates. We build tools based on predicate abstraction to substantially automate the latter problem. The theorems and tools together constitute a framework. We demonstrate the applicability of the framework in the verification of several concurrent protocol implementations.

The remainder of the paper is organized as follows. In Section 2, we review the ACL2 logic and discuss formal models of reactive programs. Sections 3 through 6 present different facets of the framework, including the formalization of stuttering trace containment, some illustrative reduction theorems, integration of a formal notion of fairness, and a tool for automating invariant proofs. In Section 7 we demonstrate applications by verifying a number of concurrent programs. In Section 8 we discuss related work. In Section 9 we present arguments for the suitability of our approach in reasoning about large concurrent programs. We conclude in Section 10.

Although ACL2 is used as a formal basis of the work, the paper itself assumes no familiarity with ACL2. We use standard mathematical notation instead of ACL2's Lisp syntax, and the relevant features of the logic are reviewed in Section 2.1. For the interested reader, preliminary ACL2 scripts for verification of some of the protocols described in this paper are distributed with the theorem prover in the directory `books/concurrent-programs/`. The description here is consistent with these scripts but we focus on the high-level structure rather than the details of the mechanized proofs. A short five-page position statement summarizing the framework was presented previously in EC^2 2011 [66]; technical details of some of the components of the framework, as well as verification efforts for some of the applications have been published in previous papers [70–73, 65]. This paper describes the overall framework in greater detail, focusing on how the individual pieces fit together and are orchestrated and how the framework is used in formal verification of diverse concurrent programs.

2 Reactive Concurrent Programs in ACL2

We start with a review of the ACL2 logic and discuss how concurrent programs are modeled in ACL2. In this review we limit ourselves to the facets of the logic that are relevant to our work. The reader interested in ACL2 is encouraged to consult the ACL2 web page [36] for an extensive description of the theorem prover.

2.1 The ACL2 Logic

ACL2 is a first-order logic. The kernel of the logic [37] includes (1) a formal syntax for terms, (2) some axioms, and (3) some rules of inference. The kernel syntax describes terms composed of variables, constants, and function symbols each applied to a fixed number of argument terms. The only relation symbol is equality; thus, formulas are constructed from equalities between terms by the usual propositional connectives.

The formal syntax of ACL2 is the prefix-normal syntax of Lisp: the application of a binary function f on arguments a and b is represented by `(f a b)` rather than the more standard $f(a, b)$. However, in this paper we typically use the latter form, although we sometimes refer to the concrete syntax for clarity and disambiguation. We also use conventional notations for some commonly used functions. For instance we write binary arithmetic functions in infix (*e.g.*, $(x + y)$ instead of `(+ x y)` and (*if* x *then* y *else* z) instead of `(if x y z)`), dropping parentheses when it is unambiguous to do so.

ACL2 provides axioms formalizing about 200 functions. These axioms constitute the initial boot-strap theory, called the *ground-zero theory* ($\mathcal{GZ}$ for short). Every theory in ACL2 is constructed from $\mathcal{GZ}$ by application of extension principles (see below). The axioms of $\mathcal{GZ}$ include most Common Lisp functions that are free from side effects. For example, the following are two axioms relating the Common Lisp functions *cons*, *car*, and *cdr*.

Axioms.
car(*cons*$(e, x)) = e$
cdr(*cons*$(e, x)) = x$

The syntax of the logic is quantifier-free. The semantics of a formula assumes implicit universal quantification over all its free variables. For instance, the first axiom can be read: "For all e and x, the function *car* applied to the result of applying *cons* on e and x returns e." Theorems can be proven about the axiomatized functions. The rules of inference constitute propositional calculus with equality and instantiation, together with well-founded induction up to ϵ_0.

The universe of ACL2 is axiomatized to include numbers, characters, strings, ordered pairs, and symbols. Two special symbols included in the universe are `T` and `NIL`. The symbol `T` is axiomatized to be Boolean true, while the symbol `NIL` represents both Boolean false and the empty list. $\mathcal{GZ}$ contains predicates axiomatized to recognize members of each data type: the predicate *consp* holds for an argument x if x is an ordered pair, and *natp* holds if x is a natural number. Note however, that all the Lisp functions axiomatized in ACL2 (contrary to their Common Lisp counterparts) are total. For instance, in Common Lisp the return value of the function *car* is undefined if its argument is not a *cons* and not the symbol `NIL`. But the axioms of ACL2 allow one to prove *car*$(2) =$ `NIL`.

The ACL2 universe also contains representations of ordinals up to ϵ_0. Ordinals [13] are key to well-foundedness arguments in ACL2, and we make extensive use of such arguments. Ordinals are represented in ACL2 using Cantor Normal Form, and $\mathcal{GZ}$ axiomatizes functions that manipulate such representations. In particular, a unary predicate *o-p* and a binary relation $\prec$ are axiomatized as follows: *o-p*(x) holds if x is an (ACL2 representation of an) ordinal, and $\prec$ is the standard total "less than" relation on the ordinals. The ordinals are axiomatized to be well-founded under $\prec$, which means that there is no infinitely decreasing chain of ordinals $\langle \ldots o_2 \prec o_1 \prec o_0 \rangle$.

In addition to formalizing Lisp functions, ACL2 provides *extension principles* to axiomatize new function symbols. The extension principles include (1) a *definitional principle* for introducing new total (recursive) function definitions, (2) an *encapsulation principle* for introducing constrained, partially defined function symbols, and (3) a *defchoose principle* for introducing Skolem functions and first-order quantified formulas.[2]

Definitional Principle: The most commonly used principle is the *definitional principle* that is used to introduce new total function definitions. For example, we can invoke it to introduce a function *factorial* axiomatized to compute the factorial.

Definitional Axiom.
$\mathit{factorial}(x) = \mathit{if}\ \mathit{zp}(x)\ \mathit{then}\ 1\ \mathit{else}\ x * \mathit{factorial}(x-1)$

Here $\mathit{zp}(x)$ (axiomatized in $\mathcal{GZ}$) returns `NIL` if and only if x is a positive natural number and `T` otherwise. The effect of the definition is to extend the logic with an axiom (called the *definitional axiom*) that equates calls to the function *factorial* with its body. To ensure consistency of the extended logic, one must first prove that the recursive calls terminate [9]. This is done by exhibiting a "measure" that maps the arguments of the function to the ordinals, and shows that the measure decreases (according to the relation $\prec$) along every recursive call. For this *factorial* function, an appropriate measure is the function *nfix* below, which is axiomatized in ACL2. It always returns a natural number and hence an ordinal.

Definitional Axiom.
$\mathit{nfix}(x) = \mathit{if}\ \mathit{natp}(x)\ \mathit{then}\ x\ \mathit{else}\ 0$

Encapsulation Principle: The encapsulation principle permits introduction of a set of function symbols with a set of user-specified constraints rather than complete definitions. Consider introducing three functions, *ac*, *ac-id*, and *ac-p*, axiomatized to satisfy the following five constraints.

Encapsulation Axioms.
$\mathit{ac}(\mathit{ac}(x,y),z) = \mathit{ac}(x,\mathit{ac}(y,z))$
$\mathit{ac}(x,y) = \mathit{ac}(y,x)$
$\mathit{ac\text{-}p}(\mathit{ac}(x,y)) = \mathtt{T}$
$\mathit{ac\text{-}p}(\mathit{ac\text{-}id}()) = \mathtt{T}$
$\mathit{ac\text{-}p}(x) \Rightarrow \mathit{ac}(\mathit{ac\text{-}id}(),x) = x$

The axioms merely stipulate that *ac* is an associative-commutative function always returning an object in the domain recognized by the predicate *ac-p*, and *ac-id* is an identity over that domain. To ensure consistency, the user has to exhibit some total function, called the *witness*, that satisfies the constraints. For our example, the following functions are appropriate witnesses.

$\mathit{ac}(x,y) = 42$
$\mathit{ac\text{-}p}(x) = \mathit{if}\ (x = 42)\ \mathit{then}\ \mathtt{T}\ \mathit{else}\ \mathtt{NIL}$
$\mathit{ac\text{-}id}() = 42$

[2] ACL2 also has an extension principle that permits introduction of an arbitrary formula as an axiom. However, introducing arbitrary axioms can introduce inconsistency, and we will not consider it in this paper.

Definitional Axiom.

ac-list(l) = *if* ¬*consp*(l) *then* `NIL`
else if ¬*consp*(*cdr*(l)) *then* *car*(l)
else *ac*(*car*(l), *ac-list*(*cdr*(l)))

Fig. 1 Definition of *ac-list*.

For functions admitted using encapsulation, the only axioms introduced are the constraints. Thus any theorem about such functions is valid for any other set of functions that also satisfy the constraints. This observation is encoded in ACL2 by a derived rule of inference called *functional instantiation* [8]. To illustrate functional instantiation, consider the function *ac-list* shown in Fig. 1, which returns the result of applying *ac* to a list of objects. It is straightforward to prove the theorem named `ac-reverse-relation` below, which states that the repeated application of *ac* along the list l has the same effect as the repeated application of *ac* along the list obtained by reversing l. Here the list reversing function, *reverse*, is axiomatized in $\mathcal{GZ}$.

Theorem `ac-reverse-relation`**:**
ac-list(*reverse*(l)) = *ac-list*(l)

In order to use functional instantiation, we note that the binary addition function "+" is associative and commutative and always returns a number, 0 is the left identity over numbers, and $\mathcal{GZ}$ has a predicate *acl2-numberp* that returns `T` if and only if its argument is a number. Thus, if we define the function *sum-list* that repeatedly adds all elements of a list l, then functional instantiation of the theorem `ac-reverse-relation` enables us to prove the following theorem under the functional substitution of "+" for *ac*, 0 for *ac-id*, *acl2-numberp* for *ac-p*, and *sum-list* for *ac-list*.

Theorem `sum-reverse-relation`**:**
sum-list(*reverse*(l)) = *sum-list*(l)

Defchoose Principle: The *defchoose principle*, which essentially implements the Hilbert choice operator, allows the extension of a theory with a Skolem function. Although the syntax of the ACL2 logic is quantifier-free, defchoose permits arbitrary quantified first-order formulas by the axiomatization of appropriate Skolem witnesses. Let τ be a term with free variables $x_1, \ldots, x_n, y$; then defchoose allows us to extend $\mathcal{T}$ with a new n-ary function symbol f with the following *defchoose axiom* for function f, formula τ, and sequence $\langle x_1, \ldots, x_n \rangle$ of variables.

Defchoose Axiom.
$\tau \Rightarrow \tau/\{y \leftarrow f(x_1, \ldots, x_n)\}$

To illustrate how the defchoose principle supports quantification, assume that P is a binary predicate introduced in an ACL2 theory, and we wish to introduce a unary predicate *exists-P* such that *exists-P*(x) holds if and only if there exists some y such that $P(x, y)$. In traditional first-order logic, we would define *exists-P* as follows:

$$\textit{exists-P}(x) = \exists y : P(x, y)$$

To mimic the same effect in ACL2, we first introduce a unary function *wit*(x) as follows:

Defchoose Axiom.
$P(x, y) \Rightarrow P(x, \mathit{wit}(x))$

The function $\mathit{wit}(x)$ is a Skolem function, and the principle is essentially a rendering of Hilbert choice in the ACL2 logic. We then introduce our desired predicate *exists-P* through the definitional principle.

Definitional Axiom.
$\mathit{exists\text{-}P}(x) = P(x, \mathit{wit}(x))$

ACL2 provides a macro called `defun-sk` to define quantified formulas via Skolemization conveniently by applying the above two-step process. The macro also supports universally quantified predicates by exploiting the duality between universal and existential quantification. For more details on quantification in ACL2, refer to the documentation topics `defchoose`, `defun-sk`, and `conservativity-of-defchoose` in the ACL2 User Manual.

Quantification is critical to derive some of the proof rules we describe in Section 4, although we do not show explicit application of the facility in the paper.

2.2 Modeling Reactive Systems

We model a reactive concurrent program as a labeled transition system. Formally, a reactive system M is described by three functions, namely $M.\mathit{init}$, $M.\mathit{next}$, and $M.\mathit{label}$, which are interpreted as follows.

- $M.\mathit{init}()$ returns the initial state of the system.
- Given a state s and input i, $M.\mathit{next}(s, i)$ returns the next state of M.
- For any state s, $M.\mathit{label}(s)$ returns the observable component of s.[3]

Concurrent programs in practice are not written using this formalism. Details of the translation of a concurrent program implementation (written in C, Java, or Verilog) into this formulation are beyond the scope of this paper. But the basic idea is that for a given concurrent program, the definitions of $M.\mathit{init}$ and $M.\mathit{next}$ correspond to the initial state and state transformations derived from the execution of the concurrent program within the context of the formal semantics of the programming language. The definition of $M.\mathit{label}$ is determined by the components of the program relevant to the notion or specification of correct execution of the concurrent program. We can automatically translate programs in a limited subset of C, Java, and Verilog, into this formalism. We have defined a macro `define-system` to make it convenient for the user to define a concurrent program with this formalism. The macro takes the name of a system M, together with the characterizations (often obtained from translation of C, Java, or Verilog implementation) of its initial state, transition function, and observable components; it introduces the three functions $M.\mathit{init}$, $M.\mathit{next}$, and $M.\mathit{label}$ and the $M.\mathit{exec}[\mathit{env}]$ defined below (for an uninterpreted unary function *env* that models the sequence of external stimuli).

