Computing and Proving Well-founded Orderings through Finite Abstractions

Rob Sumners
Centaur Technology
rsumners@centtech.com

A common technique for checking properties of complex state machines is to build a finite abstraction then check the property on the abstract system — where a passing check on the abstract system is only transferred to the original system if the abstraction is proven to be representative. This approach does require the derivation or definition of the finite abstraction, but can avoid the need for complex invariant definition. For our work in checking progress of memory transactions in microprocessors, we need to prove that transactions in complex state machines always make progress to completion. As a part of this effort, we developed a process for computing a finite abstract graph of the target state machine along with annotations on whether certain measures decrease or not on arcs in the abstract graph. We then iteratively divide the abstract graph by splitting into strongly connected components and then building a measure for every node in the abstract graph which is ensured to be reducing on every transition of the original system guaranteeing progress. For finite state target systems (e.g. hardware designs), we present approaches for extracting the abstract graph efficiently using incremental SAT through GL and then the application of our process to check for progress. We present an implementation of the Bakery algorithm as an example application.

1 Introduction

In order to admit a recursive function to ACL2, the user must prove that the function terminates by showing that a function of the inputs exists which returns an ordinal (recognized by `o-p`) and which strictly decreases (by `o<`) on every recursive call of the function. The epsilon-0 ordinals recognized by `o-p` and ordered by `o<` are axiomatized in this way to be well-founded in ACL2. Our goal in this work is to present a new way to prove that certain relations are well-founded. Referring to Figure 1 and given a relation `(r x y)`, we produce a measure function `(m x)` and proofs of the properties `m-is-an-ordinal` and `m-is-o<-when-r`. This entails that the given relation `r` is well-founded.

```
(encapsulate (((r * *) => *) ((m *) => *))
    ...
  (defthm m-is-an-ordinal (o-p (m x)))
  (defthm m-is-o<-when-r  (implies (r x y) (o< (m y) (m x)))))

(defchoose choose-r (y) (x) (r x y))

(defun seq-r (x)  ;; any sequence of objects related by R must terminate
  (declare (xargs :measure (m x)))
  (if (r x (choose-r x)) (seq-r (choose-r x)) x))
```

Figure 1: Proving a relation is well-founded

In order to admit recursive functions, ACL2 has built-in heuristics for guessing appropriate measures and attempts to prove them. These heuristics often work for functions with common recursive patterns

G. Passmore, R. Gamboa (Eds.): ACL2 Workshop 2020
EPTCS 327, 2020, pp. 47–60, doi:10.4204/EPTCS.327.4

with user specification of measures covering the remaining cases. The theorem prover ACL2s [1] (the sedan) has a built-in procedure which builds so-called Calling Context Graphs (or CCGs) [7] and checks that there are no infinite paths through the CCG such that some measure doesn't decrease infinitely often while never increasing. The CCG checker in ACL2s significantly increases the number of functions which can be admitted without user specification of measures. Our work shares some similarities at a high level to the work on CCGs in ACL2s but the approach and target applications are quite different — we will cover these differences in greater detail in Section 7.

Our primary focus is proving well-founded relations `(r x y)` derived from software and hardware systems comprised of interacting state machines. In particular, for the work presented in this paper, we focus on systems defined as the composition of finite state machines. We present a procedure in this paper which takes the definition of `(r x y)`, a finite domain specification for `x` and `y` and a mapping from the concrete finite domain for `x` and `y` to an abstract domain. The procedure leverages existing bit-blasting tools in ACL2 to construct the abstraction of `r`, builds an abstract measure descriptor, and then translates this back to a proven measure on the concrete domain. In Section 2, we cover a version of the Bakery algorithm which will be the primary example for this paper. In Section 3, we present an overview of the procedure for generating and proving the needed measures and in Sections 4, 5, 6, we go into the details of the steps of the procedure along with demonstration via application to the example. We conclude the paper in Section 7 with a discussion of related work and future considerations.

2 Example: Bakery Algorithm

We will use a finite version of the Bakery algorithm as an example application throughout this paper. The Bakery algorithm was developed by Lamport [5] 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, 10].

The Bakery algorithm operates by allowing each process that wants exclusive access to first choose a number to get its position in line and then later compares the number against the numbers chosen by the other processes to determine who should have access to the exclusive section. The Bakery algorithm definition we will use for presenting this work is defined in Figure 2 where the macros `(tr+ ..)` and `(sh+ ..)` are shorthand for functions which update the specified fields of the `bake-tr-p` and `bake-sh-p` data structures.

The function `(bake-tr-next a sh)` takes a local bakery transaction state "`a`" and a shared state "`sh`" and updates the local bakery state. The function `(bake-sh-next sh a)` takes the same "`sh`" and "`a`" and produces the updated shared state (a single variable `sh.max` storing the next position in line). The function `(bake-tr-blok a b)` defines a blocking relation which denotes when one bakery process "`a`" is blocked by another bakery process "`b`".

