Proof Reduction of Fair Stuttering Refinement of Asynchronous Systems and Applications

Rob Sumners
Centaur Technology
rsumners@centtech.com

We present a series of definitions and theorems demonstrating how to reduce the requirements for proving system refinements ensuring containment of fair stuttering runs. A primary result of the work is the ability to reduce the requisite proofs on runs of a system of interacting state machines to a set of definitions and checks on single steps of a small number of state machines corresponding to the intuitive notions of freedom from starvation and deadlock. We further refine the definitions to afford an efficient explicit-state checking procedure in certain finite state cases. We demonstrate the proof reduction on versions of the Bakery Algorithm.

1 Introduction

Much of hardware and software system design focuses on how to optimize the execution of tasks by dividing the tasks into smaller computations and then scheduling and distributing these computations on the available resources. The natural specification for these systems is an assurance that the systems eventually complete the supplied tasks with results consistent with an atomic (or as atomic as feasible) execution of the task. We refresh the notion of fair stuttering refinements [10] as a means of codifying these specifications – a fair stuttering refinement between two systems ensures that every infinite run of a lower-level system with fair selection and finite stuttering maps to a similarly restricted infinite run of a higher-level system. This notion of refinement can allow sequences of smaller steps in the implementation to be mapped to single steps in the specification while additionally requiring that every task makes progress to completion.

Many previous efforts [10] have attempted to improve the capability of theorem provers in reasoning about refinements for distributed and concurrent systems. Previous efforts in regards to the ACL2 theorem prover [4] focused on trying to reduce the proofs of stuttering refinements with additional structures added to define fair selection and ensuring progress. These efforts generally boiled down to showing that a specification could match the step of an implementation or the implementation stuttered and some `rank` function decreased. The primary difficulty in these proofs was defining and proving an inductive invariant (either through ACL2 or trying to prove the invariant through some form of state exploration). In addition, the inclusion of additional structures to track fairness and progress as well as the resulting definition of `rank` functions proved complex. Further, the additional structures at times obfuscated whether the specification was complete and accurate.

In this paper, we take a different tack. We assume certain characteristics of the system we are trying to verify and leverage these characteristics in reducing the proof obligations. In particular, we first assume that the systems we are trying to verify are asynchronous in terms of how tasks make progress to completion. Further, we require the system definition to split the normal next-state transition relation into a next-state relation which only takes forward steps and a blocking relation which defines precisely when a task is blocked from making progress. From these assumed characteristics, we define proof reductions

Warren Hunt Jr. and Anna Slobodova (Eds.):
ACL2 Theorem Prover and its Applications
EPTCS 249, 2017, pp. 78–94, doi:10.4204/EPTCS.249.6

which reduce the goal of proving fair stuttering refinement to proving properties of a few task steps in relation to each other. These proof reductions have been formally defined and mechanically proven in ACL2 and are included in the supporting materials for this paper. In the remainder of this paper, we will cover two stages of proof reductions, review the application of the reductions to a version of the Bakery Algorithm. We conclude the paper with further reductions targeting efficient automatic checks in the finite state case.

2 Preliminaries

Commonly, systems are defined by an initial state predicate: `(init x)` and a next-state relation: `(next x y)`. A run of the system is then simply a sequence of states where the first state satisfies `(init x)` and each pair of states in the sequence satisfies `(next x y)`. We extend this basic construction in a couple of ways.

First, our goal is to reason about fair executions of a system (either as an assumption of fair selection for which task will update next or as a guarantee that every task makes progress). Thus, we assume that there is some set of task identifiers recognized by a predicate `(id-p k)` and add a task id parameter to the next-state relation: `(next x y k)` where this now relates state `x` to state `y` for an update to the task with id `k`. We also assume only one task updates at each step of the system without any prescribed order of task updates – essentially, the system is asynchronous at the level of task updates.

Second, we will find it useful to require the definition of an additional relation `(blok x k)` which returns true when the task identified by `k` is currently blocked from making progress in state `x`. Further, with this required definition of `(blok x k)`, we will also require the theorem: `(not (next x x k))` be proven and use inequality of next-states as a marker that a task is making progress to completion.

A system is then defined by three functions: `(init x)`, `(next x y k)`, and `(blok x k)`. Our final goal is to prove that the fair runs of an implementation system map to fair runs of a specification system with an allotment for finite stuttering and some guarantee of progress. A run of a system is a function `(run i)` which takes a natural `i` and returns a state of the system. Runs will naturally need to satisfy some constraints as detailed in Figure 1. For a given system named `sys`, the macro `(def-inf-run sys)` assumes the definition of `(sys-init x)`, `(sys-next x y k)`, `(sys-blok x k)`, `(sys-pick i)`, `(sys-run i)` and generates the definitions and theorems defining the properties for the run as in Figure 1.

Of particular note, the function `(step x y k)` relates states `x` and `y` via `(next x y k)` only if `k` is not blocked in `x` and we are not stuttering (denoted by the input `k` being `nil`) – (as a note, the only requirement we place on `id-p` is that `(not (id-p nil))`). So, an infinite run is defined by two functions `(run i)` which defines the sequence of states and `(pick i)` which defines the sequence of task identifiers selected. We constrain `(pick i)` to only return an `id-p` or `nil`. We can now naturally define fair selection of `(pick i)` by positing the existence of a function `(fair k i)` which returns natural numbers and for each task id `k` will strictly decrease when `k` is not selected – see Figure 2. The macro `(def-fair-pick sys id-p)` assumes the definitions of `(sys-pick i)`, `(sys-fair k i)`, and `(id-p k)` and produces the theorems in Figure 2. We use the term *fair run* for an infinite run with a fair picker.