Remark 1 Following standard practice in the ACL2 community, we model transitions in the system as a next-state function $M.\mathit{next}$ rather than as a relation over states (as is typical, for example, with formalisms based on Kripke structures). Of particular interest, the next-state function takes the additional input i. Although this parameter

[3] The function $M.\mathit{label}$ is akin to the *state labeling function* in Kripke Structures.

models the environmental stimulus to the system, our foundations do not require the input to be externally visible. Thus one way to view the stimulus is as a "choice" parameter to non-deterministically select one of the possible next states; this provides a straightforward approach to mimic relational models within our formalism. We will not consider relational models in the remainder of the paper, but most of our framework can be used without modification for such models.

Remark 2 Given the use of a single 0-ary function $M.\mathit{init}()$ to model the initial state, it may at first appear that our formalization does not account for concurrent programs with multiple initial states. However, recall that functions can be introduced in ACL2 through the encapsulation principle by specifying a set of constraints rather than a complete definition. This allows us to formalize programs with more than one initial state as follows. Suppose that the initial states of M are stipulated by some initial condition P which is a predicate on the states of M. We then formalize $M.\mathit{init}()$ as an encapsulated function with the associated constraint $P(M.\mathit{init}()) = \mathtt{T}$. Our macro `define-system` mentioned above permits such formalization.

It will be convenient to talk about the state that M reaches after n transitions. This state depends on the sequence of external stimuli applied to M. Let *env* be a unary function such that $\mathit{env}(k)$ is the stimulus received at time k. Then the function $M.\mathit{exec}[\mathit{env}]$ below returns the state of M after n transitions, and can be introduced in ACL2 by the definitional principle.

Definitional Schema.
$M.\mathit{exec}[\mathit{env}](n) = \mathit{if}\ \mathit{zp}(n)\ \mathit{then}\ M.\mathit{init}()\ \mathit{else}\ M.\mathit{next}(M.\mathit{exec}[\mathit{env}](n-1), \mathit{env}(n-1))$

The choice of the unconventional names above illustrates one of the "tricks" involved in formalizing reactive systems in the logic of ACL2. Formally, this definitional axiom should be more appropriately thought of as a "definition schema": given two different functions env_1 and env_2 prescribing two different environmental stimulus sequences, this schema provides two different definitions. Because the executions of reactive systems involve infinite computations, formalizing them necessarily involves infinite sequences of system states. An infinite sequence of inputs is typically viewed as a *function* over a natural-valued time; however, because the logic is first-order, functions in ACL2 cannot take arbitrary functions as arguments. Thus, the notion of an infinite execution of a system must be written as a schema rather than as a single, closed-form definition. Given the schema, we view the function $M.\mathit{exec}[\mathit{env}]$ as representing the infinite execution sequence of M when presented with the (infinite) sequence of stimuli represented by the function *env*.

Remark 3 Note that in the logic there is obviously no function $M.\mathit{exec}$ that can take a parameter *env*; rather, whenever the metatheory demands the use of such a function (for a specific defined or constrained function *env*), an instance of $M.\mathit{exec}[\mathit{env}]$ is introduced "on the fly".[4] The framework supports introduction of such functions through appropriate use of macros. The same remark applies to other higher-order functions introduced in the metatheory.

[4] The function actually introduced in ACL2 logic is `|M.exec[env]|`. Lisp permits unconventional names (*e.g.*, names containing square brackets) when such symbols are enclosed in vertical bars.

Definition.

$\mathit{stutter}[\mathit{ctr}](n) = (\mathit{ctr}(n) \prec \mathit{ctr}(n-1))$
$M.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n) = \mathit{if}\ \mathit{zp}(\mathrm{n})\ \mathit{then}\ \mathrm{M}.\mathit{init}()$
$\quad \mathit{else\ if}\ \mathit{stutter}[\mathit{ctr}](n)\ \mathit{then}\ M.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n-1)$
$\quad\quad \mathit{else}\ M.\mathit{next}(M.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n-1), \mathit{stim}(n-1))$

Fig. 2 Definition of a Stuttering Trace

3 Stuttering Trace Containment

We will now formalize the notion of stuttering trace containment in the ACL2 logic. Informally, the statement we wish to express is the following: a system $\mathcal{S}$ is related to $\mathcal{I}$ by trace containment if and only if for every execution σ of $\mathcal{I}$ there exists an execution π of $\mathcal{S}$ with the same (infinite) sequence of labels up to finite stuttering. This statement includes notions such as "for all executions of $\mathcal{I}$" and "there exists an execution of $\mathcal{S}$"; a direct formalization of this notion requires quantification over functions. To achieve this in ACL2, we will exploit the encapsulation principle. First we need this technical definition.

Definition 1 (Stuttering Controller) A unary function ctr will be called a *stuttering controller* if the following formula is a theorem.

Well-foundedness Requirement.
$\mathit{o\text{-}p}(\mathit{ctr}(n))$

We will view the environment stimulus function $\mathit{env}(n)$ as a pair $\langle \mathit{stim}(n), \mathit{ctr}(n) \rangle$, where stim is a unary function over natural numbers. Informally, we wish to view the environment as providing, in addition to the actual input stimulus (defined by stim), a sequence that determines if a transition is stuttering or not.

We now define the function $M.\mathit{trace}[\mathit{stim}, \mathit{ctr}]$ in Fig. 2 to formalize stuttering trace. Informally, a stuttering trace of M is simply an execution of M in which some states are repeated a finite number of times. The **Well-foundedness Requirement** and the definition of $\mathit{stutter}[ctr]$ together guarantee that stuttering is finite. We insist that stuttering be finite because we want our abstract specifications to characterize both safety and liveness properties of the implementation. We will return to the significance of the finiteness of stuttering in Section 9.

The following definition now captures the notion of stuttering trace containment.

Definition 2 (Stuttering Refinement) Let tstim and tctr be functions such that (1) tstim is uninterpreted, and (2) tctr is constrained to be a stuttering controller. We will say that $\mathcal{I}$ is a *stuttering refinement* of $\mathcal{S}$, denoted $(\mathcal{S} \rhd \mathcal{I})$ if and only if there are unary functions stim and ctr such that ctr is a stuttering controller and the following condition is satisfied.

Stuttering Trace Containment (STC).
$\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{trace}[\mathit{tstim}, \mathit{tctr}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n))$

We refer to the system $\mathcal{S}$ as a *stuttering abstraction* of $\mathcal{I}$.

For a given implementation system $\mathcal{I}$ and a specification system $\mathcal{S}$, our notion of correctness is to show $(\mathcal{S} \rhd \mathcal{I})$. Once the functions ctr and stim are defined, the characterization reduces to a first-order obligation that can be proven with ACL2.

Remark 4 The notion of stuttering trace containment permits stuttering on both $\mathcal{S}$ and $\mathcal{I}$, all the concurrent programs we verified required only $\mathcal{S}$ to stutter. This is natural; because the atomicity of the implementation is typically more fine-grained than that of the specification, it is expected that several transitions of the implementation correspond to one transition of the specification. Nevertheless, we decided to define specifications in terms of two-sided stuttering rather than one-sided for two reasons. First, we believe that a uniform treatment of specification and implementation systems (*viz.*, removal of stutter on both sides) provides a more intuitive definition of correspondence between the two systems. Second, it facilitates proof of the **Oblivious Refinement Rule** (cf. Section 4.3) which allows introduction of auxiliary variables.

4 Reduction Theorems

Given two systems $\mathcal{I}$ and $\mathcal{S}$, we now discuss how we go about proving $(\mathcal{S} \rhd \mathcal{I})$. When $\mathcal{I}$ is a complex low-level implementation and $\mathcal{S}$ is an abstract high-level specification, the proof needs to be decomposed into more tractable obligations. To facilitate such decomposition, we have formalized and mechanically derived a number of *reduction theorems*, which are used as mechanized proof rules. Here we describe three such rules, that permit us to decompose the proof of **STC** to a series of refinements, break each such refinement proof obligation to a collection of single-step theorems, and enable the introduction of auxiliary variables. Section 7 explains the use of the rules on illustrative examples.

4.1 Reduction via Stepwise Refinement

The first trivial property of **STC** is that the notion is transitive. This observation is formalized by the following proof rule.

Stepwise Refinement Rule.
Derive $(\mathcal{S} \rhd \mathcal{I})$ from $(\mathcal{S} \rhd \mathcal{I}_1)$ and $(\mathcal{I}_1 \rhd \mathcal{I})$

The stepwise refinement rule allows us to introduce a sequence of intermediate models at different levels of abstraction starting from the implementation $\mathcal{I}$ and leading to the specification $\mathcal{S}$, and prove $(\mathcal{S} \rhd \mathcal{I})$ by showing correspondence between each pair of consecutive models in the sequence.

Remark 5 The statement of the stepwise refinement rule, albeit simple, is higher-order. We mechanize it in ACL2 by using the encapsulation principle and macros. In particular, we use encapsulation to introduce the three systems $\mathcal{S}$, $\mathcal{I}_1$, and $\mathcal{I}$ with the only exported constraints being the properties $(\mathcal{S} \rhd \mathcal{I}_1)$ and $(\mathcal{I}_1 \rhd \mathcal{I})$. From these conditions, we develop an ACL2 proof of $(\mathcal{S} \rhd \mathcal{I})$. We call this theorem a *generic theory for stepwise refinement*. Subsequently, for concrete systems, (*i.e.*, specific instances $\mathcal{S}_c$, $\mathcal{I}_{1c}$, and $\mathcal{I}_c$ of $\mathcal{S}$, $\mathcal{I}_1$, and $\mathcal{I}$), applying the stepwise refinement rule amounts to functional instantiation of the generic theory. We implement a macro named `defstepwise` that automates the application of the rule; it takes the definitions of $\mathcal{S}_c$, $\mathcal{I}_{1c}$, and $\mathcal{I}_c$, and functionally instantiates the generic theory. The proof obligations generated by the functional instantiation are exactly that $(\mathcal{S}_c \rhd \mathcal{I}_{1c})$ and $(\mathcal{I}_{1c} \rhd \mathcal{I}_c)$. A similar approach (generic theory, functional instantiation, and a macro to automate the application of

functional instantiation) is used for mechanizing the rest of the rules presented in this paper as well. For the remainder of the paper, we omit discussion on mechanization of the rules and treat them as closed-form higher-order theorems.

4.2 Reduction to Single-step Theorems

Using the stepwise refinement rule, we can transform a proof of stuttering trace containment into a series of more tractable trace containment proofs using intermediate systems. Our next rule is intended to reduce each of these individual proofs to the definition of a more tractable collection of proof obligations. In particular, none of the following proof obligations requires reasoning about more than a single transition of either $\mathcal{S}$ or $\mathcal{I}$.

Definition 3 (Well-founded Refinement) Given two systems $\mathcal{S}$ and $\mathcal{I}$, we will say that $\mathcal{I}$ is a *well-founded refinement* of $\mathcal{S}$, written ($\mathcal{S} \trianglerighteq \mathcal{I}$) if and only if there exist functions *inv*, *skip*, *rep*, *rank*, *good*, and *pick* such that the following formulas are theorems for all states s and inputs i.

SST1: $\mathit{good}(s) \Rightarrow \mathcal{I}.\mathit{label}(s) = \mathcal{S}.\mathit{label}(\mathit{rep}(s))$
SST2: $\mathit{good}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathcal{I}.\mathit{next}(s,i)) = \mathit{rep}(s)$
SST3: $\mathit{good}(s) \wedge \neg\mathit{skip}(s,i) \Rightarrow \mathit{rep}(\mathcal{I}.\mathit{next}(s,i)) = \mathcal{S}.\mathit{next}(\mathit{rep}(s), \mathit{pick}(s,i))$
SST4: $\mathit{good}(s) \wedge \mathit{skip}(s,i) \Rightarrow \mathit{rank}(\mathcal{I}.\mathit{next}(s,i)) \prec \mathit{rank}(s)$
SST5: $\mathit{o\text{-}p}(\mathit{rank}(s))$
SST6: $\mathit{inv}(\mathcal{I}.\mathit{init}())$
SST7: $\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathcal{I}.\mathit{next}(s,i))$
SST8: $\mathit{inv}(s) \Rightarrow \mathit{good}(s)$

These conditions are influenced by similar obligations devised by Manolios et al. [51] for well-founded bisimulations (WEBs). Given a state s of $\mathcal{I}$ the function *rep*(s) returns a corresponding state of $\mathcal{S}$ with the same label. The predicate *skip* governs stuttering. If *skip*(s, i) does not hold then **SST3** guarantees that $\mathcal{S}$ has a transition that matches the transition of $\mathcal{I}$ from state s on input i, otherwise **SST2** guarantees that a stuttering transition of $\mathcal{S}$ matches the transition of $\mathcal{I}$ on the current input. if $\mathcal{S}$ does not stutter, then *pick*(s, i) defines the input of $\mathcal{S}$ for the matching transition of $\mathcal{I}$ from state s on input i. **SST4**, **SST5** and the well-foundedness of the ordinals guarantee that stuttering is finite. **SST6** and **SST7** specify that the predicate *inv* is an *inductive invariant* of $\mathcal{I}$. That is, *inv* holds at $\mathcal{I}.\mathit{init}()$ and if it holds at a state s then it holds after any transition from s. **SST8** stipulates that the predicate *good* is logically implied by *inv*. Thus *good* is an invariant, *i.e.*, it must hold for all reachable states of $\mathcal{I}$. This allows us to assume *good*(s) in the hypothesis of conditions **SST1**-**SST4**.