Each task will start in program location 0 in which it starts its `a.choosing` phase. During the `a.choosing` phase, the task will grab the current shared max variable `sh.max` and then set its own position `a.pos` to be 1 more than `sh.max` — possibly wrapping around to a position of 0. If we do wrap the position around to 0, then the local Bakery process will cycle through the other Bakery processes that are completing to allow them to flush out before proceeding. After this check, the process will perform an atomic compare-and-update at program location (or `a.loc`) 6 to the shared `sh.max` variable and ends its `a.choosing` phase.

After the `a.choosing` phase, the process at `a.loc` 7 will enter another loop to check if it can proceed to the critical section. In locations 8, 9, and 10, the process checks if the current process we are checking

(at index `a.loop`) is either still choosing a position in line or is ahead of us in line (with ties broken by checks on the order of process indexes at location 10). Finally, after the process enters and exits the critical section at location 13, the process will decrement the `a.runs` outer loop count and either branch back to location 0 or complete and set `a.done` at location 17.

```
(define bake-tr-next ((a bake-tr-p) (sh bake-sh-p))
  (b* (((bake-tr a) a) ((bake-sh sh) sh))
    (case a.loc
      (0  (tr+ a :loc 1
               :choosing t))
      (1  (tr+ a :loc 2
               :temp sh.max))
      (2  (tr+ a :loc 3
               :pos (ctr-1+ a.temp) ;; can wrap to 0
               :loop (loop-start)))
      (3  (tr+ a :loc 4))  ;; possibly blocked here
      (4  (tr+ a :loc 5
               :loop (1- a.loop)))
      (5  (tr+ a :loc (if (= a.loop 0) 6 3)
               :pos-valid (= a.loop 0)))
      (6  (tr+ a :loc 7))  ;; update shared variable
      (7  (tr+ a :loc 8
               :choosing nil
               :loop (loop-start)))
      (8  (tr+ a :loc 9))  ;; possibly blocked here
      (9  (tr+ a :loc 10)) ;; possibly blocked here
      (10 (tr+ a :loc 11)) ;; possibly blocked here
      (11 (tr+ a :loc 12
               :loop (1- a.loop)))
      (12 (tr+ a :loc (if (= a.loop 0) 13 8)))
      (13 (tr+ a :loc 14   ;; critical section
               :pos-valid nil))
      (14 (tr+ a :loc 15
               :runs (1- a.runs)))
      (15 (tr+ a :loc (if (= a.runs 0) 16 0)))
      (t  (tr+ a :loc 17   ;; process is done
               :done t)))))

(define bake-sh-next ((sh bake-sh-p) (a bake-tr-p))
  (b* (((bake-tr a) a) ((bake-sh sh) sh))
    (case a.loc
      (6 (if (not (ctr-> sh.max a.temp)) ;; careful on wrap to 0
             (sh+ sh :max a.pos)
           sh))
      (t sh))))

(define bake-tr-blok ((a bake-tr-p) (b bake-tr-p))
  (b* (((bake-tr a) a) ((bake-tr b) b))
    (and (= a.loop b.ndx)
         (case a.loc
           (3  (and (= a.pos 0) b.pos-valid))
           (8  (and (not (= b.pos 0)) b.choosing))
           (9  (and b.pos-valid (< b.pos a.pos)))
           (10 (and b.pos-valid (= b.pos a.pos)
                    (< b.ndx a.ndx)))))))
```

Figure 2: Bakery Process Definitions

Our goal is to prove that a system comprised of some number of bakery processes updating asynchronously will eventually reach a state where all bakery processes are done. This is codified by admitting the function `(bake-run st)` defined in Figure 3. The `bake-run` function takes a state `st` consisting

of a list `st.trs` of transaction states (one for each bakery process) and a shared variable state `st.sh`. The function checks if all bakery processes are done (i.e. `(bake-all-done st.trs)`) and simply returns if so. Otherwise, the function chooses a process which is ready and updates the state for that process along with the shared state and recurs. The function `(choose-ready st.trs st.sh oracle)` is constrained (via `encapsulate`) to return an index for a bakery process state which is not done and is not blocked by any other process state (via `bake-tr-blok`). Given the function `choose-ready` is constrained to represent any legal input selection, then `bake-run` represents all legal bakery runs and its termination ensures that all runs end with all bakery processes done. We note that this does not prove the Bakery algorithm ensures mutual exclusion and it does not prove that the Bakery algorithm avoids livelock or starvation — these issues were covered in [10] but require more complex specifications involving infinite runs which are not closed-form in ACL2. The work presented in this paper to generate proven measures for well-founded relations has been applied to the more complete proof framework presented in [10].

```
(define bake-run (st orcl)
  (b* (((bake-st st) st))
    (if (bake-all-done st.trs)
        st
      (b* ((n   (choose-ready st.trs st.sh orcl))
           (a   (nth n st.trs))
           (trs (update-nth n (bake-tr-next a st.sh) st.trs))
           (sh  (bake-sh-next st.sh a)))
        (bake-run (make-bake-st :trs trs :sh sh) (next-oracle orcl))))))
```