Fair selection of task identifiers ensures that each run only has finite stuttering and that each task gets a chance to make progress, but it does not guarantee that tasks actually make progress. We introduce the term *valid run* for a run which is not only fair but ensures progress for each task. In order to ensure progress, we define a function `(prog k i)` similar to `(fair k i)` but in addition to ensuring `pick`

```
(encapsulate
 ((run  (i) t)
  (pick (i) t))
 (local (defun run  (i) ....))
 (local (defun pick (i) ....))

 (defun step (x y k)
   (if (or (null k)    ;; finite stutter
           (blok x k)) ;; or k is blocked in x
       (equal x y)
     (next x y k)))

 (defthm run-init-thm (implies (zp i)   (init (run i))))
 (defthm run-step-thm (implies (posp i) (step (run (1- i)) (run i) (pick i))))
)
```

Figure 1: Definition of an infinite run in ACL2

```
(defthm fair-nat-thm (natp (fair k i)))

(defthm pick-fair-thm
  (implies (and (posp i)
                (id-p k)
                (not (equal (pick i) k)))
           (< (fair k i) (fair k (1- i)))))
```

Figure 2: *Fair Runs*: fair task selection during a run

```
(defthm prog-is-nat (natp (prog k i)))

(defthm run-prog-thm
  (implies (and (posp i)
                (id-p k)
                (or (not (equal (pick i) k))
                    (equal (run i) (run (1- i)))))
           (< (prog k i) (prog k (1- i)))))
```

Figure 3: *Valid Runs*: ensuring task progress during a run

eventually equals k, we also need to ensure that a state change actually occurs. The properties in Figure 3 ensure a *valid run* and the macro (def-valid-run sys id-p) produces these theorems for id-p, sys-run, sys-pick, and sys-prog. We note that a valid run is also a fair run and thus our notion of refinement is compositional – but it is better to prove that all fair runs of the implementation are valid runs and then restrict the refinement to valid runs mapping to valid runs and reduce the proof requirements accordingly at each step. This is straightforward from what we present in this paper but we do not focus on it in this paper.

3 Proof Reduction to Single System Steps

The principle objective of fair stuttering refinement is to prove that the fair runs of an implementation map to valid runs of a specification. The first set of proof reductions we present refresh similar attempts in past work [10, 8] in transferring these proof requirements on infinite runs to properties about single steps of two systems impl and spec. The difference between these past efforts and the work presented is that we directly specify properties related to guaranteeing progress for each task in the system and we leverage the definition of the blocking relation. In addition, while the proof reduction to single step presented in this section could be used as is, the design of the reduction is influenced by the needs of subsequent proof reductions over tasks presented in Section 4. The book "general-theory.lisp" in the supporting materials covers the work in this section.

The goal is to show that if one were to prove certain properties about steps of an implementation system impl and a specification system spec, then one could infer a *fair stuttering refinement* – every fair run of impl maps to a valid run of spec. We wish to prove this for any specification and implementation system, so specifically, for any impl and spec and any fair run impl-run of the implementation, if we have proven the required properties then we can map impl-run to a valid run spec-run of spec. An overview of the structure of the book "general-theory.lisp" is provided in Figure 4 and attempts to codify this goal. The definitions of the impl and spec systems and the fair run impl-run of impl are constrained within an encapsulate to only have the properties: (def-inf-run impl), (def-fair-pick impl id-p), (def-system-props impl id-p), (def-valid-system impl id-p), and (def-match-systems impl spec id-p). From this fair run impl-run and the properties proven on spec and impl, we can build a valid run spec-run. While it is not possible to make this a closed-form statement of correctness in ACL2, we believe the structure of the book is sufficient to establish the claim.

The function (spec-run i) in Figure 4 defines the spec state at each time to simply be (impl-map (impl-run i)) and the function (spec-pick i) is simply (impl-pick i) except that we introduce finite stutter (i.e. return nil) if the mapped state doesn't change. It is customary to define some notion of observation or labeling of states that must be preserved to ensure correlation of behavior between spec and impl – we assume human review has ensured that the mapping from impl states to spec states preserves any observations relevant to the specification. In this regard, it is relevant that the mapped run on the spec is relatively simple in definition as it avoids errors or oversights in specification due to an obfuscation of how the implementation and specification are correlated.

The properties we need to prove for impl and spec are defined by the macros def-system-props, def-valid-system, and def-match-systems. Along with the functions defining the impl and spec systems, additional definitions are required for each of these macros. We will shortly go into greater detail on the properties we will assume as constraints for these functions, but first, we refer to the listing provided in Figure 5.

```
(encapsulate
 (....) ;; constrained functions defining impl and spec.
 ....   ;; local def.s and prop.s to show constraints.

;; ASSUMPTIONS:
   ;; assume relevant properties of given systems impl and spec:
 (def-system-props  impl id-p)
 (def-valid-system  impl id-p)
 (def-match-systems impl spec id-p)
   ;; assume an infinite run of the impl system:
 (def-inf-run       impl)
 (def-fair-pick     impl id-p)
)

.... ;; def.s and theorems to establish results.

;; Define the corresponding (assumed to preserve "observations") spec run:
(defun spec-run (i) (impl-map (impl-run i)))

;; spec-pick will introduce stutter into spec-run when the mapped state doesn't change:
(defun spec-pick (i)
  (and (not (equal (impl-map (impl-run (1- i)))
                   (impl-map (impl-run i))))
       (impl-pick i)))

.... ;; additional def.s and theorems to establish results.

;; CONCLUSIONS:
   ;; and prove that the corresponding spec-run is indeed a valid run of spec:
(def-inf-run   spec)
(def-valid-run spec id-p)
```

