Reducing Invariant Proofs To Finite Search via Rewriting*

Rob Sumners
Advanced Micro Devices (AMD) Inc.
robert.sumners@amd.com

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
sandip@cs.utexas.edu

Abstract

We present a tool-supported methodology for proving predicates invariant over all time for any behavior of a given system. We prove invariants by exploring a finite graph generated from the definition of the predicate using rewrite rules from proven ACL2 theorems. The methodology provides a means for proving invariants which avoids the complexity and cost of defining an inductive invariant while still allowing the proof of invariants for reactive systems modeled in an expressive language. We present two examples of the application of the methodology: a simple critical section example, and a slightly more complex ESI cache coherence model.

1 Motivation

A common problem encountered in the formal proof of correctness of reactive systems (i.e. systems which continuously operate through an ongoing interaction with an environment) is the need to prove a certain target predicate is invariant over time. These proofs generally require the definition and proof of an *inductive invariant* which is a predicate which implies the target invariant, and can be proven to be true by simple induction over time. If the reactive system is effectively defined by a finite number of reachable states, then a model checker may be used to explore the reachable states to determine if a state exists which does not satisfy the target invariant. If a model checker is effective, then the invariant proof process is automatic with the additional benefit of concrete counterexample behaviors when the target invariant is not valid. If a model checker is not effective, then a theorem prover is generally used to define and prove a sufficient inductive invariant and the process requires far more user guidance and interaction with the primary requirement of the user being to define a sufficient inductive invariant. For complex reactive systems, the definition of a sufficient inductive invariant can be prohibitively expensive. Further, inductive invariant proofs are brittle in that as the design and definition of the reactive system evolves over time, the definition and proof of the inductive invariant may require extensive modification in order to accommodate the changes in system definition.

We attempt to bridge this gap of automation between theorem proving and model checking while still affording the ability to prove invariants of systems modeled in an expressive language. We have developed a tool which we term an *invariant prover* to support our methodology. The prover operates primarily by using rewrite rules (extended as needed by the user from theorems proven with ACL2) to transform the invariant proof into a reachability query on a finite-state graph. This reachability test is then performed using an efficient breadth-first search and if no states are found which violate the target invariant, then the target invariant is true at all times. If the reachability test fails, then a pruned failure witness is reported to the user. Even if a failure is found, the target invariant may still be valid and it is simply the case (similar to a theorem prover), the user needs to provide rewrite rules which guide the invariant prover to the generation of a more accurate finite-state graph. Thus, it is important that the user has sufficient control of and feedback

*Support for this work was provided in part by the SRC under contract 02-TJ-1032.

from the invariant prover in order to determine the appropriate input to provide the tool to lead it to a proof of the target invariant.

Our approach shares many common traits with model checking approaches based on abstraction whereby the reachability question for a system with a large number of states is reduced to a system with a small number of states. Previous research into abstract model checking fell primarily into two categories. The first category consists of approaches where the reduced model was generated automatically from the larger system exploiting properties about the structure of the states and transitions in the larger system (e.g. state symmetries [5]). This category of tools are essentially automatic, but will enforce limitations on how the systems are defined or the possible reduction in the number of states. The other category of approaches involve the use of theorem provers to prove the correlation between the larger system and the abstract model (e.g. stuttering bisimulation proofs [9]). Approaches in this category will require the definition of the abstract model (perhaps in terms of the larger system) and an interactive proof of the correspondence between the two systems (which may require the proof of an inductive invariant). Our approach can be seen as a hybrid of these two categories. Similar to the first category of approaches, we define a tool which automatically generates an abstract model from the larger system which is guaranteed to correspond with the larger system by construction. Similar to the second category of approaches, we make no restrictions on the definitions of systems and allow the tool to be extended through the interaction with a theorem prover.