Figure 3: Bakery System Run Function

3 Overview

Our goal is to define `bake-run` and admit it by proving its termination. In support of this goal, we need to prove that two relations are well-founded orderings. First, we need to build a measure showing that on updates with `bake-tr-next`, each bakery process makes progress to a done state. The other measure we need to define and prove is a little more subtle. The function `choose-ready` must return a process index with a state which is not done and is not blocked. In the case of a deadlock between some number of process states (a cycle of the `bake-tr-blok` relation), choosing an unblocked process may not be possible. We need to build a measure showing that no blocking cycles exist between states; this measure will allow us to define a function which always finds an unblocked process state. In each of these cases, we begin with a relation `(r x y)` that we want to show is well-founded requiring the definition of a measure `(m x)` which preserves the properties `m-is-an-ordinal` and `m-is-o<-when-r` from Figure 1.

A standard ACL2 proof that these relations are well-founded would not only require defining a decreasing measure `(m x)`, but in addition and invariably, invariant properties of the reachable process states. In order to prove these invariants, the user will need to strengthen the invariant definitions to be inductive. For complex systems, the definition of inductive invariants can be prohibitively expensive. In addition, inductive invariant definitions are fragile and require maintenance when system definition changes. The same is also true of defining measures for proving well-founded relations in complex systems — these measure definitions can also be extensive and brittle to changes in system definition. Our goal is to build a procedure which allows the generation of inductive invariants as well as decreasing

1. **Bakery algorithm definition** :
   - file: `bakery.lisp`
     - Defines the types and functions for the Bakery algorithm process states.
     - Defines `GL` shape specifiers and theorems to connect `GL` shapes to the types.
2. **Building abstract models with `GL` and Incremental SAT** :
   - file: `gl-fin-set.lisp`
     - Defines function for computing possible values for a term with finite variable bindings.
     - Uses `GL` functions to translate the term to CNF formula.
     - Uses `IPASIR` (Incremental SAT) to efficiently find a number of values resulting from the given term.
   - file: `gen-models.lisp`
     - Defines reachable abstract graph builders using the functions from `gl-fin-set.lisp`.
     - Defines functions to tag these graphs with which orderings decrease along the arcs.
3. **Building measure descriptors from abstract models** :
   - file: `cycle-check.lisp`
     - Defines graph algorithms for processing an abstract graph with ordering tags which produces either a cycle in the graph for a failing case or a measure descriptor in a passing case.
   - file: `bake-models.lisp`
     - Calls abstract model generation from `gen-models.lisp` to build abstract models for the Bakery example.
     - Calls graph algorithms from `cycle-check.lisp` to build measure descriptors for the Bakery example.
4. **Building proven measures from measure descriptors** :
   - file: `wfo-thry.lisp`
     - Builds a theory relating properties of the abstract model and generated measure descriptor to proving a measure.
     - Uses theory of natural number lists and lists of natural lists defined in `bounded-ords.lisp`.
   - file: `bake-proofs.lisp`
     - Instantiates theory from `wfo-thry.lisp` with abstract models and measure descriptors from `bake-models` to build proofs of the needed measures.
     - Also proves some auxiliary properties needed for Bakery proof.
5. **Admitting `bake-run` using proven measures** :
   - file: `top.lisp`
     - Uses proven measures from `bake-proofs.lisp` to define `choose-ready` and build a measure to admit `bake-run`

Figure 4: Overview of the Procedure

measures which prove our target relations to be well-founded. Computing invariant and measure definitions not only has the benefits of requiring less human definition and tracking design change, but can also provide more direct debug output from failed attempts. We provide an overview of the steps in our procedure (as well as pointers to where these steps are defined in the supporting materials) in Figure 4. We cover each of these steps in greater detail in the remaining sections of the paper as well as covering their application to our Bakery algorithm example.

4 Building Models with GL and Incremental SAT

The tool `GL` [12] is an extension to ACL2 (primarily an untrusted clause processor) which targets proving theorems on finite domains by translating the theorems to boolean formulas using symbolic simulation and then checking the boolean formulas through BDDs or SAT with boolean AIG transformations and simplifications. There are different ways to direct `GL` to translate a term to a boolean formula, but the most basic form is to take a `hyp` term and `concl` term along with a shape specification `g-bindings` for the free variables in the `hyp` and `concl` terms. `GL` uses the shape specification to provide a symbolic value for the free variables and then symbolically evaluates the `concl` and checks if the resulting boolean formula is valid. If the boolean check passes, then `GL` checks that the `hyp` implies the constraints specified by the shape specification. The first step in our procedure uses the setup for `GL` but instead of proving that a term is always true, we will instead compute the set of values that a term can return under evaluation with variable bindings consistent with `g-bindings`.