Remark 6 It is possible to use *inv* in conditions **SST1**-**SST4** above in place of *good* (and thereby eliminate the need for *good*); thus the use of a separate predicate *good* that is logically implied by *inv* is not germane to the metatheory. However, the use of *good* illustrates an important practical consideration. Note that *inv* is required (by Conditions **SST7** and **SST8**) to be an inductive invariant. Constructing the inductive invariant is a complex process; in practice it is the most complicated step in the proof of well-founded refinement. Thus it is important to isolate that step as much as possible from the remaining proof obligations. The use of *good* disentangles the

proof obligations to ensure correspondence between the executions of $\mathcal{I}$ and $\mathcal{S}$ (*viz.*, **SST1**-**SST4**) from the process of discovery and proof of an inductive invariant. In our experience, it is easy to construct a (not necessarily inductive) invariant *good* to satisfy these proof obligations. Furthermore, the predicate *good* so defined can be used as "seed" to compute the inductive invariant *inv*. In Section 5 we discuss an invariant discovery procedure based on predicate abstraction, that essentially involves strengthening a user-proposed invariant *good*. Admittedly, it is possible for the user to define a predicate *good* to satisfy **SST1**-**SST4** only to find subsequently that *good* is not an invariant (*viz.*, there is no inductive invariant *inv* implying *good*). However, we found in practice that defining a correct and sufficient (non-inductive) invariant rarely involves subtle reasoning, unlike the definition of the inductive invariant *inv*.

Well-founded Refinement Rule.
Derive $(\mathcal{S} \rhd \mathcal{I})$ from $(\mathcal{S} \trianglerighteq \mathcal{I})$

The proof of correctness of this rule follows the lines of the proof of the corresponding rule about WEBs and has been mechanically checked with ACL2. Notice that none of the conditions involves more than one transition of $\mathcal{S}$ or $\mathcal{I}$. However, while stuttering trace containment allows both $\mathcal{S}$ and $\mathcal{I}$ to stutter, the well-founded refinement rule only allows stuttering in $\mathcal{S}$. We found this sufficient in practice because $\mathcal{S}$ is designed to be an abstract system with coarse transitions that correspond to sequences of fine transitions in $\mathcal{I}$.

Finally, while the notion of correspondence specified by stuttering trace containment is linear-time, **SST1**-**SST8** guarantees that all branching-time behaviors of $\mathcal{I}$ are preserved by $\mathcal{S}$ (up to stuttering). In other words, well-founded refinements guarantee that $\mathcal{S}$ is a *stuttering simulation* of $\mathcal{I}$. We make use of this observation in the derivation of the next proof rule. It is possible to formalize stuttering simulation directly in the ACL2 logic (analogous to the way we formalized stuttering trace containment via encapsulation). Indeed, Manolios [49] provides a collection of sound and complete rules of stuttering simulation analogous to the well-founded refinement rule. Nevertheless, in the context of ACL2, infinite trees are more cumbersome to formalize than infinite sequences, and given our need to add and prove new proof rules in the meta-theory and the sufficiency of trace containment as a means of specification, we prefer the linear time notion of trace containment as the formal notion of correctness.

4.3 Equivalences and Auxiliary Variables

In this section we consider a special (but important) case of refinement, *viz.*, equivalence.

Definition 4 (Equivalence up to Stuttering) If $(\mathcal{S} \rhd \mathcal{I})$ and $(\mathcal{I} \rhd \mathcal{S})$ both hold then we say that $\mathcal{S}$ is *equivalent* to $\mathcal{I}$ (up to stuttering). We write $(\mathcal{S} \diamond \mathcal{I})$ to mean that $\mathcal{S}$ and $\mathcal{I}$ are equivalent.

One way of proving equivalence is through *oblivious well-founded refinements*, defined below. For that definition, recall that one of the conditions for well-founded refinements (*viz.*, **SST3**) required the existence of a function *pick*: given s and i, $\mathit{pick}(s, i)$ returns the matching input for the specification system $\mathcal{S}$ corresponding to a non-stuttering transition of $\mathcal{I}$ from state s on input i.

Definition 5 (Oblivious Refinement). We call $\mathcal{I}$ an *oblivious refinement* of $\mathcal{S}$, written $(\mathcal{S} \trianglerighteq_o \mathcal{I})$ if the following two conditions hold:

1. $(\mathcal{S} \trianglerighteq \mathcal{I})$
2. the function *pick* involved in the proof of (1) is the identity function on its second argument, that is, $\mathit{pick}(s, i) = i$ is a theorem.

The following proof rule connects oblivious refinement with stuttering equivalence.

Oblivious Refinement Rule.
Derive $(\mathcal{S} \diamond \mathcal{I})$ from $(\mathcal{S} \trianglerighteq_o \mathcal{I})$

Oblivious refinements are useful when $\mathcal{S}$ contains more state components than $\mathcal{I}$. To understand their use, consider an example due to Abadi and Lamport [1]. Let $\mathcal{I}$ be a 1-bit digital clock, $\mathcal{S}$ be a 3-bit clock, and the *label* of a state in both systems be the low-order bit. Clearly, $(\mathcal{S} \triangleright \mathcal{I})$ because the systems have the same behavior up to stuttering. However, we cannot prove $(\mathcal{S} \trianglerighteq \mathcal{I})$, because there is no mapping that can define the state of a 3-bit clock as a function of 1 bit. But it is trivial to show $(\mathcal{I} \trianglerighteq_o \mathcal{S})$: given a state s of $\mathcal{S}$, the function *rep* merely projects the low-order bit. Then we can use the **Oblivious Refinement Rule** to show the desired result.

Oblivious refinements allow us to add *auxiliary variables* to systems. For showing $(\mathcal{S} \triangleright \mathcal{I})$, we often construct an intermediate system $\mathcal{I}^+$ as follows. A state of $\mathcal{I}^+$ includes the variables of a state of $\mathcal{I}$ but also has additional variables. For instance, one sometimes needs *history variables* that explicitly store the control decisions and non-deterministic choices encountered by the system [68]. The variables common to $\mathcal{I}$ and $\mathcal{I}^+$ are updated in exactly the same manner in both systems and the *label* of a state in $\mathcal{I}^+$ is the same as the label of a state in $\mathcal{I}$. The information stored in these additional variables is then used to facilitate proving $(\mathcal{S} \trianglerighteq \mathcal{I}^+)$. In order to prove $(\mathcal{S} \triangleright \mathcal{I})$ we must now show $(\mathcal{I}^+ \triangleright \mathcal{I})$ so that we can apply the rule **SR**. However, because $\mathcal{I}^+$ contains more state components than $\mathcal{I}$ we cannot directly prove $(\mathcal{I}^+ \trianglerighteq \mathcal{I})$. However, we can easily prove $(\mathcal{I} \trianglerighteq_o \mathcal{I}^+)$ and invoke the **Oblivious Refinement Rule** to derive $(\mathcal{I}^+ \diamond \mathcal{I})$.

The soundness of introducing auxiliary variables for proving system correspondence is normally assumed in the metatheory. Oblivious refinements reduce it to an automatic proof step requiring no metatheoretic justification. We note that the oblivious refinement rule allows augmentation of systems with both history and prophecy variables.

5 Automating Invariant Proofs

The reduction theorems allow us to decompose the proof of correspondence between an implementation and its specification to a sequence of correspondences by defining intermediate models, augmenting them with auxiliary variables, and proving the correspondence between each pair of consecutive models in the sequence by exhibiting well-founded refinements. Chaining together a sequence of refinement steps merely involves functional instantiation of the **Stepwise Refinement Rule**. User attention is thus focused on the definition of intermediate models and the proofs of well-founded refinements relating the consecutive pairs of the sequence.

In practice, the proof of $(\mathcal{S} \trianglerighteq \mathcal{I})$ is decomposed into two phases:

1. Define *good*, *rep*, *skip*, *pick*, and *rank* and prove obligations **SST1**-**SST5**.

2. Define *inv* and prove **SST6-SST8**.

In our experience, the first phase is relatively straightforward. For instance, assume that $\mathcal{I}$ is a multiprocess system implementing a cache coherence protocol and that $\mathcal{S}$ is a system of processes that atomically access and update the main memory. We will see such system examples in Section 7. In a typical proof of such a system, *good* needs to posit that the caches are coherent, *rep* projects the visible components (processes and memory) of the cache system, *skip* holds for transitions that cause no access to the memory, and *rank* counts the number of transitions before a visible component is updated.[5] The proof obligations **SST1-SST5** required for well-founded refinements can usually be discharged without significant manual effort.

In the second phase, the user must define an inductive invariant *inv* that logically implies *good*. Recall that the existence of the inductive invariant guarantees that *good* is an *invariant*, that is, holds for all reachable states of $\mathcal{I}$. Defining *inv*, however, requires non-trivial user insight, since by condition **SST7** *inv* must be preserved by *every* transition of $\mathcal{I}$. Thus the definition of *inv* may have to consider every reachable state. Defining inductive invariants is recognized as one of the most expensive steps in formal verification [48]. Furthermore, changes in the implementation can require substantial modifications to the definitions of inductive invariants.

We have implemented a tool based on *predicate abstractions* [27] to significantly automate invariant proofs. The goal is to reduce an invariant proof of a (possibly infinite-state) system to model-checking on a finite abstraction; the states of the abstract model correspond to valuations of predicates in the concrete system. Predicate abstraction is part of a number of formal verification tools [5,30,42]. Our approach can be viewed as an implementation of Namjoshi and Kurshan's [61] idea of predicate abstraction through syntactic transformation of certain formulas over the system actions. However, our approach is designed to leverage the expressiveness and flexibility of theorem proving for discovering predicates: useful predicates are "mined" by applying term rewriting on the definition of the state transition function of the implementation. Rewriting is guided by *rewrite rules* that are taken from theorems proven by the theorem prover. In this section we illustrate the use of this tool with a simple example. Automating invariant proofs, of course, is a topic of independent research interest, and previous papers [72,73,65] cover the technical details of our implementation.

Consider a system $\mathcal{I}$ consisting of two components `C0` and `C1`. Initially both components hold 0. Given a state s and an external stimulus i, the components are updated as follows.

- If i is `NIL`, `C0` gets the previous value of `C1`; otherwise `C0` is unchanged.
- If i is `NIL`, `C1` is assigned the value `42`; otherwise `C1` is unchanged.

For state s, let $\mathit{C0}(s)$ and $\mathit{C1}(s)$ be the values of `C0` and `C1` in s. The predicate *good* that we want to establish as invariant is given by: $\mathit{good}(s) = \mathit{natp}(\mathit{C0}(s))$. An inductive invariant that establishes the invariance of *good* is the predicate *inv* below:

Inductive Invariant Definition.
$\mathit{inv}(s) = \mathit{natp}(\mathit{C0}(s)) \wedge \mathit{natp}(\mathit{C1}(s))$

Instead of requiring the user to define this inductive invariant, the predicate $\mathit{natp}(\mathit{C1}(s))$ is discovered by rewriting. The necessary rewrite rules are shown in Fig. 3. The first

[5] Actually the function *rank* only needs to count an upper bound. This is critical to our ability to support fairness constraints. See Section 6.

Rewrite Rules.
1. $\mathit{C0}(\mathcal{I}.\mathit{init}()) = 0$
2. $\mathit{C1}(\mathcal{I}.\mathit{init}()) = 0$
3. $\mathit{C0}(\mathcal{I}.\mathit{next}(s,i)) = \mathit{if}\ i\ \mathit{then}\ \mathit{C0}(s)\ \mathit{else}\ \mathit{C1}(s)$
4. $\mathit{C1}(\mathcal{I}.\mathit{next}(s,i)) = \mathit{if}\ i\ \mathit{then}\ \mathit{C1}(s)\ \mathit{else}\ 42$
5. $\mathit{natp}(\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = \mathit{if}\ x\ \mathit{then}\ \mathit{natp}(y)\ \mathit{else}\ \mathit{natp}(z)$

Fig. 3 Rewrite Rules for Proving Invariant of the Simple System

four rules are derived from the transition function. Using rules 3 and 5, we rewrite $\mathit{natp}(\mathit{C0}(\mathcal{I}.\mathit{next}(s,i)))$ to the term **T0** below:

$$\textbf{T0:}\quad \mathit{if}\ i\ \mathit{then}\ \mathit{natp}(\mathit{C0}(s))\ \mathit{else}\ \mathit{natp}(\mathit{C1}(s))$$

The tool treats the term **T0** as a Boolean combination of i, $\mathit{natp}(\mathit{C0}(s))$, and $\mathit{natp}(\mathit{C1}(s))$, and classifies $\mathit{natp}(\mathit{C1}(s))$ as a new predicate. Application of term rewriting on the term $\mathit{natp}(\mathit{C1}(\mathcal{I}.\mathit{next}(s,i)))$ using rules 4 and 5 and the computed fact $\mathit{natp}(42) = \mathtt{T}$, yields the term **T1**:

$$\textbf{T1:}\quad \mathit{if}\ i\ \mathit{then}\ \mathit{natp}(\mathit{C1}(s))\ \mathit{else}\ \mathtt{T}$$

We classify the discovered terms $\mathit{natp}(\mathit{C0}(s))$ and $\mathit{natp}(\mathit{C1}(s))$ as *state predicates* (named **S0** and **S1** respectively). Then **T0** and **T1** specify how the state predicates are "updated" at different times. This view is made explicit by constructing a directed *abstraction graph* G as follows.

- A node is a pair $\langle b_1, b_2\rangle$, where $b_1, b_2 \in \{\mathtt{T}, \mathtt{NIL}\}$. The components correspond to the possible valuations of **S0** and **S1**.
- The *initial node* p_0 is the pair $\langle \mathtt{T}, \mathtt{T}\rangle$ that correspond to the valuation of **S0** and **S1** at the initial state.
- Let $p \doteq \langle b_1, b_2\rangle$ and $q \doteq \langle b_1', b_2'\rangle$ be two nodes. There is an edge from p to q if there exists some $b_i \in \{\mathtt{T}, \mathtt{NIL}\}$ such that $b_0' = \mathtt{if}\ (b_i,\ b_0,\ b_1)$ and $b_1' = \mathtt{if}\ (b_i,\ b_1,\ \mathtt{T})$.