Figure 4: Structure of the book ‘‘`general-theory.lisp`’’

- IMPL system definition:
  - `(impl-init x)` – initial predicate on states `x` for impl system
  - `(impl-next x y k)` – state `x` transitions to state `y` on selector `k`
  - `(impl-blok x k)` – state `x` blocked for transitions for selector `k`
- SPEC system definition:
  - `(spec-init x)` – initial predicate on states `x` for spec system
  - `(spec-next x y k)` – state `x` transitions to state `y` on selector `k`
  - `(spec-blok x k)` – state `x` blocked for transitions for selector `k`
- Definitions needed for `(def-system-props impl id-p)` macro:
  - `(impl-iinv x)` – inductive invariant for states in impl
- Definitions needed for `(def-match-systems impl spec id-p)` macro:
  - `(impl-map x)` – maps impl states to corresponding spec states
  - `(impl-rank k x)` – ordinal decreases until spec matches transition for `k`
- Definitions needed for `(def-valid-system impl id-p)` macro:
  - `(impl-noblk k x)` – is task id `k` invariantly unblocked in state `x`
  - `(impl-nstrv k x)` – ordinal decreases until `k` is in a `noblk` state
  - `(impl-starver k x)` – potential starver of `k` in `x` which is not blocked

Figure 5: Function Definitions for Single-Step System-Level Properties

The macro `(def-system-props impl id-p)` expands into simple theorems ensuring `(not (id-p nil))`, ensuring `(impl-next x x k)` is not valid, and ensuring the state predicate `(impl-iinv x)` is an inductive invariant for `impl` – namely that `(impl-iinv x)` holds in the initial state and persists across `(impl-next x y k)` transitions.

The `(def-match-systems impl spec id-p)` macro requires defining `(impl-map x)`, a mapping from impl states to spec states and a ranking function `(impl-rank k x)` which returns an ordinal for each task id `k`. The main properties generated by `def-match-systems` are the following:

```
(defthm map-matches-next
  (implies (and (impl-iinv x) (id-p k) (!= (impl-map x) (impl-map y))
                (impl-next x y k)
                (not (impl-blok x k)))
           (and (spec-next (impl-map x) (impl-map y) k)
                (not (spec-blok (impl-map x) k)))))

(defthm map-finite-stutter
  (implies (and (impl-iinv x) (id-p k) (= (impl-map x) (impl-map y))
                (impl-next x y k))
           (o<  (impl-rank k y) (impl-rank k x))))

(defthm map-rank-stable
  (implies (and (impl-iinv x) (id-p k) (id-p l) (!= k l)
                (impl-next x y l))
           (o<= (impl-rank k y) (impl-rank k x))))
```

The theorem `map-matches-next` ensures that on any step `(impl-next x y k)` for task `k` which is not blocked in `x` and where the mapped specification state changes (i.e. `(!= (impl-map x) (impl-map y))`) then the spec must be able to match the transition and the spec state cannot be blocked in the spec for task `k`. The theorem `map-finite-stutter` ensures that when the mapped implementation state does not change on an update for task `k` in impl, then the ordinal returned by `impl-rank` must strictly decrease and the theorem `map-rank-stable` ensures that this ordinal does not increase when task `k` is not selected. The clear intent of these properties is to ensure that as long as a task `k` is not indefinitely blocked when it is selected for update in impl, then eventually a matching spec transition must be generated. The question is then naturally how to ensure that a task is not indefinitely blocked. This concept of being indefinitely blocked is commonly called "starvation" in the literature and the `def-valid-system` macro will generate properties intended to ensure that no task is starved.

The `(def-valid-system impl id-p)` macro requires the definition of a predicate `(impl-noblk k x)` which is true when the task `k` can no longer be blocked in state `x` and a function `(impl-nstrv k x)` which nominally returns an ordinal that decreases until `(impl-noblk k x)` is true. Once a task `k` reaches an `impl-noblk` state, it can no longer be blocked until it transitions and thus the fair selection of `k` will ensure a transition of `k` occurs. Unfortunately, a task's progress to an `impl-noblk` state may be dependent on any number of other tasks or components in the impl state. At this general level of system definition, we only have system states `x` and task ids `k`, so we imagine that for any `k` and `x`, we could define a set of task ids called the *starve-set* which need to make progress before `k` can reach a `noblk` state. Updates to ids which are not in this starve-set should simply have no effect on this progress and so we will assume that `(impl-nstrv k x)` will strictly decrease on transitions for ids in the starve-set and remain unchanged otherwise. Unfortunately, it might be possible that all of the tasks in the *starve-set* are blocked and so we need the additional definition of an `(impl-starver k x)` which returns an id in this *starve-set* which is currently not blocked in state `x`. Additionally, we need to ensure that when an element outside of the starve-set is chosen, that the `(impl-starver k x)` remains unchanged. The

encoding of these properties as ACL2 theorems are generated from the `def-valid-system` macro and are listed here:

```
(defthm noblk-blk-thm
  (implies (and (iinv x) (id-p k)
                (noblk k x))
           (not (blok x k))))

(defthm noblk-inv-thm
  (implies (and (iinv x) (id-p k) (id-p l) (!= k l)
                (next x y l)
                (noblk k x))
           (noblk k y)))

(defthm starver-thm
  (implies (and (iinv x) (id-p k)
                (not (noblk k x)))
           (not (blok x (starver k x)))))

(defthm nstrv-decreases
  (implies (and (iinv x) (id-p k) (!= k (starver k x))
                (next x y (starver k x))
                (not (noblk k x)))
           (o< (nstrv k y) (nstrv k x))))

(defthm nstrv-holds
  (implies (and (iinv x) (id-p k) (id-p l) (!= k l)
                (next x y l)
                (not (noblk k x)))
           (o<= (nstrv k y) (nstrv k x))))

(defthm starver-persists
  (implies (and (iinv x) (id-p k) (id-p l) (!= k l) (!= l (starver k x))
                (next x y l)
                (not (noblk k x))
                (= (nstrv k y) (nstrv k x)))
           (= (starver k y) (starver k x))))
```