We define the function `(compute-finite-values trm hyp g-b num)` which takes terms `trm` and `hyp`, shape specification `g-b`, and natural number `num` and attempts to return (up-to `num`) values from the set of possible values for `trm` under the assumption of `hyp` with free variables consistent with `g-b`. The function `compute-finite-values` will also return a boolean `is-total` which is true if the list of values returned is the entire set of possible values. We use the `GL` symbolic evaluation functions to provide a translation from terms (with shape-specifications) to boolean CNF formula and then iterate through the set of possible boolean values discovered using incremental SAT via the `IPASIR` library [11]. The resulting boolean valuations from repeated IPASIR tests are then translated back to ACL2 objects and returned.

This inner IPASIR Incremental SAT loop begins with installation of the CNF formula (from the GL translation) into the IPASIR clause database. The literals in the CNF formula corresponding to the output of `trm` are also recorded. Then, within each iteration of the loop, we first call IPASIR to find a satisfying assignment. If it is unsatisfiable then there are no more values and we return. Otherwise, we retrieve the boolean values for the `trm` literals from IPASIR and add this boolean valuation to our accumulated return set. We then add the negation of the equality of the `trm` literals to the retrieved boolean values as a new clause in the IPASIR database and iterate through the loop. We terminate the loop by either reaching an unsatisfiable IPASIR instance or exceeding the user-specified maximum number `num` of values. The chief benefit of using incremental SAT is the amortization of the translation and installation of the CNF formula along with (and more importantly) the incremental benefits of any learned clauses that the SAT solver determines through each iteration of this loop. [1]

Using the `compute-finite-values` function, we construct an abstract graph from the concrete

[1] We note that independent of the work presented in this paper, a new revision of `GL`, named `FGL`, was added to ACL2 and integrates incremental SAT in addition to other features. In particular, `FGL` makes it much easier and direct to define exploration functions like `compute-finite-values` using rewrite rules, but we decided to present the approach in `GL` given that `GL` is more familiar to the ACL2 community at the time of the writing of this paper.

```
(define bake-rank-map ((a bake-tr-p))
  (b* (((bake-tr a) a))
    `((:loc    ,a.loc)
      (:done   ,a.done)
      (:loop=0 ,(equal a.loop 0))
      (:runs=0 ,(equal a.runs 0))
      (:inv    ,(and (>= a.loop 0)
                     (>= a.runs 0))))))

(defconsts (*bake-rank-reach* state)
  (comp-map-reach :init-hyp 't
                  :init-trm '(bake-rank-map (bake-tr-init n r))
                  :step-hyp '(and (equal (bake-rank-map a) ,*src-var*)
                                  (not (bake-tr-done a)))
                  :step-trm '(bake-rank-map (bake-tr-next a sh))))
```

Figure 5: Bakery state mapping and reachable graph construction

system definition. The function `(comp-map-reach init-hyp init-trm step-hyp step-trm)` in `gen-models.lisp` builds the reachable graph beginning with the set of values from `init-trm` and iteratively reached by steps in `step-trm`. Returning to the Bakery example, Figure 5 defines the mapping `(bake-rank-map a)` taking a bakery process state and returning the state information needed to build a measure of progress to a done state — or, intuitively, the mapping `bake-rank-map` includes enough state information to ensure a bakery process makes progress to a done state. This includes the current location `a.loc`, whether or not we are done `a.done`, whether or not the inner loop or outer loop variables have counted down to 0, and then a predicate ensuring that the `a.loop` and `a.runs` counters are natural. This last `:inv` predicate field is actually an inductive invariant attached to the abstract state and we include it in the abstract state to effectively prove and use this inductive invariant during the building of the abstract graph and later, when we add ordering information to the graph.

The `comp-map-reach` function in Figure 5 builds the abstract reachable graph by setting up calls to `compute-finite-values`. It first calls `compute-finite-values` to return the set of values for `(bake-rank-map (bake-tr-init n r))` where `n` is defined to be an index for the process state and `r` defines the number of runs or number of iterations of the outer `bake-tr-next` loop. The function `comp-map-reach` builds a shape specification based on the variables in the terms and then computes the initial states. The function `comp-map-reach` will then iterate by computing the values for the step term `(bake-rank-map (bake-tr-next a sh))` at each node with the hypothesis `(equal (bake-rank-map a) ,*src-var*)` — during each step, the special variable `*src-var*` is bound to the value of the current node in the reachable graph exploration (i.e. a reached result of `bake-rank-map`).