We can prove that *good* is an invariant by checking that in each node p of G that is reachable from p_0, the first component has the value T. The formula representing the set of reachable states of G defines the relevant inductive invariant of $\mathcal{I}$.

Our tool demonstrates the flexibility afforded by the use of general-purpose mechanical theorem proving as a formal basis for designing a verification framework. In particular, heuristics for predicate discovery are disentangled from the abstraction process and encoded in rewrite rules. Thus by "plugging in" different sets of rewrite rules we can apply the *same* tool on diverse application domains. No restriction is imposed on the language to express systems and their properties.

Of course, our approach puts the onus on the user to define and prove rewrite rules. When the current set of rewrite rules is not sufficient, simplification may not succeed resulting in models that are too detailed or too abstract. However, rewrite rules, unlike inductive invariants, are *generic* properties of the functions involved in the definition of the system and its properties, and the same rules are useful for reasoning about *different* systems modeled using the same functions. ACL2 has several databases of lemmas [11, 38, 19], and we found that most of the necessary rules are usually already available in such libraries. Even domain-specific rules are typically reusable for verification of

different programs in the same domain; we will see an example in Section 7. Feedback from our tool assists in the crafting of new rules. The feedback includes the predicates generated via rewriting with the current rules, highlighting terms that could not be simplified from the current rule set, and directing the user to a set of functions for which rewrite rules are not available. The user is then responsible for extending the current rule set with rewrite rules to simplify terms containing such functions. Note that the presence of a theorem prover is critical to our ability to extend the set of available rules in a sound manner. In our experience, creativity and user insights are necessary to develop the (domain-specific) rules when the tool is applied to a new domain for the first time; however, the library becomes stable after a few applications and can be used as is for a new program in the same domain.

6 Fairness Assumptions and Requirements

We conclude this description of the refinement proof methodology with a discussion of how to deal with fairness conditions integrated with the refinement proofs. Fairness conditions often arise for showing that a system satisfies progress properties [57]. For instance, consider a system $\mathcal{I}$ that models the asynchronous composition of a set of processes. At each transition of $\mathcal{I}$, a process is selected non-deterministically and then updates its local state and any shared state. Assume that any update of a shared variable requires exclusive access and that some arbiter grants access to a single process at a time. We might want to show that each initiated update of a shared variable is eventually completed. However, this property is invalidated in executions of $\mathcal{I}$ in which the process that is given exclusive access is never subsequently selected. A more appropriate formulation of the property is as follows: "Assuming fair selection of processes in $\mathcal{I}$, every update of the shared state completes." We now describe how fairness conditions are integrated within our framework to make such reasonings possible. As with our description of the rest of the framework, our treatment here uses standard mathematical notation to convey the main ideas. For the reader interested in the ACL2 formalization, Sumners [71] discusses the technical issues involved in reasoning about fairness in ACL2.

Recall that a trace of a system $\mathcal{I}$ is specified by defining a stimulus function and a stuttering controller. To formalize the notion of a *fair trace*, we will first introduce the concept of a fair selector by appropriately constraining these two functions in the definition below.

Definition 6 (Fair Selector) We say that a stimulus function *stim* and a stuttering controller *ctr* produce a *fair selector* if for any time n and any input i, there is a time $m > n$ such that the following two conditions hold.

F1: $\mathit{stutter}[\mathit{ctr}](m) = \texttt{NIL}$
F2: $\mathit{stim}(m) = i$

Condition **F1** is necessary to guarantee that the fair input is actually used and not simply bypassed by stuttering. **F1** and **F2** provide a simplistic notion of unconditional (weak) fairness in the ACL2 logic, *viz.*, that the environment selects *each* legal input stimulus infinitely often. Many logics designed with the explicit goal of reasoning about executions of reactive systems provide more general notions of fairness. For instance, Unity [16] has notions of both weak (unconditional) and strong (conditional) fairness.

For the examples we describe in this paper, our simple notion is sufficient. Nevertheless, the framework allows reasoning about more general notions of fairness using approaches similar to the one we describe below. For instance, Sumners' paper [71] discusses an integration of conditional fairness where one associates a set of legal inputs with each system state and fairness ensures that an input that is *infinitely often legal* is selected infinitely often.

We now formalize the notion of a *fair trace*. Similar to the definition of a trace in Section 3, we will define a fair trace using the encapsulation principle. In particular, we specify a pair of functions *fctr* and *fstim*, constrained to produce a fair selector based on the definition above. Defining witness functions that satisfy these constraints, however, is tricky. In particular, we must assume that the set of possible inputs is enumerable.[6] Enumerability is necessary because it is impossible to define an infinite sequence where each member of an uncountable set is selected infinitely often. Given enumerability, it is sufficient to merely exhibit a fair selector for natural numbers.

A fair selector for natural numbers can be easily defined as a state machine, as follows. At any instant the machine has a fixed upper bound, and it counts down from the upper bound at every step. When the countdown reaches 0, the upper bound is incremented by 1 and the counter then resets to the new bound. Since any natural number i becomes eventually less than the ever-increasing bound, each natural number must be eventually selected in a finite number of steps from each instant n.

We will now augment refinements with the notion of fairness. Stuttering trace containment involves specifying correlations between executions of two systems $\mathcal{S}$ and $\mathcal{I}$. Fairness conditions can be imposed on either $\mathcal{S}$ or $\mathcal{I}$ or both. The following definitions capture the relevant fairness characterization.

Definition 7 (Fairness Assumption) Let *cfctr* and *cfstim* be constrained unary functions, with the only constraint being that they produce a fair selector. We say that a system $\mathcal{I}$ is a stuttering refinement of system $\mathcal{S}$ *under fairness assumption* (denoted $(\mathcal{S} \rhd_F \mathcal{I})$) if there exist unary functions *stim* and *ctr* such that the formula **Fair Implementation Correspondence** is a theorem.

Fair Implementation Correspondence.
$\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{trace}[\mathit{cfstim}, \mathit{cfctr}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{trace}[\mathit{stim}, \mathit{ctr}](n))$

Definition 8 (Fairness Requirement) Let *uctr* and *ustim* be constrained unary functions with the only constraint being that *uctr* is a stuttering controller. We say that $\mathcal{I}$ is a stuttering refinement of $\mathcal{S}$ *with fairness requirement*, written $(\mathcal{S}\ _F \rhd \mathcal{I})$, if there are functions *fctr* and *fstim* such that (1) *fctr* and *fstim* produce a fair selector, and (2) the formula **Fair Specification Correspondence** below is a theorem.

Fair Specification Correspondence.
$\mathcal{I}.\mathit{label}(\mathcal{I}.\mathit{trace}[\mathit{ustim}, \mathit{uctr}](n)) = \mathcal{S}.\mathit{label}(\mathcal{S}.\mathit{trace}[\mathit{fstim}, \mathit{fctr}](n))$

We write $(\mathcal{S}\ _F \rhd_F \mathcal{I})$ to mean that both $(\mathcal{S} \rhd_F \mathcal{I})$ and $(\mathcal{S}\ _F \rhd \mathcal{I})$ hold. The following proof rules are trivial to verify with ACL2 and can be used to decompose refinement proofs with fairness conditions.

Derivation Rules for Fairness.
Derive $(\mathcal{S} \rhd_F \mathcal{I})$ from $(\mathcal{S} \rhd \mathcal{I})$.

[6] We have had conversations with the authors of ACL2 regarding extending the ACL2 ground zero theory with an axiom positing enumerability of the universe. If this is done, no assumption will be necessary to formalize fairness.

Derive $(\mathcal{S} \rhd_F \mathcal{I})$ from $(\mathcal{S} \rhd_F \mathcal{R})$ and $(\mathcal{R} \rhd_F \mathcal{I})$.
Derive $(\mathcal{S}_F \rhd_F \mathcal{I})$ from $(\mathcal{S}_F \rhd_F \mathcal{R})$ and $(\mathcal{R}_F \rhd_F \mathcal{I})$.
Derive $(\mathcal{S}_F \rhd \mathcal{I})$ from $(\mathcal{S}_F \rhd_F \mathcal{R})$ and $(\mathcal{R}_F \rhd \mathcal{I})$.

While fairness *assumptions* are necessary for ensuring progress properties of implementations, fairness *requirements* are usually necessary for the purpose of composition. In particular, suppose we want to prove $(\mathcal{S} \rhd \mathcal{I})$ by introducing an intermediate model $\mathcal{R}$. If we need a fairness assumption in the proof of correspondence between $\mathcal{S}$ and $\mathcal{R}$, then chaining the sequence of refinements requires that we prove the correspondence between $\mathcal{R}$ and $\mathcal{I}$ with a fairness requirement.

We now discuss how we make fairness constraints work with the single-step conditions for well-founded refinements. We will first consider fairness assumptions. Two relevant concepts in formalizing fairness as a set of single-step obligation are *fair augmented system* and *fair ranking function*, defined below.

Definition 9 (Fair Augmented System) The system $\mathcal{I}^f$ is a *fair augmentation* of a system $\mathcal{I}$ if the following conditions hold.

- Each state variable of $\mathcal{I}$ is a variable of $\mathcal{I}^f$, but $\mathcal{I}^f$ contains an additional variable *clock* that is not a variable of $\mathcal{I}$. $\mathcal{I}^f$ contains no variable other than the variables of $\mathcal{I}$ and *clock*.
- Each variable v in $\mathcal{I}^f$ that is a variable of $\mathcal{I}$ is updated by $\mathcal{I}^f.\mathit{next}$ exactly the same way as it is updated by $\mathcal{I}.\mathit{next}$. The variable *clock* is 0 at the state $\mathcal{I}^f.\mathit{init}$ and is incremented by 1 at each transition by $\mathcal{I}^f.\mathit{next}$.

Definition 10 (Fair Ranking Function) Let $\mathcal{I}^f$ be a fair augmentation of system $\mathcal{I}$. A binary function frnk will be called a *fair ranking function* if it is an encapsulated function constrained to satisfy the following two conditions.

- $\mathit{natp}(\mathit{frnk}(s, j))$
- $(i \neq j) \Rightarrow \mathit{frnk}(\mathcal{I}^f.\mathit{next}(s, i), j) < \mathit{frnk}(s, j)$

By the above definition, for any legal input j and any state s, if j is not selected in the transition from s then frnk decreases. Because $\mathcal{I}^f$ tracks the "current time", frnk can be constructed from the constrained fair selector using **F1** and **F2** above: $\mathit{frnk}(s, j)$ is merely the number of transitions after s till j is selected. We are now ready to define well-founded refinements under fairness assumption.

Definition 11 (Well-founded Refinement Under Fairness Assumption) Let $\mathcal{I}^f$ be a fair augmentation of system $\mathcal{I}$. We will say that $\mathcal{I}$ *is a well-founded refinement of system* $\mathcal{S}$ *under fairness assumption* (denoted $(\mathcal{S} \trianglerighteq_F \mathcal{I})$) if $(\mathcal{S} \trianglerighteq \mathcal{I}^f)$.

The following proof rule connects well-founded refinements with stuttering trace containment under fairness assumptions.

Well-founded Refinement Rule under Fairness Assumption.
Derive $(\mathcal{S} \rhd_F \mathcal{I})$ from $(\mathcal{S} \trianglerighteq_F \mathcal{I})$.

Note that the definition of well-founded refinement under fairness assumption does not involve fair ranking function but only augmentation. However, fair ranking functions appear in discharging the proof obligations for $(\mathcal{S} \trianglerighteq \mathcal{I}^f)$. In particular, the proof obligations **SST4** and **SST5** for the proof of $(\mathcal{S} \trianglerighteq \mathcal{I}^f)$ (cf. Definition 3) involve showing

that a function *rank* decreases (according to some well-founded order) on every stuttering transition of $\mathcal{I}^f$; we make use of the constrained function *frnk* for defining the appropriate function *rank* to discharge these proof obligations. In particular, a typical application of well-founded refinements under fairness assumptions appears in asynchronous protocols, where the external stimulus selects the index of a process to take the next step from a state s. Suppose that in some state s, process 0 holds the lock to some shared resource, and every other process subsequently waits till the lock is released. To prove that such a protocol is a refinement of one in which processes access the shared resource atomically, we must invoke fairness. To do so, we use well-founded refinements under fairness assumption, by defining the *rank* as a lexicographic tuple containing $\mathit{frnk}(s, 0)$ as a component; we call this component the *fairness measure* for process 0. As long as process 0 is not selected, the value of this component decreases along every transition, which enables us to ensure finiteness of stuttering. We will see a more involved example of this in the proof of the Bakery algorithm in Section 7.3.

We now turn to well-founded refinements under fairness requirements. In order to show $(\mathcal{S} \sqsupseteq_F \mathcal{I})$, we must show that each execution of $\mathcal{I}$ is fair. The definition of *fairness guarantee* formalizes this intuition.