And with these properties assumed as constraints, we return to the goal of proving that the infinite run defined by `(spec-run i)` and `(spec-pick i)` from Figure 4 is indeed a valid run of `spec`. In order to do that we need to define a function `spec-prog` which satisfies the requirements set out in Figure 3. First, it is useful to define an `(impl-prog k i)` and show that the `impl-run` is indeed a valid run.

The definition of `(impl-prog k i)` is in Figure 6 and essentially looks forward into `impl-run` until we reach an `i` where `k` is picked and the state changes. The key point is obviously the question of what is the measure for demonstrating that this function terminates and this follows from our earlier discussion about the `(impl-noblk k x)`, `(impl-nstrv k x)`, and `(impl-starver k x)` functions. If we have `(impl-noblk k ..)` at the current state, then the task with id `k` cannot be blocked and we can simply countdown the `(impl-fair k i)` measure until task `k` is selected – the state will change at that time since `k` will still be unblocked and `impl-next` must change the state. If `(impl-noblk k ..)` does not currently hold then we know there is a task id `(impl-starver k ..)` which cannot be blocked in the current state and either `(impl-nstrv k ..)` strictly decreases or `(impl-starver k ..)` will not change. Thus, at each step, either the `impl-nstrv` measure strictly decreases or the fair measure for `impl-starver` counts down and will eventually expire and `impl-nstrv` will strictly decrease.

This `(impl-prog k i)` thus ensures that task `k` is picked and changes state in `(impl-run i)` but we now must guarantee that the mapped state changes in `spec`. In the case that the mapped state doesn't change, we know that the `(impl-rank k ..)` must decrease and that the impl-rank remains unchanged

```
(defun ord-nat-pair (o n)
  ;; simple function which returns lex. product of an o-p o and natp n:
  (make-ord (if (atom o) (1+ o) o) 1 n))

;; First prove that the implementation run is a valid run...
(defun impl-prog (k i)
  (declare (xargs :measure
                  (if (impl-noblk k (impl-run i))
                      (impl-fair k i)
                    (ord-nat-pair (impl-nstrv k (impl-run i))
                                  (impl-fair (impl-starver k (impl-run i)) i)))))
  (cond
   ((or (not (and (natp i) (id-p k)))              ;; ill-formed inputs.. or
        (and (= (impl-pick (1+ i)) k)              ;;    impl-pick matches k
             (!= (impl-run (1+ i)) (impl-run i)))) ;;  ..and k makes progress
    0)
   (t (1+ (impl-prog k (1+ i))))))

;; ...And use that to show that the mapped spec run is also valid
(defun spec-prog (k i)
  (declare (xargs :measure
                  (ord-nat-pair (impl-rank k (impl-run i))
                                (impl-prog k i))))
  (cond
   ((or (not (and (natp i) (id-p k)))              ;; ill-formed inputs.. or
        (and (= (spec-pick (1+ i)) k)              ;;    spec-pick matches k
             (!= (spec-run (1+ i)) (spec-run i)))) ;;  ..and k makes progress
    0)
   (t (1+ (spec-prog k (1+ i))))))
```

Figure 6: Defined Measure Functions on Infinite Runs

when other ids are selected. This is the basis for the definition `(spec-prog k i)` in Figure 6.

4 Proof Reduction to a Small Bounded Number of Tasks

In the previous section, we presented a proof reduction of the requirements for fair stuttering refinement from reasoning about infinite runs of systems to reasoning about single steps of systems. We did not make any assumption about the state structure of the systems other than that updates occurred asynchronously at some prescribed task level. In this section, we will assume a structure on the states of a system and show how to reduce the requisite properties from across the large state structure to the properties on components of the state. Throughout this section and the next, we will use the set `(s k v r)` and get `(g k r)` operations from the records book [5]. In particular, `(g k r)` takes a record `r` and returns either the value previously set for key `k` in record `r` or `nil` as default.

The book "trans-theory.lisp" in the supporting materials for this paper includes the definitions and proofs relating to this section. The structure of this book is similar to that shown for "general-theory.lisp" in Figure 4 in that there is an encapsulation which entails the system definitions and properties we want to assume and then outside of the encapsulation, we prove the derived results. For the previous section, in "general-theory.lisp", we proved the property in Figure 8 (in an abuse of notation pretending ACL2 were higher-order for a moment), For this section, our goal is to define systems at a task level and derive the system-level results. In the same higher-level-abuse format as above, we have the property from "trans-theory.lisp" also in Figure 8.