The result of `comp-map-reach` is a graph defined as an alist where each pair in the alist associates a node to a list of nodes which form the directed arcs — the nodes are the results of `bake-rank-map` for reachable bakery process states. The nodes in the `*bake-rank-reach*` graph are included in Figure 6. There are 21 nodes consisting of 1 node per location and 2 nodes for locations 5, 12, and 15. The extra nodes for these locations is due to a split based on whether `a.loop` is equal to 0 in locations 5 and 12 and whether `a.runs` is equal to 0 in location 15.

The reachable abstract graph for `bake-rank-map` is not sufficient to build a measure of progress to a done state — there are 3 backward arcs at locations 5, 12, and 15. We could solve this by adding the full values for `a.loop` and `a.runs` to `bake-rank-map` but this would dramatically increase the number of nodes in the resulting abstract graph and is clearly not viable in general. The better approach is to tag arcs in the abstract graph with whether or not certain measures strictly decrease or possibly increase. This ordering information is defined by the function `(bake-rank-ord a o)` and added via the function

```
ACL2 !>(strip-cars *bake-rank-reach*)
(((:LOC 0)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 1)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 2)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 3)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 4)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 5)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 5)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 6)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 7)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 8)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 9)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 10) (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 11) (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 12) (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T))
 ((:LOC 12) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 13) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 14) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 15) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T))
 ((:LOC 15) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 T)   (:INV T))
 ((:LOC 16) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 T)   (:INV T))
 ((:LOC 17) (:DONE T)   (:LOOP=0 T)   (:RUNS=0 T)   (:INV T)))
```

Figure 6: Nodes in reachable abstract graph from `comp-map-reach`

```
(define bake-rank-ord ((a bake-tr-p) (o ord-p))
  (b* (((bake-tr a) a))
    (cond ((eq o 'runs) (nfix a.runs))
          ((eq o 'loop) (nfix a.loop))
          (t 0))))

(defconsts (*bake-rank-ord-graph* state)
  (comp-map-order :reach    *bake-rank-reach*
                  :ordr-hyp `(and (equal (bake-rank-map a) ,*src-var*)
                                  (equal (bake-rank-map b) ,*dst-var*)
                                  (equal (bake-tr-next a sh) b))
                  :ordr-trms (make-ordr-trms *bake-rank-ords*
                                             'bake-rank-ord 'a 'b)))
```

Figure 7: Bakery component measures and ordering tag construction

comp-map-order in Figure 7. The function bake-rank-ord takes a bakery state and symbol identifying a component measure and returns the measure value (natural values in this case but in general can be a list of natural numbers). The function comp-map-order takes the reachable abstract graph and computes (using compute-finite-values) tags for each arc in the reachable graph encoding whether the specified component measure is either strictly-decreasing, not-increasing, or possibly-increasing along that arc. The runs measure strictly decreases on the arcs from the node at location 14 to the nodes at location 15 and is non-increasing on all arcs. The loop measure strictly decreases on arcs 4 to 5 and from 11 to 12, increases on arcs 2 to 3 and 7 to 8 and is non-increasing on all other arcs. This tagged reachable graph is used in the next section to compute a measure descriptor covering the concrete relation used to build the graph — in this case, the next-state bakery function bake-tr-next.

5 Building Measures with SCC decomposition

In the previous section, we used GL and IPASIR to construct abstract reachable graphs with arcs tagged based on which component measures decreased or increased. The next step in our procedure is to use an algorithm based on the decomposition of strongly connected components (SCCs) to build an object describing how to build a full measure across the concrete relation represented by the abstract graph. The algorithm consists of two alternating phases operating on subgraphs of the original graph (starting with the original graph itself) and produces a mapping of the nodes in the graph to a measure descriptor which is a list comprised of symbols (representing a component measure) or natural numbers.

- if the current subgraph is an SCC:
  1. search for a component measure which never increases and decreases at least once.
  2. if no such component measure is found, then find the minimal non-decreasing cycle and fail.
  3. otherwise, remove the component measure's decreasing arcs from the graph and recur.
  4. cons the component measure onto the measure descriptors from the recursive calls.
- otherwise:
  1. partition the graph into SCCs using a standard algorithm.
  2. recursively build measure descriptors for each of the SCCs.
  3. build the directed acyclic graph of SCCs and enumerate the SCCs in the graph.
  4. cons the enumeration of each SCC onto the measure descriptors from the recursive calls.