Definition 12 (Fairness Guarantee) We say that a binary function *fsel* provides a *fairness guarantee* with respect to binary functions *skip* and *pick* if and only if the following conditions hold:

1. $\mathit{o\text{-}p}(\mathit{fsel}(s, j))$
2. $\mathit{skip}(s, i) \vee (\mathit{pick}(s, i) \neq j) \Rightarrow \mathit{fsel}(\mathcal{I}.\mathit{next}(s, i), j) \prec \mathit{fsel}(s, j)$
3. $\neg \mathit{skip}(s, i) \Rightarrow \mathit{fsel}(s, j) \not\prec \mathit{fsel}(\mathcal{I}.\mathit{next}(s, i), j)$

Recall that for well-founded refinements, the function *skip* stipulates whether $\mathcal{S}$ stutters on a transition of $\mathcal{I}$ and *pick* determines the matching input for $\mathcal{S}$ on non-stuttering steps. The conditions for fairness ensure that $\mathit{fsel}(s, j)$ returns an ordinal that never increases at any transition until j is used by $\mathcal{S}$ to match a transition from $\mathcal{I}$, while it strictly decreases in the stuttering steps of $\mathcal{S}$. Thus j must be selected eventually by *pick* to match a transition of $\mathcal{I}$.

Definition 13 (Well-founded Refinement Under Fairness Requirement) We then say that $\mathcal{I}$ is a well-founded refinement of $\mathcal{S}$ under fairness requirement (written $(\mathcal{S} \sqsupseteq_F \mathcal{I})$), if the following conditions hold.

1. $\mathcal{S} \sqsupseteq \mathcal{I}$
2. There exists a fairness guarantee function *fsel* with respect to the functions *skip* and *pick* used in the obligations for condition 1.

Finally we relate well-founded refinements with stuttering trace containment with fairness requirement by the following proof rule.

Well-founded Refinement Rule Under Fairness Requirements.
Derive $(\mathcal{S} \rhd_F \mathcal{I})$ from $(\mathcal{S} \sqsupseteq_F \mathcal{I})$.

7 Examples

We now demonstrate the utility of the framework by describing the verification of a series of concurrent program examples. Our framework has been used to reason about

many concurrent programs, and the examples here are carefully selected to illustrate different facets of the framework. The first application is a concurrent deque implementation. In this example we discuss how stepwise refinements and stuttering are used to incrementally abstract implementation details and reduce verification complexity. The second application is cache coherence protocols, which illustrates the strength and weaknesses of the predicate abstraction procedure. The third example, a model of the Bakery algorithm, demonstrates the use of fairness. In Section 7.4, we briefly mention some of the other illustrative systems verified using our framework.

Given a specification $\mathcal{S}$ and an implementation $\mathcal{I}$ our proof methodology involves the following steps.

1. Construct intermediate models $\mathcal{I}_0, \ldots, \mathcal{I}_n$, with $\mathcal{I}_0 = \mathcal{I}$ and $\mathcal{I}_n = \mathcal{S}$ by successively eliminating implementation complexity.
2. Prove $(\mathcal{I}_{k+1} \rhd \mathcal{I}_k)$ (resp., $(\mathcal{I}_{k+1} \rhd_F \mathcal{I}_k)$ or $(\mathcal{I}_{k+1\ F} \rhd \mathcal{I}_k)$) for $k = 0, \ldots, n-1$. This involves one of the following two approaches.
   (a) Prove $(\mathcal{I}_{k+1} \trianglerighteq \mathcal{I}_k)$ (resp., $(\mathcal{I}_{k+1} \trianglerighteq_F \mathcal{I}_k)$ or $(\mathcal{I}_{k+1\ F} \trianglerighteq \mathcal{I}_k)$) using **Well-founded Refinement Rule** (resp., with fairness assumptions or requirements). Use the automated invariant proving procedure to discover and discharge the necessary invariants.
   (b) Prove $(\mathcal{I}_{k+1} \diamond \mathcal{I}_k)$ using the **Oblivious Refinement Rule**.
3. Chain the results from Step 2 using the **Stepwise Refinement Rule** to derive $(\mathcal{S} \rhd \mathcal{I})$.

Admittedly, there is still significant manual effort involved in the application of this methodology for practical concurrent systems. The key human components of the proof are in the development of rewrite rules and appropriate intermediate abstractions to support Step 2. In Section 5 we have discussed how the cost of developing rewrite rules is amortized by reusability. Unfortunately, intermediate models are system-specific. Defining them requires significant design insight to identify the relevant implementation features to be abstracted, and the effect of the abstraction on other features. The framework provides some automation to this process. For instance, local transitions within a single process with no update to externally visible components are automatically abstracted into single atomic transitions; some user interface and annotation mechanisms are also provided. However, for most systems in practice we have found that the development of intermediate models involves significant domain insights and creativity, as we will see for the concurrent deque implementation in Section 7.1. However, because the intermediate models can be defined in the same language as the implementation, we have found that their definition requires system design insights (rather than an understanding of logical formalism), that can be obtained from the designers. Indeed, this is a core reason for our choice of transition systems as a uniform specification and verification device. We will elaborate on this point in Section 9.

7.1 A Concurrent Deque Implementation

Our first example is a restricted concurrent deque implementation.[7] A *deque* is a data structure that allows insertion and deletion of items at either end. A *concurrent deque*

[7] The deque implementation was used by Sumners as an early case study in the course of development of the framework, and the result reported in the ACL2 workshop [70]; this paper contains detailed ACL2 models of the deque implementation and the intermediate models. The formalizations of the deque, as well as the intermediate models used in our framework, are the

```
void pushBottom (Item item)
 1   load localBot := bot
 2   store deq[localBot] := item
 3   localBot := localBot + 1
 4   store bot := localBot

Item popTop()
 1   load oAge := age
 2   load localBot := bot
 3   if localBot ≤ oAge.top
 4         return NIL
 5   load item := deq[oAge.top]
 6   nAge := oAge
 7   nAge.top := nAge.top + 1
 8   cas (age, oAge, nAge)
 9   if oAge = nAge
10         return item
11   return NIL
```

```
Item popBottom()
 1   load localBot := bot
 2   if localBot = 0
 3         return NIL
 4   localBot := localBot - 1
 5   store bot := localBot
 6   load item := deq[localBot]
 7   load oAge := age
 8   if localBot > oAge.top
 9         return item
10   store bot := 0
11   nAge.top := 0
12   nAge.tag := oAge.tag + 1
13   if localBot = oAge.top
14         cas (age, oAge, nAge)
15         if oAge = nAge
16               return item
17   store age := nAge
18   return NIL
```

Fig. 4 Methods for the Concurrent Deque Implementation. Line numbers represent values of the program counter. Variables `age`, `bot`, and `deq` are shared; all other variables are local. Variable `age` is a record containing two fields `age.top` and `age.tag`. The deque is represented by the portion of the array `deq` between `bot` and `age.top`. Instruction `cas` is "compare-and-swap": if the value of the (shared) variable `a` is equal to the value of `b`, `cas(a,b,c)` atomically swaps the values of `a` and `c`; it is a no-op otherwise.

allows arbitrary interleaving of executions of two or more processes that invoke the insertion or deletion operations. The deque implementation we analyze is due to Arora, Blumofe, and Plaxton [3], and used as part of the *work stealing algorithm* [6] for thread scheduling in multiprogrammed systems. The deque is restricted as follows.

1. Items are inserted only at the bottom of the deque.
2. A designated process called the *owner* inserts and removes items at the bottom of the deque; other processes (called *thieves*) only remove items from the top.

Arora *et al.* implement three methods `pushBottom`, `popBottom`, and `popTop` corresponding to insertion in the deque, deletion from the bottom, and deletion from the top respectively. The system is an asynchronous composition of processes invoking these methods. The owner process can invoke `pushBottom` and `popBottom`, while thieves can only invoke `popTop`. Fig. 4 shows the implementation of each method in pseudo-code. Concurrent execution of the methods by different processes, of course, can result in contentions and race conditions. Given the restrictions above, two thieves can contend to remove the same top item or a thief can contend with the owner when the deque contains a single item.

Although the methods together involve less than 40 lines of code, the implementation is quite complex. The complexity arises from the fact that the methods are

same as those reported by Sumners. However, the previous work did not use this framework, and did not have automation support (*e.g.*, for invariant discovery), verified reduction rules, or facility to generate formal models from the C implementation. Rather, a sequence of ACL2 models was defined manually and a chain of well-founded refinements proven. A key observation reported was the tediousness and difficulty of defining inductive invariants, which provided motivation to techniques for invariant discovery.

non-blocking: slow or preempted processes cannot prevent other processes from making progress. This property is crucial to the efficiency of the work stealing algorithm. Blumofe, Plaxton, and Ray [7] present a hand proof of the system. This proof involves consideration of a large number of cases, and many of the cases require subtle arguments. It therefore makes sense to use our framework to mechanically verify the implementation.

We will refer to our formal model of the implementation as the system **cdeq**. A state of **cdeq** is composed of (1) a vector of *process states* indexed by the corresponding process indices, and (2) a valuation of the shared variables `age`, `top`, and `deq`. A process state, in turn, is represented as a record consisting of the copy of each local variable and the program counter. At each instant, the external stimulus determines the index of the process making the next transition. A transition updates the local state of the transiting process (for example storing a new value in the program counter) together with possibly the shared variables. We formally model the updates by defining the effect of each instruction in Fig. 4 on each state component.

Our specification is an "abstract deque". We name the specification system **adeq**. In **adeq**, the deque is represented as a list and an invocation of a method performs the prescribed insertion and deletion atomically on this structure. Verification of **cdeq** then amounts to showing ($\mathbf{adeq} \triangleright \mathbf{cdeq}$). Here we highlight how the problem is decomposed by incrementally abstracting implementation details. A previous paper [70] gives an elaborate technical account of the proof.

The mechanical proof involves defining three intermediate models $\mathbf{icdeq}^+$, **icdeq**, and $\mathbf{cdeq}^+$ and showing the following chain of refinements.

$$(\mathbf{adeq} \trianglerighteq \mathbf{icdeq}^+ \diamond \mathbf{icdeq} \diamond \mathbf{cdeq}^+ \diamond \mathbf{cdeq})$$

The proof is then completed by the stepwise refinement rule **SR**.

It is illuminating to understand the motivations behind the definitions of these systems. The model **icdeq** is particularly illustrative. In **cdeq**, the deque is represented as an array `deq` whose two ends are specified by the shared variables `bottom` and `age.top`; an insertion or deletion of an item is reflected by incrementing or decrementing these variables. On the other hand **adeq** represents the deque as a list. This "abstraction gap" is bridged by the system **icdeq**. Here we use the list representation of the deque as in **adeq**. But unlike **adeq** the owner and thief transitions are more fine-grained, which makes the correspondence between **icdeq** and **cdeq** tractable. In particular, while in **adeq** the invocation of each method is atomic, in **icdeq** we only "collapse" local steps of a process. For instance the transitions of a thief (executing `popTop`) in **cdeq** passing through program counter values $\mathtt{6} \to \mathtt{7} \to \mathtt{8}$ do not change the shared variables; in **icdeq** this sequence is replaced with a single transition in which a thief transits directly from program counter value `6` to `8`. Because the notion of correspondence is insensitive to finite stuttering it is trivial to relate the executions of a concrete system with those of an abstract system in which the local steps of the processes are collapsed. To show such correspondence using well-founded refinements we define *skip* to hold along a transition of the concrete system in which the transiting process is poised to take a local step. The function *rank* at state s is then defined as follows. We count for each process index i the number of local steps l_i that i must take from s before taking a "visible" step. For instance in **cdeq**, if i is a thief with program counter value 6, then l_i is equal to 2. We then define $\mathit{rank}(s) = \Sigma_i l_i$, where the sum is over all process indices. Thus *rank* is a natural number (and hence an ordinal), and it decreases every time a process takes a local step, satisfying **SST4**.

A second goal in the definition of **icdeq** is to simplify the role of the shared variables. In particular, consider the variable `age` in Fig. 4. It has two components `age.top` and `age.tag`. The component `age.top`, together with `bottom`, determines the deque boundary. To understand the role of `age.tag`, consider lines 11-14 of the `popBottom` method. This corresponds to the actions of the owner successfully removing the last item from a deque or finding the deque empty. The owner then resets `bottom` and `age.top` to `0`. However, to ensure that a thief that might have read a "stale" `top` value of `0` earlier does not attempt to remove an item from the empty deque after reset, the value of `age.tag` is incremented. This value is monotonically increasing and therefore would not match with the value that the thief would have read.

Since the role of `age` in **cdeq** is so complex, it is preferable to abstract it in **icdeq**. We do so by replacing `age` with a counter that is incremented every time a process (owner or thief) removes an item from the top of the deque. The monotonicity of the counter can then be used by the thief to determine if its local copy of `top` is stale.

The endeavor to simplify `age` necessitates the definition of the model $\mathbf{cdeq}^+$. Recall that to prove ($\mathbf{icdeq} \trianglerighteq \mathbf{cdeq}$) we must construct a mapping *rep* from the states of **cdeq** to the states of **icdeq**. Unfortunately, because `age` is updated by different processes it is difficult to determine a consistent value of the counter for this mapping from a state of **cdeq**.

We solve this ambiguity by defining the system $\mathbf{cdeq}^+$, which is merely an augmentation of **cdeq** with auxiliary variables. In particular, we store the counter as an explicit state component of $\mathbf{cdeq}^+$ that is updated by different processes simultaneously while updating `age`. By oblivious refinements, the proof of ($\mathbf{cdeq}^+ \diamond \mathbf{cdeq}$) is automatic. Furthermore, because the counter is explicit in $\mathbf{cdeq}^+$, the mapping from the states of $\mathbf{cdeq}^+$ to those of **icdeq** is trivial.