We take the state of the system to be a record associating keys to task states.. what we call *t-states*. The task id selected on input is now simply one of these keys and the update of the state will only update

- TR-IMPL system definition:
  - `(tr-impl-t-init a k)` – initial state predicate for t-state `a` and key `k`
  - `(tr-impl-t-next a b x)` – t-state `a` transitions to t-state `b` in state `x`
  - `(tr-impl-t-blok a b)` – t-state `a` is blocked from stepping by t-state `b`
- TR-SPEC system definition:
  - `(tr-spec-t-init a k)` – initial state predicate for t-state `a` and key `k`
  - `(tr-spec-t-next a b x)` – t-state `a` transitions to t-state `b` in state `x`
  - `(tr-spec-t-blok a b)` – t-state `a` is blocked from stepping by t-state `b`
- Definitions needed for `(def-tr-system-props tr-impl)` macro:
  - `(tr-impl-iinv x)` – inductive invariant as previously.. no change at task-level
- Definitions needed for `(def-match-tr-systems tr-impl tr-spec)` macro:
  - `(tr-impl-t-map a)` – maps tr-impl t-states to corresponding tr-spec t-states
  - `(tr-impl-t-rank a)` – ordinal decreases until mapped t-state must change
- Definitions needed for `(def-valid-tr-system tr-impl)` macro:
  - `(tr-impl-t-noblk a b)` – is t-state `a` invariantly not-blocked by t-state `b`
  - `(tr-impl-t-nstrv a b)` – positive natural which strictly decreases until `(t-noblk a b)`
  - `(tr-impl-t-nlock k x)` – ordinal strictly decreases on from `k` to blocker of `k` in `x`

Figure 7: Function Definitions for Single-Step Task-Level Properties

```
"general-theory.lisp":
  (implies (and (def-system-props  impl id-p)
                (def-valid-system  impl id-p)
                (def-match-systems impl spec id-p))
           (implies (and (def-inf-run   impl)
                         (def-fair-pick impl id-p))
                    (and (def-inf-run   spec)
                         (def-valid-run spec id-p))))

"trans-theory.lisp":
  (implies (and (def-tr-system-props  tr-impl)
                (def-valid-tr-system  tr-impl)
                (def-match-tr-systems tr-impl tr-spec))
           (and (def-system-props  tr-impl key-p)
                (def-valid-system  tr-impl key-p)
                (def-match-systems tr-impl tr-spec key-p)))
```

Figure 8: High-Level properties in for theory files definitions

the corresponding entry of the record. We presume and constrain a fixed finite set of keys – (keys) – of arbitrary size and composition and membership in this set will define the id-p test for task id selection. The state of the system is then a record mapping members of this finite set (keys) to t-states and the system will be defined on the task level. We define task-based systems by assuming the pertinent definitions on task states in the system and derive the system-level definitions across the state. We name these systems derived from the task-level definitions as tr-impl and tr-spec. In Figure 5 from the previous section, we listed the function definitions required for the single-step system-level properties – we do the same for the single-step task-level properties in Figure 7.

Many of the system-level derived functions follow simply from the task-level. The system-level (tr-impl-init x) predicate checks that (tr-impl-t-init (g k x) k) holds for all keys k. The system-level (tr-impl-next x y k) only updates (g k x) as (tr-impl-t-next (g k x) (g k y) x) and leaves all other keys untouched in x. The system-level block function (tr-impl-blok x k) checks if there is any key l such that (tr-impl-t-blok (g k x) (g l x)). The system-level mapping function simply goes through all keys and calls tr-impl-t-map for the corresponding t-state and the system level rank just calls (tr-impl-t-rank (g k x)) directly. The inductive invariant does not change; there is just one inductive invariant defined on the entire record defining the system state. Additionally, the system-level proofs for (def-system-props tr-impl key-p) and (def-match-systems tr-impl tr-spec key-p) are straightforward and follow from these system-level definitions and properties of task-level definitions.

The functions and properties for proving progress and valid impl runs are more involved. For the sake of brevity and readability, we will drop the tr-impl- prefix from the system-level and task-level defintions for the remainder of this section. In addition to ensuring that t-nlock returns an ordinal and t-nstrv returns a positive natural number[1], the macro (def-valid-tr-system tr-impl) introduces the following properties:

```
(defthm t-noblk-blk-thm
  (implies (and (iinv x) (key-p k) (key-p l)
                (t-noblk (g k x) (g l x)))
           (not (t-blok (g k x) (g l x)))))

(defthm t-noblk-inv-thm
  (implies (and (iinv x) (key-p k) (key-p l)
                (t-noblk (g k x) (g l x))
                (t-next (g l x) c x))
           (t-noblk (g k x) c)))

(defthm t-nlock-decreases
  (implies (and (iinv x) (key-p k) (key-p l)
                (t-blok (g k x) (g l x)))
           (o< (t-nlock l x)
               (t-nlock k x))))

(defthm t-nstrv-decreases
  (implies (and (iinv x) (key-p k) (key-p l)
                (not (t-noblk (g k x) (g l x)))
                (not (t-noblk (g k x) c))
                (t-next (g l x) c x))
           (< (t-nstrv (g k x) c)
              (t-nstrv (g k x) (g l x)))))
```

[1] In the supporting materials for this paper, t-nstrv is generalized to be a list of natural numbers which is then summed and combined into a list of lists of naturals, but for the sake of clarity and brevity in this paper, we keep a simpler definition for t-nstrv. We could not use a generic ACL2 ordinal for t-nstrv since we needed to form lexicographic products of sums of these ordinals and that is not possible for arbitrary ordinals in ACL2.

The system-level (noblk k x) definition simply checks that (t-noblk (g k x) (g l x)) holds for every key l and as such, the task-level t-noblk-blk-thm and t-noblk-inv-thm are task-level projections of their system-level counterparts and the system-level properties follow fairly easily. The more interesting case comes up in defining the system-level (nstrv k x) and (starver k x). For the task-level, the property t-nlock-decreases ensures that we don't have any "deadlocks" or simply that for any set of keys, there is always some key in that set which is not blocked in x by some other key in that set. The combination of t-nstrv-decreases and the properties of t-noblk ensure that no task can be starved by another task.