Returning to the Bakery algorithm example, the resulting alist associating abstract graph nodes to measure descriptors produced by this algorithm is given in Figure 8. The measure descriptors follow the control flow through the locations of the bakery process state. The outer loop forms an SCC with the runs component measure breaking the arc from location 14 to 15. Within the outer loop, the inner loops form SCCs with the loop component measure decreasing to break the SCCs further. The measure descriptor that results allows us to build a measure showing that the relation (and (equal y (bake-tr-next x sh)) (not (bake-tr-done x))) is well-founded. In the next section, we cover the theory and its instantiation which allows the transfer of these measure descriptor results into actual proofs of well-founded relations.

```
;;                (<abstract node -- reachable bake-rank-map>) . (<measure-descriptor>)
;;                ------------------------------------------------------------------
((((:LOC 0)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 11 0))
 (((:LOC 1)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 10 0))
 (((:LOC 2)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 9  0))
 (((:LOC 3)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 8  LOOP 2 0))
 (((:LOC 4)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 8  LOOP 1 0))
 (((:LOC 5)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 8  LOOP 3 0))
 (((:LOC 5)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 7  0))
 (((:LOC 6)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 6  0))
 (((:LOC 7)  (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 5  0))
 (((:LOC 8)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 4  LOOP 4 0))
 (((:LOC 9)  (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 4  LOOP 3 0))
 (((:LOC 10) (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 4  LOOP 2 0))
 (((:LOC 11) (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 4  LOOP 1 0))
 (((:LOC 12) (:DONE NIL) (:LOOP=0 NIL) (:RUNS=0 NIL) (:INV T)) . (4 RUNS 4  LOOP 5 0))
 (((:LOC 12) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 3  0))
 (((:LOC 13) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 2  0))
 (((:LOC 14) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 1  0))
 (((:LOC 15) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 NIL) (:INV T)) . (4 RUNS 12 0))
 (((:LOC 15) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 T)   (:INV T)) . (3 0))
 (((:LOC 16) (:DONE NIL) (:LOOP=0 T)   (:RUNS=0 T)   (:INV T)) . (2 0))
 (((:LOC 17) (:DONE T)   (:LOOP=0 T)   (:RUNS=0 T)   (:INV T)) . (1 0)))
```

Figure 8: Result of computing measure descriptors for `bake-rank-map`

```
(encapsulate
  (((test-rel-p * *)         => *)
   ((test-map-e *)           => *)
   ((test-map-o * *)         => *)
   ((test-o-bnd)             => *)
   ((test-a-dom)             => *)
   ((test-nexts *)           => *)
   ((test-chk-ord-arc * * *) => *))

  ... <local defuns> ...

  (def-valid-wf-corr-assumption test) ;; macro generating assumptions with "test-" prefix
)
(def-valid-wf-corr-conclusion test)   ;; macro generating derivations with "test-" prefix
```

Figure 9: "Theory" for proving generated measures

6 Proving the Generated Measures and Wrapping Up

In the book `wfo-thry.lisp` from the supporting materials, a "theory" is developed connecting the successful computation of a measure descriptor from the abstract graph to the definition of a measure proving that the concrete relation is well-founded. The structure of this book is essentially the definition of two macros — one macro codifies the assumptions made of a set of definitions and a second macro generates the conclusions and results derived from these assumptions. Each macro takes a name prefix parameter which is prepended to all of the definition and theorem names generated in the macro. [2] The end of the `wfo-thry` book concludes with the forms in Figure 9. The function `(rel-p x y)` is the relation we want to prove is well-founded. The function `(map-e x)` is essentially equivalent to the `bake-rank-map` function from our example and `(map-o x o)` is the `bake-rank-ord` function. The constant `(a-dom)` is the set of nodes in the abstract graph (as in Figure 6) and `(nexts x)` is a

[2] It is worth noting that this would be better carried out with functional instantiation in ACL2, but due to technical issues, that has not worked in all cases — we are working on rectifying this.

function taking a node in `(a-dom)` and returning a list of successor nodes pulled from the abstract graph. The function `(chk-ord-arc x y o)` takes two nodes in the abstract graph `x` and `y` and a component measure name `o` and returns either `:<<` if the measure is strictly-decreasing, `t` if it is non-increasing, and `nil` if it is possibly-increasing.

```
(defthm map-e-member-nexts
  (implies (rel-p x y)
           (in-p (map-e y) (nexts (map-e x)))))

(defthm map-o-decrement-strict
  (implies (and (rel-p x y)
                (equal (chk-ord-arc (map-e x) (map-e y) o) :<<))
           (bnl< (map-o y o) (map-o x o) (o-bnd))))

(defthm map-o-decrement-non-strict
  (implies (and (rel-p x y)
                (equal (chk-ord-arc (map-e x) (map-e y) o) t))
           (bnl<= (map-o y o) (map-o x o) (o-bnd))))
```

Figure 10: Assumptions for proving generated measures

The main theorems assumed about these functions are provided in Figure 10. These assumed properties provide the correlation between `(rel-p x y)` and the abstract graph defined by `(nexts x)` with the tagging of component measure ordering on the arcs defined by `chk-ord-arc`. The relation `(bnl< m n o)` is defined in the book `bounded-ords.lisp` and orders lists of naturals `m` and `n` of length `o` as the lexicographic product of the naturals in the list in order. The `bounded-ords` book defines functions and relations for building and ordering lists of naturals (of the same length) and lists of natural lists (potentially differing lengths). These are recognized as `bnlp` and `bnllp` and ordered with `bnl<` and `bnll<` respectively. As an aside, we note that in the supporting books for the paper, we use `bplp` instead of `bnlp` where `bplp` is the same as `bnlp` but requires the last natural in the list to not be 0. The reason to use `bplp` is to allow a list of all 0s to be used as a bottom element in certain constructions. We use `bnl` and `bnll` throughout the paper for clarity.