The descriptions above elucidate a general principle behind defining intermediate systems for decomposing verification. The **icdeq** system was defined with the explicit objective of hiding some specific implementation details, namely the complexity of `age`, the representation of the deque, and collapsing of the local process transitions. In general, verification of a complex system requires that we decompose the verification into a chain of refinements with every "link" representing some specific details to be hidden. Exhibiting the correspondence between consecutive models in the chain requires augmentation of the more "concrete" model with auxiliary variables tracking the history of the execution, in this case the counter in $\mathbf{cdeq}^+$ tracking the removal of the top item.

7.2 Cache Coherence Protocols

Our second application of the framework involves cache coherence protocols. These protocols maintain data consistency between caches of distributed shared memory systems. Developing effective reasoning techniques for cache coherence protocols is a topic of significant research interest in the formal verification community [64, 22, 42, 41]. Verification of these protocols using our framework provides some interesting insights on reusability and scalability of the automated invariant proving component of our framework. To illustrate the insights we will discuss the verification of two protocols; first, a simple **ESI** protocol and then, an elaborated version of the German protocol.

Remark 7 The use of our invariant prover on the **ESI** and German protocol models discussed here was reported in an earlier paper [65]; that paper covers the details

of the implementation of the invariant prover, and discusses the role of the different optimizations on the invariant proof of these systems. Thus we omit detailed description of their models and refer the interested reader to the previous paper. The focus of the original paper was on the invariant prover alone with the relevant property defined as a predicate named *coherent* to be proven as an invariant. Here we cast these protocols into the refinement framework, which proves that they are indeed refinements of a simple memory system. Note that the predicate *good* we prove as invariant below, is not equivalent to the *coherent* predicate mentioned above. The *coherent* predicate was defined by a state machine that tracks the content of an arbitrary location in the cache through the different system transtions; the predicate then says that a valid location contains the last value written. The predicate *good* defined below merely says that the valid data in the cache lines for the protocol must be the same as the valid data in the central memory of the specification system. The difference in the definition of the two putative invariants (*viz.*, *good* below and *coherent* in the previous work) arise from the difference in goals. In previous work, coherence was formalized in the standard way without concern for how (or whether) that permits relating the implementation to a memory system specification. In the current work, *good* was defined specifically with the latter goal. Indeed, in proving the invariance of *good* in the current work, the invariant prover *generates* the predicates to track the most recent value written to a memory location as specified manually by *coherent*. Thus the resulting abstraction graph is the same as in previous work and is enabled by the same set of (both generic and domain-specific) rewrite rules. However, to keep the paper self-contained we include the discussion of the use of the invariant prover on *good* because it illustrates the key usage model of the invariant prover, and demonstrates both its strength (*viz.*, robustness and scalability) and weakness (*viz.*, requirement of manual introduction of domain-specific rewrite rules).

In the **ESI** protocol, a number of client processes communicate with a single controller process to access cache lines. Cache lines consist of addressable data. A client can read the data from an address if its cache contains the corresponding line. A client acquires a cache line by sending a *fill* request to the controller; requests are tagged for **E**xclusive or **S**hared access. A client with shared access can only read the data in the cache line. A client with exclusive access can also write data. The controller can request a client to **I**nvalidate a cache line and if the line was exclusive then its contents are copied back to memory.

Our specification for cache coherence protocols is a simple memory system that we call **memory**. It has a single array `mem` that is updated atomically at each write. We call our model of the **ESI** protocol the system **esi**. A state of **esi** consists of the following components.[8]

- A 1-dimensional array called `mem` that is indexed by cache lines. For any cache line `c`, `mem[c]` is a finite mapping of addresses in `c` to data values.
- A 1-dimensional array `valid`. For each cache line `c`, `valid[c]` contains a set of process indices that have a copy of `c` in their local caches.
- A 1-dimensional array `excl`. For each cache line `c`, `excl[c]` contains a set of process indices that have **E**xclusive access to `c`.

[8] The protocol definition was automatically translated from an existing Verilog implementation.

$$\mathit{in}(e, \mathit{insert}(a, s)) = \mathit{in}(e, s) \vee (a = e)$$
$$\mathit{in}(e, \mathit{drop}(a, s)) = \mathit{in}(e, s) \wedge (a \neq e)$$
$$\mathit{get}(a, \mathit{set}(b, v, r)) = \begin{cases} v & \text{if } a = b \\ \mathit{get}(a, r) & \text{otherwise} \end{cases}$$

Fig. 5 Rewrite rules for using predicate abstraction on **esi** system. Here $\mathit{in}(e, s)$ holds if element e is a member of set s, $\mathit{insert}(e, s)$ inserts e in set s, $\mathit{drop}(e, s)$ removes e from s, $\mathit{get}(a, r)$ returns the value of field a in record r, and $\mathit{set}(a, v, r)$ returns the value of record obtained by setting the value of field a to v in record r.

– A 2-dimensional array `cache`, which is indexed by process index and cache line. For any process index `p` and any cache line `c`, `cache[p][c]` returns the local copy of the contents of cache line `c` in the cache of `p`.

We verify **esi** by showing (**memory** $\trianglerighteq$ **esi**). We do not need stuttering, so we define $\mathit{skip}(s, i) =$ `NIL`. Thus the obligations **SST2**, **SST4**, and **SST5** are vacuous for this example. We also define $\mathit{pick}(s, i) = i$. The representative function *rep* is defined as follows.

– For each cache line `c`, if `valid[c]` is empty in s then `mem[c]` in $\mathit{rep}(s)$ is the same as `mem[c]` in s. Otherwise, let p be an arbitrary but fixed process in `valid[c]`. Then `mem[c]` in $\mathit{rep}(s)$ is the same as `cache[p][c]`.

Our *good* predicate formalizes cache coherence. We do this as follows.

– Let s' be $\mathit{rep}(s)$. Then s satisfies *good* if and only if for any cache line `c` and any process p in s if `valid[c]` contains p, `cache[p][c]` in s is the same as `mem[c]` in s', otherwise `mem[c]` in s is the same as `mem[c]` in s'.

With these definitions it is easy to prove conditions **SST1** and **SST3** (the two non-trivial obligations for (**memory** $\trianglerighteq$ **esi**) other than the invariance of *good*). For instance to prove **SST3** note that the update to the `cache` occurs in **esi** when the stimulus i specifies the operation `"store"`, and the system **memory** matches this transition by making the corresponding update to the `mem` array. We can therefore focus our attention to the proof that *good* is an invariant.

Although it is not difficult to define an inductive invariant for this simple system, it is nevertheless instructive to instead prove the invariance of *good* directly using the tool we described in Section 5. Since the system is modeled using sets and records, we use a generic library of lemmas for normalizing terms built out of operations on these data structures. Fig. 5 shows some of the rules from this generic library. To use our tool for proving the invariance of *good*, we need the following rule in addition to those in the library.

Customized Rewrite Rule.

$$\begin{aligned} \mathit{in1}(e, s) = \; & \mathit{if}\ \mathit{empty}(s)\ \mathit{then}\ \texttt{NIL} \\ & \mathit{else\ if}\ \mathit{singleton}(s)\ \mathit{then}\ (e = \mathit{choose}(s)) \\ & \quad \mathit{else}\ \mathit{hide}(\mathit{in1}(e, s)) \end{aligned}$$

The function *in1* is the same as the set membership function *in*, but applied to sets that are expected to be either empty or singleton. The function *choose* is the choice function constrained to return some element of a set s if s is non-empty. In the logic *hide* is defined as $\mathit{hide}(x) = x$. But it is given special treatment by our tool for supporting user-guided abstraction: a term containing *hide* in any of its subterms is turned into a

free input during predicate abstraction. The rule above is illustrative of how protocol-level intuitions are made explicit using rewrite rules for effective predicate abstraction without restricting language expressiveness. In particular, the user conveys the intuition that `excl[c]` for any cache line `c` is always empty or a singleton by using *in1* to "tag" the membership tests on this component. Then the rule above causes terms involving such tests to be rewritten introducing a case-split for cases where the set is empty, singleton, or otherwise, and abstracts the third (presumed) irrelevant case. With the rule, our tool proves the invariance of *good* by generating an abstraction graph on 9 state predicates and 25 input predicates.

The reader may wonder if the manual effort required is unrealistically large. Note that in order for the customized rewrite rule with *in1* to be useful the user has to adjust the model, using *in1* in place of *in* for checking membership of sets that are expected to be empty or singleton.[9] However, the effort is amortized by reusability of the rules on similar systems. To illustrate this point, we consider the verification of a cache coherence protocol by Steven German, that can be viewed as a realistic implementation of **esi**. In this protocol, the communication between the clients and the home process occurs via three channels as follows. Clients send requests for cache lines in channel 1. Channel 2 is used by home to grant clients access to cache lines and send invalidate requests. Channel 3 carries the invalidate acknowledgments. Our model is derived from the UCLID model of the protocol by Lahiri and Bryant [41] with unbounded FIFOs for channels, and additionally contains datapaths and memory.

We prove the same property about this protocol that we proved for **esi**, *viz.*, that it is a (well-founded) refinement of **memory**. Note that an inductive invariant for **german**, if defined manually, would be significantly more complex than that for **esi**.[10] But the additional manual effort necessary for this verification over **esi** is small; most of the definitions and concepts can be reused. In particular, the definitions of *rep* and *good* do not change. Furthermore, the customized rule replacing *in* with *in1* carries over to this system. Given these rules the tool can prove the invariance of *good* in less than 5 minutes on a standard desktop machine running Linux by computing an abstraction graph over 46 state predicates.

7.3 An Implementation of the Bakery Algorithm

Our third example is an implementation of the Bakery algorithm [43]. The Bakery algorithm is one of the most celebrated solutions of the mutual exclusion problem in multiprocess systems. In this algorithm, each process has two local variables, namely a Boolean variable `choosing` and an integer variable `pos`. A process expresses interest to enter the critical section by setting its local copy of `choosing` to `T`. Each interested process is allotted a number that is stored in the `pos` variable; the number allotted to

[9] The tool provides some automated support. For instance, the user can define a model $\mathcal{M}$ where set memberships are checked with *in* but annotate it marking sets that can be empty or singleton. The tool then automatically generates a new model $\mathcal{M}'$ in which membership tests for the corresponding sets use *in1* and proves $(\mathcal{M} \diamond \mathcal{M}')$. In general for each function f in the current theory, the tool maintains a user-extendible set of alternative functions $f_1, \ldots, f_k$ that are provably equivalent to f. Furthermore, for each f_i the tool maintains a set of annotations that trigger generation of intermediate models replacing f with f_i.

[10] Out of curiosity, the authors performed the exercise of manually defining the inductive invariants. While the inductive invariant for **esi** is trivial, that for **german** took a couple of days.

```
Procedure Bakery
 1    choosing := T
 2    temp := max
 3    pos := temp + 1
 4    cas(max, temp, pos)
 5    choosing := nil
 6    indices := keys(procs)
 7    if indices = nil
            goto 11
      else
            current := indices.first
      endif
 8    curr := procs[current]
 9    if choosing[curr] == T
            goto 9
10    if (pos[curr] ≠ nil) and
            ((pos[curr], current) <_l (pos[p],j))
            goto 10
      else
            indices := indices.rest; go to 7
11    ⟨ critical section ⟩
12    pos := nil
13    ⟨ non-critical section ⟩
14    goto 1
```

Fig. 6 The Bakery Program Executed by Process p with Index `j`. The numbers to the left of the program instructions are the pc values. The function *keys*(`procs`) returns the list of indices of the processes in `procs`. The relation $<_l$ denotes "lexicographic less than": Given numbers a, b, c, and d, and, $(a, c) <_l (b, d)$ holds if either $a < b$ or $a = b$ and $c < d$.

a process is higher than all the currently allotted numbers. Processes enter the critical section in ascending order of `pos` values. Since executions of different processes are overlapped, more than one process can have the same `pos`; ties are broken by giving priority to the process with the lower index.

We formally model this algorithm as an asynchronous system analogous to the deque system above, with each state comprised of the vector of local process states and the value of the shared variables. We call this system **bakery**. Fig. 6 shows the actions of each process in the **bakery** system. Our model is inspired by a microarchitectural implementation and optimized in some aspects. In particular, we optimize the allotment of `pos` to a process by using the shared variable `max` to keep track of the maximum number already allotted; `max` is updated using the compare-and-swap instruction in line 4.

Our specification **bspec** is an asynchronous system defined as follows. Each process is a state machine that passes sequentially through states `"idle"`, `"wait"` and `"critical"`; a process transits from `"wait"` to `"critical"` state only if no other process is in the `"critical"` state.

Verification of **bakery** now entails showing correspondence between the executions of **bakery** with those of **bspec**. It is worth noting that exhibiting such correspondence amounts to proving both the safety (mutual exclusion) and progress properties of the system. The progress property for **bakery** can be informally stated as follows. If there are processes waiting for access to the critical section in a state in which no process is actually executing the critical section, then some process is eventually granted access.

This requirement is imposed since in **bspec** a process transits from the `"wait"` state to the `"critical"` state atomically and our notion of correspondence restricts stuttering to be finite.

Unfortunately there are executions of the **bakery** system that do not satisfy the progress property. Consider the following scenario. Assume that some process i sets its local copy of `choosing` to `T` and subsequently never makes a transition. Another process j then "wants" access to the critical section, obtains a `pos`, and reaches the program point given by the program counter value `9`. At this point, it waits for every other process that had already set their `choosing` to pick their `pos` and reset `choosing`. Thus in this scenario, process j indefinitely waits for i and cannot proceed, even if no other process is in the critical section. Indeed, as long as process i does not make a transition, *no* subsequent process can proceed to the critical section.