The intuition behind defining the system-level (nstrv k x) begins by recognizing that if (not (noblk k x)) then there is some set of keys l such that (not (t-noblk (g k x) (g l x))). We will call this set of keys the *may-block set*. But since t-noblk persists once we reach it, then we could sum up the (t-nstrv (g k x) (g l x)) for this may-block set and the resulting ordinal would decrease until we reached a state where k was t-noblk for all l and thus noblk. Assume for the moment that k were not blocked (i.e. we could set (starver k x) to be k), then consider an update for some key l. If that key were in the may-block set of k then the ordinal would decrease. If l is not in the may-block set of k then (t-noblk (g k x) (g l x)) and the transition of l cannot change the blocked status of k and it cannot change the may-block set for k and so progress is made. Unfortunately there is no guarantee that k is not blocked and thus we cannot pick a suitable starver which ensures progress when selected.

But from the property t-nlock-decreases, starting with k in x, we can find a key which is not blocked by checking if the key is blocked and recurring on the first blocking key we find if we are blocked. This is the definition of the function (starver k x) and is included here:

```
(defun starver (k x)
  (declare (xargs :measure (t-nlock (g k x))))
  (if (and (iinv x) (key-p k) (blok x k))
      (starver (pikblk k x) x)
    k))
```

The function (pikblk k x) returns the first key we find such that (t-blok (g k x) (g (pikblk k x))). So, from k, we can find a key which is unblocked, but the question is then how to build a measure from the starve-set including k and (starver k x). The answer is to build a natural list where each element is the sum of t-nstrv for the may-block set (as we described before) in each step along the path from k to (starver k x) and define our ordinal as the lexicographic product of the naturals in this list. The first observation is that at the end of this list we will have the summation of t-nstrvs for the may-block set of (starver k x) and since (starver k x) is not blocked, it will make progress as we discussed before. The other key observation is that at each step, the (pikblk k x) key will be in the may-block set of k and thus even though a transition of (pikblk k x) may modify its may-block set and potentially increase the measure from that point, the measure for the may-block set of k will decrease and the ordinal over all will decrease. This list of naturals is defined by the function (nstrvs* k x) as follows where the function (scar s) and (scdr s) return the first element and remainder of a set respectively and (card s) returns the cardinality of the set.

```
(defun sum-nsts* (k x s)
  (declare (xargs :measure (card s)))
  (if (null s) 1
    (+ (if (t-noblk (g k x) (g (scar s) x)) 0
         (t-nstrv (g k x) (g (scar s) x)))
       (sum-nsts* k x (scdr s)))))

(defun sum-nsts (k x) (sum-nsts* k x (keys)))

(defun nstrvs* (k x)
  (declare (xargs :measure (t-nlock (g k x))))
  (if (and (iinv x) (key-p k) (blok x k))
      (cons (sum-nsts k x) (nstrvs* (pikblk k x) x))
    (list (sum-nsts k x))))

(defun nats->o (n l)
  (cond ((zp n) 0)
        ((atom l) (make-ord n 1            (nats->o (1- n) ())))
        (t        (make-ord n (1+ (car l)) (nats->o (1- n) (cdr l))))))

(defun tr-impl-nstrv (k x)
  (nats->o (card (keys)) (nstrvs* k x)))
```

As we mentioned, the function `nstrvs*` returns a natural list and we build a suitable ordinal from this list using the function `nats-o`. But because the length of the path to `(starver k x)` from `k` could change and thus the length of the nstrvs* list could change, we need to make the defined ordinal "first-aligned" – where the first element in the list is mapped to a coefficient of the same exponent no matter the length of the rest of the list. We use `(card keys)` as the starting exponent and prove separately that the length of the list returned by `nstrvs*` can never exceed `(card keys)`.

This construction also shows one of the reasons we assume an arbitrary fixed finite set of `(keys)` (in order to put a bound on `(len (nstrv* k x))`), but this restriction makes sense for other reasons as well. If the set of keys were not finite, then we would need some additional requirement to ensure that a task were not persistently blocked by an infinite sequence of newly instantiated tasks. Other options exist to avoid this (such as requiring that all new tasks cannot block existing tasks) but these alternatives end up imposing constraints we believe are too restrictive.

5 Example – A Bakery Algorithm

We use the Bakery algorithm as an example application of the proof reductions we present in this paper. The Bakery algorithm was developed by Lamport [7] as a solution to mutual exclusion with the additional assurance that every task would eventually gain access to its exclusive section. The Bakery algorithm has also been a focus of previous ACL2 proof efforts [9].

The essential idea of the algorithm is that each task first goes through a phase where it chooses a number (much like choosing a number in a bakery) and then later compares the number against the numbers chosen by the other tasks to determine who should have access to the exclusive section. The version of the Bakery algorithm we will use is defined in Figure 9 (the `(upd r .. updates ..)` simply expands into a nest of record sets).