There are also conversion functions `bnl->o` and `bnll->o` for converting `bnl`s and `bnll`s to ACL2 ordinals preserving the well-founded ordering `o<` on ACL2 ordinals. We use the bounded ordinals from `bounded-ords` instead of ACL2 ordinals because these bounded ordinals are closed under lexicographic product while ACL2 ordinals are not. This allows us to build constructions using these bounded ordinals that would be far more difficult (and in some cases, not even possible) with ACL2 ordinals.

```
(defthm msr-is-o-p (o-p (msr x m)))

(defthm relp-well-founded
  (implies (and (valid-omap m)
                (rel-p x y))
           (o< (msr y m) (msr x m)))

(defthm mk-bnl-is-bnlp
  (implies (valid-omap m)
           (bnlp (mk-bnl x m) (bnl-bnd m)))

(defthm mk-bnl-transfers-rel-p-bnl<
  (implies (and (valid-omap m)
                (rel-p x y))
           (bnl< (mk-bnl y m) (mk-bnl x m) (bnl-bnd m))))
```

Figure 11: Derivations for proving generated measures

Given the assumptions from Figure 10 (and some additional typing assumptions), we generate a measure function `(msr x m)` which takes an ACL2 object `x` and a measure descriptor mapping produced from the SCC decomposition (as in Figure 8 and termed an `omap`) and produces an ACL2 ordinal. If the mapping `m` satisfies the generated check `(valid-omap m)`, then the `(msr x m)` returns a strictly decreasing ordinal which shows that `(rel-p x y)` is well-founded. The key derived properties generated from the instantiation of this theory are provided in Figure 11. In addition to generating the definition of a measure function returning ACL2 ordinals, we also generate a definition producing a bounded ordinal `bnlp` and related properties. The intent is to use the `mk-bnl` function when one wants to use the generated ordinal in a composition to build larger ordinals — even potentially using the procedure in this paper hierarchically where the component measure at one level is a proven generated measure at a lower level in the hierarchy.

Returning to the Bakery example, the book `bake-proofs` includes the generated abstract graphs and measure descriptor mappings (or `omaps`) from `bake-models` and sets up instantiations of the "theory" from the `wfo-thry` book. The result is the generated measures for proving our target two relations are well-founded: the relation defined by the step function `bake-tr-next` and the blocking relation `bake-tr-blok`. In the book `top.lisp`, we use these results to reach our goal of defining and admitting the `bake-run` function from Figure 3. We use the generated measure function `bank-rank-mk-bnl` to define the function `(bake-rank-bnll l sh)` which conses the `bnl`s for each bakery process state in the list `l` and returns a `bnll`. The measure we use for admitting `(bake-run st orcl)` is the conversion of the resulting `bnll` to ACL2 ordinals:

```
(bnll->o (len (bake-st->trs st))
         (bake-rank-bnll (bake-st->trs st)
                         (bake-st->sh st))
         (bake-rank-bnl-bnd))
```

Additionally, as we noted earlier in Section 2, we need the generated measure for proving `bake-tr-blok` is well-founded in order to define `choose-ready` correctly. In particular, `(choose-ready l sh o)` is a constrained function which ensures that if there is a bakery state which is not done in `l`, then `(choose-ready l sh o)` will return the index of a state which is not done and not blocked. The local witness in the encapsulation for `choose-ready` is:

```
(local (defun choose-ready (l sh o)
         (find-unblok (find-undone l) l sh)))
```

Where `(find-undone l)` returns an index in `l` for a bakery process which is not done (if one exists) and the function `(find-unblok n l sh)` takes an index `n` and finds an index which is not blocked in `l`. The function `find-unblok` is (essentially) defined as:

```
(define find-unblok ((n natp) (l bake-tr-lst-p) (sh bake-sh-p))
  (if (bake-blok (nth n l) l)
      (find-unblok (pick-blok (nth n l) l) l sh)
    n))
```

Where `(bake-blok a l)` returns true if any state in `l` blocks `a` and `(pick-blok a l)` finds an index in `l` for a bakery state which blocks `a` (and thus if `(bake-blok a l)` then `(bake-tr-blok a (nth (pick-blok a l) l))`). The measure used to admit `find-unblok` is defined using the generated measure `bake-nlock-msr` for proving the `bake-tr-blok` relation is well-founded.

An additional important property of `(find-unblok n l sh)` is that it either returns `n` (if `(nth n l)` is not blocked) or it returns an index for a process state which is blocking another state. We prove separately that no process in a done state can block another process and thus, if the state at index `n` passed to `find-unblok` is not in a done state, then the state at the index returned by `find-unblok` is also not in a done state.