Thus it is not possible to prove that **bakery** is a refinement of **bspec** with no assumption on how the processes are scheduled. Informally speaking, the **bakery** system ensures that waiting processes enter the critical section under the implicit assumption that each participating process is eventually scheduled to make progress. It therefore makes sense to exhibit the desired refinement under fairness assumption, that is, to prove ($\mathbf{bspec} \rhd_F \mathbf{bakery}$).

Our verification entails showing the following chain of refinements:

$$(\mathbf{bspec} \trianglerighteq_F \mathbf{bakery}^+ \diamond \mathbf{bakery})$$

Here we define the system $\mathbf{bakery}^+$ by augmenting **bakery** with auxiliary variables analogous to the way we augmented **cdeq** in the previous example. In particular, for any state s, one auxiliary variable records a list l of processes waiting to be granted access to the critical section; the order of the processes in the list coincides with the order in which they will be granted critical section access in the **bakery** system.

The interesting component of the verification is the definition of *rank* necessary to prove ($\mathbf{bspec} \trianglerighteq_F \mathbf{bakery}^+$). Our definition essentially involves a lexicographic product of the following natural numbers. Here p is assumed to be the process at the head of the list l.

1. The program counter of process p.
2. Fairness measure on process p.
3. If p is poised to execute line 9, that is, check if some other process q (indexed by the variable `curr`) has its local copy of `choosing` set, then the fairness measure of q, else 0.

Lexicographic products of natural numbers can be easily mapped to ordinals, and ACL2 can mechanically perform this mapping [52]. Notice that the use of fairness "resolves" the problem with the scenario we discussed above. If process p waits for q, then as long as q is not selected to make a transition, the fairness measure of q must decrease; eventual progress of q (and hence p) is therefore guaranteed by well-foundedness.

7.4 Other Systems

The examples above demonstrate how stuttering refinement can be used as a notion of correspondence to define intuitive specifications for different concurrent programs,

how reduction theorems help decompose the verification problem, and how predicate abstractions can be used in the framework to effectively automate proofs of necessary invariants. We have used our formalizations to specify and verify several concurrent program implementations. We now briefly mention some additional systems that we have verified using this framework.

Leader Election Protocol

Leader election is a classic problem in distributed systems. The goal is for a collection of processes to communicate with each other to determine the identity of the process with the lowest index. The process with the lowest index is then selected as the *leader* or *arbiter* and used to resolve different resource contentions. Various solutions to the problem appear in textbooks on distributed algorithms [47]. We verified a standard but low-level synchronous implementation of a leader election protocol on a *token ring*. A process non-deterministically initiates the protocol by sending a token to its neighbors in the ring, and each process subsequently alternates between receiving and passing the token, with a finite number of local steps in between, until all processes reach a consensus on the identity of the leader. Our specification is a simple abstract system with two processes in which the leader is selected atomically in one transition after initiation.

We use well-founded refinements to prove that the token ring implementation is a refinement of the specification. For a ring state s, the representative function *rep* "projects" the initiator process and the (eventual) leader, *skip* holds along the transitions in which the token is in transit, and the *rank* function is a lexicographic pair of natural numbers specifying (1) the distance of the token from the process with the lowest index, and (2) the sum, over all process indices i, of the number of local steps of process i before the next token transfer action. Note that a token ring protocol for a ring of an arbitrary size can be shown to be stuttering bisimilar to one on a ring of size 2 [23,60]. As a bonus, our proof provides a mechanical derivation of this result for the leader election protocol.

The Apprentice System

The Apprentice system, developed by Moore and Porter [59], is a JVM implementation of a multithreaded Java program. The program is formalized in ACL2 using an operational model of the JVM [58], developed at the University of Texas. The JVM model, called M5, specifies the operational semantics of about 138 JVM instructions, and supports features like method resolution, invocation of static, special, and virtual methods, and synchronization via monitors. In the Apprentice program, each thread executes an infinite loop that involves (1) spawning a new thread, and (2) contending with the existing threads to increment a single shared counter variable allocated in the heap.

Moore and Porter put forward the Apprentice program as a benchmark against which to measure approaches to formally proving properties of Java programs. They also prove, using ACL2, that the program satisfies the *weak monotonicity property*, that is, the counter never decreases along any transition. Note that since the program involves concurrent reads and writes to a shared variable by several threads, weak monotonicity is a non-trivial property to verify. Nevertheless, their proof does not

ensure progress, that is, the counter eventually increases. Indeed, if a thread is never scheduled after spawning a child then the counter does not increase.

The progress property of the Apprentice can be verified in our framework under the assumption that the spawned threads are scheduled fairly. In particular, under fairness, a thread must eventually proceed executing the rest of the instructions in the loop after spawning a new child. Using this progress property, it is now possible to prove that the Apprentice program is a refinement (under fairness assumption) of a machine that atomically increments the counter at each transition.

A Dining Philosopher's Problem

The Dining Philosopher's Problem [20], suggested by Dijkstra in 1965, is a classic problem in distributed computing. In this problem, a set of philosophers sitting at a round table alternate between states `thinking` and `eating`. A philosopher can non-deterministically decide to eat at any time. A philospher can eat only if both his left and right forks are available (*i.e.*, not used by an adjacent philosopher for eating). The implementation of the problem we analyze arose in the resource allocation algorithm of the chipset protocol of an industrial microprocessor design. The algorithm is roughly based on the message-passing solution by Chandy and Misra [15], is optimized for efficiency, and is designed to work with an unbounded number of philosophers (resource contenders). The implementation involves about 600 lines of Verilog code.

We prove that the implementation is a refinement up to stuttering of a simple specification system, under fairness assumption. The specification is the obvious system in which each philosopher atomically accesses both forks and finishes the eating in one transition. The "eating" here represents access to an appropriate peripheral device. Note that fairness is important to make sure that each philosopher actually completes eating; without fairness, a philosopher in eating state may never be selected again to complete the operation (and hence release his forks).

A Distributed Checkpointing Protocol

Our last example is the use of the framework in the verification of a distributed checkpointing algorithm. Checkpoint and restart (CPR) constitute the most commonly used approach to developing fault-tolerant distributed systems. The approach entails periodically saving (checkpointing) the state of the computation to stable storage; when a fault is detected, the computation is restarted (rolled back) at the last saved checkpoint. The Chandy-Lamport algorithm [14] is a classical distributed algorithm that forms the basis of a wide number of system-level checkpointing schemes. It enables a process in a distributed system to determine a global snapshot of the system during a computation. Note that the problem is non-trivial because processes may not share clocks or memory: in fact, a process can only record its own local state and the messages it sends or receives; it has no further global view of the computation. The correctness theorem of the algorithm, as specified in the original paper [14], characterizes certain properties of the state recorded by the protocol. The theorem proven about the recorded state (cf. Theorem 1 [14, §4]) may be paraphrased as follows.

Correctness Theorem.
Let S_0 be the global state in which the snapshot algorithm is initiated, and let S_1 be the global state in which it terminates. Let S^* be the global state recorded

by the algorithm then (1) S^* is reachable from S_0, and (2) S_1 is reachable from S^*.

We analyze an implementation of a distributed checkpointing protocol that makes use of a variant of the Chandy-Lamport algorithm. The protocol we analyze is a simplified version of the application-level checkpointing system implemented by Bronevetsky and Pingali [12]. We show that this system is a refinement up to stuttering of a simple specification system that performs the same computation but does not do any checkpointing and does not roll back. Note that our problem (and consequently, our proof) is more elaborate than "merely" verifying the Chandy-Lamport algorithm. In particular, Chandy and Lamport's paper only covers how each process records *its* state and the state of its incoming channel; even *state construction*, *viz.*, assembling this local information into a complete global state, were left unspecified. However, we need to model state construction, storage of snapshots, failure detection and recovery, replay of executions from snapshots, etc. The problem becomes challenging because the global state actually constructed from the local states recorded by each process may not correspond to a state that has been actually encountered in the execution. We handle this complexity by using an intermediate model that contains an auxiliary variable explicitly constructing the global state based on the current recording of the local states by individual processes; this variable is updated every time a process records a local (or channel) state. Since this variable is only updated and never read by the implementation, correspondence between the intermediate model and the implementation follows by oblivious refinement. We connect the intermediate model with the specification using the single-step, well-founded refinement rule, making use of the state recorded in this auxiliary variable to define the representative function *rep*; the proof obligation then reduces to proving the above **Correctness Theorem** for the process of recording snapshots.

8 Related Work

The specification and verification of reactive concurrent programs in a formal logic are areas of extensive research. Two expressive logics, Unity [55,56] and TLA [45] have been designed specifically for describing properties of such programs. Both model checking and theorem proving approaches have been extensively used for concurrent program verification. Among model checking approaches are the verification of TLA+ specifications of cache coherence protocols using the TLC model checker [46], and use of the *Java Pathfinder* for checking deadlocks and assertions in Java programs [74]. Model checking has also been extensively used for the verification of propositional temporal logic properties of finite state concurrent implementations. The literature on the subject is vast, with several excellent surveys [28,39]. There has also been significant research on theorem proving techniques to verify concurrent programs. Examples of such efforts include verification of distributed garbage collector algorithms [33,67, 21], formalization and verification of network and communication protocols [75,29], fault-tolerant systems [69], real-time concurrent systems [32], and multithreaded JVM bytecodes [59]. Recently a proof system has been developed for TLA+ specifications, and used in the verification of safety properties of concurrent programs [17].

Lamport [44] argues that specifications should be invariant under stuttering. Abadi and Lamport [1] use *refinement maps* to reduce proofs of stuttering trace containment

and trace equivalence to reasoning about single steps of programs, and prove a *completeness theorem* stipulating conditions for the existence of refinement maps. Our notion of correspondence is a formalization of a variant of Abadi and Lamport's notion in ACL2. Several researchers [24,31,4,34] have extended Abadi and Lamport's proof rules. In addition to the linear-time notions of trace containment and trace equivalence, analogous notions of correspondence for *branching time*, namely *simulation* and *bisimulation* [54,62], have been studied under stuttering. While in linear time an execution of a program is modeled as an infinite *sequence* of states, in branching time it is modeled as an infinite *tree*. Namjoshi [60] presents sound and complete proof rules for symmetric stuttering bisimulation. Manolios et al. [51] define a related notion, *well-founded equivalence bisimulations* (or WEBs), and use it to verify the Alternating Bit Protocol. Manolios [49] also provides sound and complete proof rules for stuttering simulation. Indeed, our single-step rules for well-founded refinements are influenced by Manolios' proof rules, although we prefer the stuttering trace containment rather than stuttering simulation as the formal notion of correspondence in the framework.

Refinement techniques are part of almost all model checking and program verification tools [18,53,10,5]. Most refinement frameworks do not include stuttering. An exception is the Atelier B toolkit (`http://www.atelierb.eu/index-en.php`) that includes rules for reasoning about stuttering. It also includes a mechanized reasoning tool, *viz.*, the B theorem prover. A more recent version of the B method called Event-B (`http://www.event-b.org`), supported by the Rodin tool set, has been used to develop a number of concurrent algorithms [2]. However, stuttering is not explicitly formalized in these tools; instead, the tool generates verification conditions that ensure stuttering refinement. Our approach is different in that the notion of stuttering trace containment is formalized in the logic of a general-purpose theorem prover and the reduction rules applied are verified by the theorem prover, permitting sound extension of the repertoire of proof rules.

Finally there has been work on formalizing notions of correctness of reactive systems by semantically embedding temporal logics in a theorem prover. Goldshlag [25] formalizes the proof rules of Unity in the Nqthm theorem prover, and uses it to verify several concurrent programs. Paulson [63] formalizes the Unity proof rules in Isabelle. Hooman [40] develops semantics of timed communicating reactive systems in PVS.

9 Discussion

The development of our framework involved three key design decisions:

1. using a general-purpose theorem prover as the underlying reasoning engine;
2. using program correspondence instead of temporal logic for specifying program correctness; and
3. including a provision for stuttering in the notion of correspondence.

The choice of a general-purpose theorem prover is governed by the need for flexibility and control. A theorem prover facilitates clean decomposition of proofs, and sound extension and careful orchestration of strategies. Indeed, many of our proof rules were crafted on demand; when the available rules were found insufficient, they were restructured or new ones were added. Furthermore, as we discussed with predicate abstractions, decision procedures can be easily integrated to effectively automate expensive proof steps. Indeed, they become more effective in this context because the invariant

prover can use user-proven lemmas to control the complexity of the abstraction on which model checking is applied.

There are two broad approaches to defining specifications of concurrent programs. One approach is to define the desired (safety and liveness) properties as formulas in a temporal logic. The second is to define an abstract system with which to relate the executions of the implementation. Of course it is well-known that if $\mathcal{S}$ preserves all executions of $\mathcal{I}$ up to stuttering then any LTL\X property of $\mathcal{S}$ can be inferred for $\mathcal{I}$ (under appropriate definition of atomic propositions and state labels of the two systems) [18]. Temporal logic specifications have some attractive advantages (namely permitting description of safety and liveness properties directly as logical formula). However, for designing a deductive verification framework, we found it suitable to define specifications in terms of executions of a simpler system for a number of reasons. First, most general-purpose theorem provers (*e.g.*, ACL2, HOL, PVS) support *classical logic*; a semantic embedding of temporal logic in such a theorem prover is non-trivial; the problem is exacerbated in a first-order theorem prover like ACL2. Second, most of our target programs are parameterized models with an unbounded number of processes; temporal logic specification of such programs requires an expressive logic (*e.g.*, allowing quantification of process indices), and the meaning of the resulting specification can be error-prone due to possible nesting of quantifiers over time, branching, and design parameters. Third, informal human review of the completeness and correctness of temporal specification requires familiarity in formal logic in general and temporal logic specifically. Specification by program refinement allows the specification and implementation to be defined in the same operational language. This helps avoid errors that may otherwise be missed due to differences in semantics between the specification and the implementation. Indeed, a similar design decision to base correctness specifications and proofs on trace containment (modulo stuttering) also underlies the (Event-)B and TLA+ formalisms.