Each task will start in program location 0 and start its `:choosing` phase. During the `:choosing` phase, the task will grab the current shared max (via the function `(curr-sh-max x)`) and then set its

```
(defun bake-impl-t-init (a k)
  (= a (upd nil :loc 0 :key k :pos 1 :old-pos 0 :temp 0 :sh-max 1)))

(defun bake-impl-t-next (a b x)
  (case (g :loc a)
        (0 (= b (upd a :loc 1 :choosing t)))
        (1 (= b (upd a :loc 2 :temp (curr-sh-max x))))
        (2 (= b (upd a :loc 3 :pos (1+ (g :temp a))
                              :old-pos (g :pos a)
                              :pos-valid t)))
        (3 (= b (upd a :loc 4 :sh-max (if (> (curr-sh-max x) (g :temp a))
                                          (curr-sh-max x)
                                        (g :pos a)))))
        (4 (= b (upd a :loc 5 :choosing nil)))
        (5 (= b (upd a :loc 6))) ;; we are potentially blocked here
        (6 (= b (upd a :loc 7))) ;; we are potentially blocked here
        (t (= b (upd a :loc 0 :pos-valid nil)))))

(defun bake-impl-t-blok (a b)
  (or (and (= (g :loc a) 5)
           (g :choosing b))
      (and (= (g :loc a) 6)
           (and (g :pos-valid b)
                (lex< (g :pos b) (ndx (g :key b))
                      (g :pos a) (ndx (g :key a)))))))
```

Figure 9: Bakery Implementation System Definition

own position `:pos` to be 1 more than the shared max. In program `:loc` 3, a compare-and-swap is implemented and the shared-max is potentially updated. The task then ends its `:choosing` phase.

After the `:choosing` phase, the task will enter program locations 5 and 6. In these locations, the `t-blok` predicate ensures that the task wait until other tasks are not `:choosing` and then wait until it has the least position (where potential ties are broken by comparing the `ndx` of the `:key` in the set `(keys)`.

In order to prove `(def-valid-tr-system bake-impl)`, we need to define the `t-nlock`, `t-noblk`, and `t-nstrv` functions. The definition of `(t-nlock x k)` needs to return an ordinal that is strictly decreasing from the blocked task to the blocking task. From the `bake-impl-t-blok` relation, we note that `:choosing` states cannot be blocked and that `lex<` is already well-founded, so we can devise a suitable `bake-impl-t-nlock`:

```
(defun bake-impl-t-nlock (k x)
  (let ((a (g k x)))
    (make-ord 2 (if (g :choosing a) 1 2)
    (make-ord 1 (1+ (nfix (g :pos a)))
                (ndx (g :key a))))))
```

For the `t-noblk` and `t-nstrv` definitions, we need to analyze where one task can no longer block another task. The simple answer is that `(t-noblk a b)` is reached once task `b` has chosen a `:pos` greater than the one in `a`, but we also have to make sure that task `b` is not choosing either. In addition, we note that if `a` cannot currently be blocked by any task, then we can set `t-noblk` and task `a` cannot be blocked if it is not in program locations 5 or 6. With that, we define `bake-impl-t-noblk`:

```
(defun bake-impl-t-noblk (a b)
  (or (and (!= (g :loc a) 5)
           (!= (g :loc a) 6))
      (and (not (g :choosing b))
           (> (g :pos b) (g :pos a)))))
```

Finally, we need to define t-nstrv which counts down until we reach the t-noblk state. The simple answer would be to count from the exit of :choosing phase until the next exit from the :choosing phase. Thus, we would return 8 if (g :loc b) was 5 and then proceed down to 6 for 7, then 5 for 0 (wrapping back), then down to 1 for 4 (end of next :choosing). This almost works.. except that it is possible for b to be in :loc 2, 3, or 4 with a :pos lower than a but a has proceeded further. Thus, we need to add a few steps for the case of being in 2,3,4 with a potentially lower :pos but when we come back around for the next :choosing, we will reach noblk:

```
(defun bake-impl-t-nstrv (a b)
  (pos-fix
   (cond ((or (and (= (g :loc b) 2)
                   (< (g :temp b) (g :pos a)))
              (and (> (g :loc b) 2)
                   (<= (g :pos b) (g :pos a))))
          (+ 8 (- 8 (g :loc b))))
         ((>= (g :loc b) 5)
          (+ 5 (- 8 (g :loc b))))
         (t
          (+ 0 (- 5 (g :loc b)))))))
```

With these definitions and a suitable invariant bake-impl-iinv, we can prove the theorems for (def-valid-tr-system bake-impl) – each of which just blasts into a big case split which pushes through. For the specification of the bakery algorithm, we have a simple system bake-spec defined in Figure 10. Each task in this system goes through the following steps: first, load up a new provisional :pos in the :load variable, then proceed to set the :pos variable and begin to arbitrate in the 'interested state. Tasks are blocked if some other task is in the 'go state or is in the 'interested state and has a lower :pos. The definitions and proof of (def-match-tr-systems bake-impl bake-spec) are fairly straightforward and included in Figure 10. We note that it is feasible (although not required) to define the supporting functions and prove (def-valid-tr-system bake-spec) – this proves that all fair runs of bake-spec are valid while the earlier proofs only ensured that the runs mapped from bake-impl runs were valid.

In previous work [10], a similar proof effort was conducted in proving a fair stuttering refinement for the definition of the Bakery Algorithm. In that effort, the proof was complicated by the need to add additional structures to track fair scheduling and to ensure correlation to a specification which had additional structures to ensure progress for each task. These complications were avoided in the proof here and as such, much less definition and details were required. The reduced proof we present here is primarily the definition and proof of a sufficient inductive invariant but much additional definition and proof was required in the earlier work [10].