7 Related Work and Future Work

The analysis of abstract reachable graphs with ordering tags is similar in some ways to analysis of automata on infinite words [4], but our search for a measure construction is not the same as language emptiness or other checks typically considered for infinite word or tree automata. As we mentioned in Section 1, our work does share similarity to the work on CCGs in ACL2s in that both build and analyze graphs with the goal of showing that no "bad" infinite paths exist through the graph — but the focus and approach of each work is significantly different. CCG termination analysis in ACL2s is used to determine if a user-specified function always terminates. A significant component of the CCG analysis is unwinding and transforming CCGs (and CCMs) until one can no longer find bad paths and thus ensure termination. The problem that CCG analysis attempts to tackle is intrinsically more difficult than the problems we target. We attempt to prove a given relation is well-founded and rely on mapping function definitions to build a closed model sufficient to then find a proven measure. Our procedure aggressively builds the model as specified by the mapping functions and proceeds assuming it is sufficient without further refinement – the user or outside heuristics are responsible for any further refinements. The AProVE program analysis tool [2] provides mechanisms for automatically checking program termination. The approach taken in AProVE is to translate the source program into a term rewriting system and apply a variety of analysis engines from direct analysis of the term rewriting to translation into checks for SAT or SMT. Similar to CCG analysis in ACL2s, the primary difference between AProVE and our work is an issue of focus. The contexts we target benefit from the assumption of mapping functions and (in the case of this paper) finite-state systems which can be processed by `GL`. This allows us to build the tagged abstract reachable graph directly with less reliance on having sufficient rewrite rules and term-level analysis and heuristics.

There are many ways to extend the work presented. We would like to add an interface into `SMTLINK` [8] for either building the abstract graphs and/or the addition of ordering tags to the arcs in the graphs. `SMTLINK` is more limited than `GL` in what ACL2 definitions it can support, but `SMTLINK` would be a nice option to have in the cases where the definitions were viable for `SMTLINK`. While we assumed the definition of mapping functions and component measures for the sake of this paper, it is not difficult to write heuristics to generate candidate mappings and measures either from datatype specifications in the source definition or from static analysis results. Further, the results and steps in the process can be analyzed to determine which refinements to apply to the mapping and component measures in an "outer loop" to our procedure. From our limited experience, this is best addressed with guidance from the domain of application and the types of relations and systems that the user wants to analyze.

References

[1] P. Dillinger, P. Manolios, D. Vroon & J.S. Moore (2007): *ACL2s: The ACL2 sedan*. *Electronic Notes in Theoretical Computer Science - ENTCS* 174, doi:10.1016/j.entcs.2006.09.018.

[2] J. Giesl, C. Aschermann, M. Brockschmidt & et al. (2017): *Analyzing Program Termination and Complexity Automatically with AProVE*. *Journal of Automated Reasoning* 58, doi:10.1007/s10817-016-9388-y.

[3] M. Kaufmann, P. Manolios & J.S. Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic, doi:10.1109/32.588534.

[4] O. Kupferman (2018): *Automata Theory and Model Checking*. *Handbook of Model Checking (2018)*, doi:10.1007/978-3-319-10575-8_4.

[5] L. Lamport (1974): *A new solution of Dijkstras concurrent programming problem*. *Communications of the ACM* 17(8), doi:10.1145/3335772.3335782.

[6] P. Manolios, K. Namjoshi & R. Sumners (1999): *Linking model-checking and theorem-proving with well-founded bisimulations*. *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1999)* 1633, doi:10.1007/3-540-48683-6_32.

[7] P. Manolios & D. Vroon (2006): *Termination Analysis with Calling Context Graphs*. *Proceedings of the 18th International Conference on Computer-Aided Verification (CAV 2006)*, doi:10.1007/11817963_36.

[8] Y. Peng & M. Greenstreet (2018): *Smtlink 2.0*. *Proceedings of 15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2018)*, doi:10.4204/EPTCS.280.11.

[9] S. Ray & R. Sumners (2013): *Specification and Verification of Concurrent Programs Through Refinements*. *Journal of Automated Reasoning* 51(3), doi:10.1007/s10817-012-9258-1.

[10] R. Sumners (2017): *Proof Reduction of Fair Stuttering Refinement of Asynchronous Systems and Applications*. *Proceedings of 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2017)*, doi:10.4204/EPTCS.249.6.

[11] S. Swords (2018): *Incremental SAT Library Integration Using Abstract Stobjs*. *Proceedings of 15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2018)*, doi:10.4204/EPTCS.280.4.

[12] S. Swords & J. Davis (2011): *Bit-Blasting ACL2 Theorems*. *Proceedings of 10th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2011)*, doi:10.4204/EPTCS.70.7.