The decision to allow for stuttering in the notion of correspondence stems from our desire to provide simple, intuitive specification. Admittedly, defining the theory of stuttering trace containment in ACL2 was complex. However, that was only performed once. For each application discussed in Section 7, the specification system was the obvious, intuitive abstraction capturing the design intent of the protocol. This is no coincidence. In practice, concurrent programs are often optimized elaborations of simpler protocols. These elaborations are designed to achieve execution efficiency, refine atomicity, match a given architecture, and so on. The simpler protocol then provides a succinct operational description of the intended behaviors of the elaborated implementation. Stuttering is used to reconcile the difference in the level of abstraction at which the implementation and specification are modeled, while limiting the amount of stuttering to be finite ensures that both safety and progress properties of the implementation are preserved in the specification. It is also worth noting that even when the specifications are defined in terms of temporal logic properties, it is common to define abstract models that hide details of the implementation and facilitate the verification process. In practice, a notion of correspondence with finite stuttering allows us to use refinements both as a specification and a proof mechanism.

10 Summary and Conclusion

We have presented a deductive framework for uniform specification and verification of reactive concurrent programs. Verification of a concurrent program implementation in this framework entails defining an abstract specification system and showing that the implementation is a refinement of the specification up to finite stuttering. The framework is formalized using the ACL2 theorem prover, and provides a uniform approach for specification and verification of both safety and progress properties of reactive systems without requiring semantic embedding of temporal logics. The framework facilitates design of intuitive specifications for a diverse range of concurrent programs. We have designed methodologies and tools for decomposing and automating refinement proofs for reasoning about reactive concurrent programs.

Admittedly, the limited expressive power of the ACL2 logic makes the mechanization of the framework difficult. In our work, the limitations are manifest principally in the formalization of the theory of trace containment, and verification of the reduction rules. Once this has been done, verification of individual programs reduces to a first-order problem for which ACL2 is well-suited. In addition, the theorem prover provides several features such as the use of an ANSI-standard programming language as the formal language for modeling and specification of systems, support for efficient simulation of formal models, and the capability of handling large formulas, that make the framework robust and scalable for application to practical systems. Nevertheless, we consider it important for the framework to provide more support for reasoning about infinitary objects. We are contemplating the possibility of exploiting a recent connection between the ACL2 and HOL4 theorem provers [26] to provide such support. Using that connection, the formalization of stuttering trace containment and verification of reduction rules will be transferred to HOL's more expressive logic, while verification of individual programs will still benefit from ACL2's automation and executability as well as the tools we have already defined in ACL2. However, this requires the linkage to be robust to permit smooth and seamless transfer of verification collateral between the two theorem provers.

Our planned future work on extending the framework involves two key research thrusts: making the framework more robust and extensible, and using it for more diverse systems. Towards the first goal, we are augmenting the framework with more reduction theorems and designing a better user interface to facilitate practical applications. One key planned augmentation is support for *real-time constraints*. Note that fairness constraints only ensure that a fair stimulus is eventually picked. Real-time systems require more stringent constraints, *e.g.*, that the stimulus is picked within a fixed upper bound of time; formalizing this within the refinement framework is challenging and requires careful thought about effective reduction rules for reasoning about the notion of a monotonically increasing time. Towards the second goal, we are using the framework for reasoning about communication protocols in microarchitecture implementations, and distributed garbage collection algorithms.

11 Acknowledgments

The research of the first author is supported in part by the National Science Foundation under Grants CNS-0910913 and CCF-0916772, and by the Defense Advanced Research Projects Agency under Grant N66001-10-2-4087. The authors thank the UT Austin

Automated Theorem Proving group, in particular, Matt Kaufmann, Robert Krug, and J Strother Moore for comments and insights. Additionally, Krug and Moore read an earlier draft of the paper and made several useful suggestions. The verification of the distributed checkpointing protocol mentioned in the paper came out of discussion with Keshav Pingali and the UT Austin Programming Language research group.

References

1. M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling in Multiprogramming Multiprocessors. *Theory of Computing Systems*, 34:115–144, 2001.
4. P. Attie. Liveness-preserving Simulation Relations. In J. Welch, editor, *Proceedings of 18th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 63–72. ACM Press, May 1999.
5. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122. Springer-Verlag, 2001.
6. R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, 1999.
7. R. D. Blumofe, C. G. Plaxton, and S. Ray. Verification of a Concurrent Deque Implementation. Technical Report TR-99-11, Department of Computer Sciences, The University of Texas at Austin, June 1999.
8. R. S. Boyer, D. Goldshlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
9. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.
10. R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV 1996)*, volume 1102 of *LNCS*, pages 428–432. Springer-Verlag, July 1996.
11. B. Brock and W. A. Hunt, Jr. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. In *Proceedings of the* 1997 *International Conference on Computer Design: VLSI in Computers & Processors (ICCD 1997)*, pages 31–36, Austin, TX, 1997. IEEE Computer Society Press.
12. G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level Checkpointing for Shared Memory Programs. In S. Mukherjee and K. McKinley, editors, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 235–247. ACM, October 2004.
13. G. Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Dover Publications Inc., 1952. Translated by P. E. B. Jourdain.
14. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
15. K. M. Chandy and J. Misra. The Drinking Philosopher's Problem. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 6(4):632–646, October 1984.
16. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Cambridge, MA, 1990.
17. K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying Safety Properties with the TLA+ Proof System. In J. Giesl and R. Hähnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR 2010)*, number 6173 in LNCS, pages 142–148. Springer, July 2010.

18. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model-Checking*. The MIT Press, Cambridge, MA, January 2000.
19. J. Davis. Finite Set Theory based on Fully Ordered Lists. In M. Kaufmann and J S. Moore, editors, 5*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, 2004.
20. E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, 1971.
21. D. Doligez and G. Gonthier. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In 21*st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1994)*, pages 70–83. ACM Press, January 1994.
22. E. A. Emerson and V. Kahlon. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In D. Geist, editor, *Proceedings of the* 12*th International Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *LNCS*, pages 247–262. Springer-Verlag, July 2003.
23. E. A. Emerson and K. Namjoshi. Reasoning about Rings. In M. Ernst, editor, *Proceedings of the* 22*nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*, pages 84–95. ACM Press, 1995.
24. K. Engelhardt and W. P. de Roever. Generalizing Abadi & Lamport's Method to Solve a Problem Posed by Pnueli. In J. Woodcock and P. G. Larsen, editors, *Industrial-strength Formal Methods,* 1*st International Symposium of Formal Methods Europe*, volume 670 of *LNCS*, pages 294–313, Odense, Denmark, April 1993. Springer-Verlag.
25. D. M. Goldshlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16(9):1005–1023, 1990.
26. M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An Integration of HOL and ACL2. In A. Gupta and P. Manolios, editors, *Proceedings on the* 6*th International Conference on Formal Methods in Computer-Aided Design (FMCAD-2006)*, pages 153–160. IEEE Computer Society Press, 2006.
27. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In O. Grumberg, editor, *Proceedings of the* 9*th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, 1997.
28. A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in Systems Design*, 2(3):151–238, October 1992.
29. K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In M. Gaudel and J. Woodcock, editors, *Proceedings of the* 3*rd International Symposium of Formal Methods Europe (FME 1996)*, volume 1051 of *LNCS*, pages 662–681. Springer-Verlag, 1996.
30. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the* 29*th ACM-SIGPLAN Conference on Principles of Programming Languages (POPL 2002)*, pages 58–70. ACM Press, 2002.
31. W. Hesselink. Eternity Variables to Simulate Specification. In *Proceedings of Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 117–130, Dagstuhl, Germany, 2002. Springer-Verlag.
32. J. Hooman. Compositional Verification of Timed Components using PVS. In B. Biel and V. Gruhn, editors, *Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 79 of *LNI*, pages 143–154, 2006.
33. P. B. Jackson. Verifying A Garbage Collector Algorithm. In J. Grundy and M. Newer, editors, *Proceedings of the* 11*th International Conference on Theorem Proving in Higher Order Logics (TPHOLS 1998)*, volume 1479 of *LNCS*, pages 225–244. Springer-Verlag, 1998.
34. B. Jonsson, A. Pnueli, and C. Rump. Proving Refinement Using Transduction. *Distributed Computing*, 12(2-3):129–149, 1999.
35. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.
36. M Kaufmann and J S. Moore. ACL2 home page. See URL `http://www.cs.utexas.edu/users/moore/acl2`.
37. M. Kaufmann and J S. Moore. A Precise Description of the ACL2 Logic. See URL `http://www.cs.utexas.edu/users/moore/publications/km97.ps.gz`, 1997.
38. M. Kaufmann and R. Sumners. Efficient Rewriting of Data Structures in ACL2. In D. Borrione, M. Kaufmann, and J S. Moore, editors, *Proceedings of* 3*rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 141–150, Grenoble, France, April 2002.

39. C. Kern and M. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, 1999.
40. M. Kyas and J. Hooman. A Semantics of Communicating Reactive Objects with Timing. *Software Tools for Technology Transfer (STTT)*, 8(4):97–112, 2006.
41. S. K. Lahiri and R. E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16*th International Conference on Computer-Aided Verification (CAV 2004)*, volume 3117 of *LNCS*, pages 135–147. Springer-Verlag, July 2004.
42. S. K. Lahiri, R. E. Bryant, and B. Cook. A Symbolic Approach to Predicate Abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Proceedings of the* 15*th International Conference on Computer-Aided Verification*, volume 2275 of *LNCS*, pages 141–153. Springer-Verlag, 2003.
43. L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.
44. L. Lamport. What Good is Temporal Logic? *Information Processing*, 83:657–688, 1983.
45. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 16(3):827–923, May 1994.
46. L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and Verifying Systems with TLA+. In E. Jul, editor, *Proceedings of the* 10*th ACM SIGOPS European Workshop*, pages 45–48, Copenhagen, Denmark, 2002.
47. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
48. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
49. P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2001.
50. P. Manolios. A Compositional Theory of Refinement for Branching Time. In D. Geist, editor, *Proceedings of the* 12*th Working Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 304–218. Springer-Verlag, 2003.
51. P. Manolios, K. Namjoshi, and R. Sumners. Linking Model-checking and Theorem-proving with Well-founded Bisimulations. In N. Halbwacha and D. Peled, editors, *Proceedings of the* 11*th International Conference on Computer-Aided Verification (CAV 1999)*, volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.
52. P. Manolios and D. Vroon. Algorithms for Ordinal Arithmetic. In F. Baader, editor, *Proceedings of the* 19*th International Conference on Automated Deduction (CADE 2003)*, volume 2741 of *LNAI*, pages 243–257, Miami, FL, July 2003. Springer-Verlag.
53. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
54. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.
55. J. Misra. A Logic of Concurrent Programing: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.
56. J. Misra. A Logic of Concurrent Programing: Safety. *Journal of Computer and Software Engineering*, 3(2):239–372, 1995.
57. J. Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Monographs in Computer Science. Springer, 2001.
58. J S. Moore. Proving Theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software*, pages 227–290. IOS Press, 2003.
59. J S. Moore and G. Porter. The Apprentice Challenge. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 24(3):1–24, May 2002.
60. K. Namjoshi. A Simple Characterization of Stuttering Bisimulation. In S. Ramesh and G. Sivakumar, editors, *Proceedings of the* 17*th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1997)*, volume 1346 of *LNCS*, pages 284–296. Springer-Verlag, 1997.
61. K. S. Namjoshi and R. P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the* 12*th International Conference on Computer-Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 435–449. Springer-Verlag, July 2000.
62. D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the* 5*th GI-Conference on Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.
63. L. Paulson. Mechanizing UNITY in Isabelle. 1(1):3–32, 2000.

64. A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 82–97. Springer-Verlag, 2001.
65. S. Ray and R. Sumners. Combining Theorem Proving with Model Checking Through Predicate Abstraction. *IEEE Design & Test of Computers*, 24(2):132–139, 2007.
66. S. Ray and R. Sumners. A Theorem Proving Approach for Verification of Reactive Concurrent Programs. In S. Burckhardt, S. Chaudhury, A. Farzan, G. Gopalakrishnen, and S. Seigel, editors, *4th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2 2011)*, July 2011.
67. D. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, 6:359–390, 1994.
68. J. Sawada and W. A. Hunt, Jr. Trace Table Based Approach for Pipelined Microprocessor Verification. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 1997)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.
69. N. Shankar. Mechanical Verification of a Generalized Protocol for Byzantine Fault Tolerant Clock Synchronization. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *LNCS*, pages 217–236. Springer-Verlag, January 1992.
70. R. Sumners. An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In M. Kaufmann and J S. Moore, editors, *2nd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2000)*, Austin, TX, October 2000.
71. R. Sumners. Fair Environment Assumptions in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J S. Moore, editors, *4th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.
72. R. Sumners and S. Ray. Reducing Invariant Proofs to Finite Search via Rewriting. In M. Kaufmann and J S. Moore, editors, *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, November 2004.
73. R. Sumners and S. Ray. Proving Invariants via Rewriting and Abstraction. Technical Report TR-05-35, Department of Computer Sciences, University of Texas at Austin, July 2005.
74. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.
75. K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: Mechanized Semantics of Sockets, Threads and Failures. In D. Le Métayer, editor, *Proceedings of the 11th European Symposium on Programming (ESOP 2002)*, volume 2305 of *LNCS*, pages 178–294. Springer-Verlag, 2002.