6 Further Reductions and Considerations

We conclude this paper with a discussion of further reductions and considerations for search procedures. We first acknowledge that some of the task-based definitions may seem overly restrictive. For example, the (blok a b) relation being defined simply on task states. In essence, this restricts us from supporting systems where a task may be blocked when only some combination of tasks exist. It is possible to extend the notion of blocking to be more general but it comes at the cost of the complexity of other definitions and checks and we have generally found that by adding auxiliary variables to the task state, we can fit any appropriate system under these restrictions.

```
(defun bake-spec-t-init (a k)
  (declare (ignore k))
  (and (= (g :loc a) 'idle) (= (g :pos a) 0) (= (g :load a) 0)))

(defun bake-spec-t-next (a b x)
  (case (g :loc a)
    (idle       (and (= (g :loc b)  'loaded)
                     (= (g :pos b)  (g :pos a))
                     (natp (g :load b))
                     (>  (g :load b) (max-pos x))
                     (>= (g :load b) (max-load x))))
    (loaded     (= b (upd a :loc 'interested
                            :pos (g :load a))))
    (interested (= b (upd a :loc 'go)))
    (go         (= b (upd a :loc 'idle)))))

(defun bake-spec-t-blok (a b)
  (and (= (g :loc a) 'interested)
       (or (= (g :loc b) 'go)
           (and (= (g :loc b) 'interested)
                (< (g :pos b) (g :pos a))))))

(defun bake-impl-t-map (a)
  (upd nil
       :loc  (case (g :loc a)
                   ((0 1) 'idle)
                   ((2 3) 'loaded)
                   ((4 5 6) 'interested)
                   (t 'go))
       :pos  (case (g :loc a)
                   (3 (g :old-pos a))
                   (t (g :pos a)))
       :load (case (g :loc a)
                   (2 (1+ (g :temp a)))
                   (t (g :pos a)))))

(defun bake-impl-t-rank (a)
  (case (g :loc a)
        (0 1)       (1 0)
        (2 1)       (3 0)
        (4 2) (5 1) (6 0)
        (t 0)))
```

Figure 10: Bakery Specification System and Definitions for Proving Matching from Impl

This paper focused on mechanized proof reductions for general system definitions, but the work also supports improvements in more efficient automatic verification (in particular when the underlying task state space is finite). For example, take a somewhat draconian restriction that `(t-next a b x)` can be defined as `(t-next a b)` and similarly, the initial state predicate ignored the input `k` – a few things develop in this case. First, we note (somewhat trivially) that for every reachable system state composed of (say) n task states, that every "substate" of $n-1$ task states can also be reached. Additionally, if the task state space were finite, then we could compute all of the potential cycles in the blocking relation and for each cycle of size n, we could determine if it was reachable by searching through the system states with only n keys. A similar check could be implemented for the other properties with no more than 2 keys needed.

Of additional interest in this case, is that reachable states of these systems have a particular characterization. Consider any run of a system.. any steps in the run can be permuted as long as the permutation does not change the blocking relationship between the tasks involved. This means that for every reachable state, one can define a set of canonical runs which involves only stepping tasks until the blocking relationship is changed with respect to another task and then switching to the blocking task or stepping back and switching to the blockee task. This property limits the structure of potential invariants and suggests procedures for proving invariants over pairs of states. The inductive invariant `iinv` over the system state can be defined by invariant definitions on single task states, pairs of states, triples, etc. and in most cases (potentially with additional auxiliary variables), sufficiently defined on single t-states and pairs of t-states. In this case, the requisite properties of the defined `t-nlock`, `t-nstrv`, `t-noblk`, `t-map` and `t-rank` definitions could be proven via GL on the specified finite t-state domain using a SAT solver with a sufficient conditions on the t-states assumed. An inductive invariant (defined on single t-states and pairs of t-states) could be defined that proved each of these sufficient condition assumptions as invariant of the system. A model checker could be used to reduce the definitional requirements further by checking invariants (not requiring inductive invariants) and by checking for bad cycles to show that one could infer the existence of suitable `t-nlock`, `t-nstrv`, and `t-rank`. The model checking problems could be limited to a small number of tasks and possibly only single task stepping depending on the conditions of the defintion. The work presented in this paper is a step into many potential future directions.

References

[1] Susanne Graf & Hassen Saïdi (1997): *Construction of Abstract State Graphs with PVS*. In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pp. 72–83, doi:10.1007/3-540-63166-6_10.

[2] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J. Strother Moore, Sandip Ray, José-Luis Ruiz-Reina, Rob Sumners, Daron Vroon & Matthew Wilding (2008): *Efficient execution in an automated reasoning environment*. *J. Funct. Program.* 18(1), pp. 15–46, doi:10.1017/S0956796807006338.

[3] Mitesh Jain & Panagiotis Manolios (2015): *Proving Skipping Refinement with ACL2s*. In: *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015.*, pp. 111–127, doi:10.4204/EPTCS.192.9.

[4] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic, doi:10.1007/978-1-4757-3188-0.

[5] Matt Kaufmann & Rob Sumners (2002): *Efficient rewriting of data structures in ACL2*. In Kaufmann M. Moore J.S. Borrione, D., editor: *Proceedings of 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*.

[6] Marcel Kyas & Jozef Hooman (2006): *Compositional Verification of Timed Components using PVS*. In: *Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik, 28.-31.3.2006 in Leipzig*, pp. 143–154.

[7] Leslie Lamport (1974): *A New Solution of Dijkstra's Concurrent Programming Problem*. *Commun. ACM* 17(8), pp. 453–455, doi:10.1145/361082.361093.

[8] Panagiotis Manolios, Kedar S. Namjoshi & Robert Summers (1999): *Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation*. In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pp. 369–379, doi:10.1007/3-540-48683-6_32.

[9] Sandip Ray & Rob Sumners (2011): *A theorem proving approach for verification of reactive concurrent programs*. In Chaudhury S. Farzan A. Gopalakrishnen G. Seigel S. Burckhardt, S., editor: *4th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC2 2011)*.

[10] Sandip Ray & Rob Sumners (2013): *Specification and Verification of Concurrent Programs Through Refinements*. *J. Autom. Reasoning* 51(3), pp. 241–280, doi:10.1007/s10817-012-9258-1.