The key to the proposed hybrid approach is to use rewrite rules proven by the user to generate the abstract model and thus we attempt to transfer the responsibility of the user from defining inductive invariants to carefully designing and proving rewrite rules. How does this transfer alleviate the problem of the user interaction required in deductive verification? While the design and proof of rewrite rules involves thoughtful interaction with a theorem prover, the rewrite rules which are required are generally not specific to a particular system but instead to the operators used in defining the system and can be used in the invariant proofs of other systems written using the same set of operators. It is the case that particular systems may require a rewrite rule to force a certain case-split or introduce a more manageable term, but the concepts driving even the definition of these special rules can be transferred from one system to another. We imagine that most of the rewrite rules which a user will require will be generic (although potentially different from the rules needed for an ACL2 proof) and can be added to a library of rules which can be reused for other systems built from the same set of operators. It is nonetheless important that the feedback from the invariant prover highlights the limitations in the current rule set and helps guide the user towards the necessary additional rules which are needed. It is also important to note that our approach benefits from the interaction with a theorem prover such as ACL2 in that the user may extend the invariant prover by defining new functions and operations and proving rewrite rules in the ACL2 logic. This approach avoids the possible introduction of soundness errors in an ad hoc extension of the prover. We note that while the invariant prover is designed to interface with the ACL2 theorem prover, we imagine a similar tool could be developed to interface with other general-purpose interactive theorem provers.

The remainder of the paper is organized as follows. In Section 2, we present an initial example application of the invariant prover. In Section 3, we present some details on the function of the invariant prover and its interface to ACL2. In Section 4, we present a more involved example based on the ESI cache coherence protocol. Finally, we conclude the paper in Section 5.

2 Initial Example

We begin with an invariant proof for a simple system which implements mutual exclusion for access to critical code sections. The simple system is defined via the `define-system` macro in Figure 2. The `define-system` macro takes a list of parameterized state variable definitions `(<var> (<args>) <init> <next>)` and generates a `mutual-recursion` form with a function for each `<var>` of the following form `(defun <var> (<args>) (if (zp n) <init> (let ((n- (1- n))) <body>)))`. In the example system in Figure 2, the `define-system` macro will create three functions `in-critical`, `critical-id`, and `status` where each function takes a natural-valued time parameter `n`. The function `status` takes an additional process identifier parameter `p` and defines the "state" of the process `p` at time `n` which cycles between the values

```
(encapsulate (((i *) => *)) (local (defun i (n) n)))

(define-system critical
 (in-critical (n) nil
   (if (in-critical n-)
       (/= (i n) (critical-id n-))
     (= (status (i n) n-) :try)))

 (critical-id (n) nil
   (if (and (not (in-critical n-))
            (= (status (i n) n-) :try))
       (i n)
     (critical-id n-)))

 (status (p n) :idle
   (if (/= (i n) p) (status p n-)
     (case (status p n-)
           (:try (if (in-critical n-) :try :critical))
           (:critical :idle)
           (t :try)))))
```

Figure 1: Simple Critical Section example system

`:idle`, `:try`, and `:critical`. The function `in-critical` is a boolean which is true when some process is in the critical section and `critical-id` is the process identifier of the process in the critical section (i.e. in the `:critical` state). The constrained function `(i n)` takes a time parameter `n` and returns the process which is selected at time `n` to take the next step. This function `(i n)` represents an arbitrary stimulus from an "environment" and in this case models an asynchronous composition of the processes interacting to maintain mutual exclusion.

The specification of mutual exclusion is the requirement that no two processes can be in the `:critical` state at the same time. This specification is codified in ACL2 by the function `(ok n)` which is defined in Figure 2. The constrained functions `(a)` and `(b)` define arbitrary process identifiers which are guaranteed to not be `equal`. This use of constrained functions is important in order to provide a means for dividing up certain parameters of a system into a finite number of key values. In the case of the critical section example, the `status` and `critical-id` components of the state are infinite and the prover will generate a finite model focusing on the process identifiers `(a)` and `(b)`. The goal of the invariant prover is to prove that for all times `n`, `(ok n)` evaluates to a non-`nil` value.

Before we consider how the invariant prover functions, we consider how a user would normally prove that `(ok n)` is true for all `n`. As we mentioned before, the normal process involves the definition of an inductive invariant which implies the target invariant and implies itself at the next time. A sufficient inductive invariant for the mutual exclusion example is defined by the predicate `ii-ok` in Figure 3 and the theorem `ok-is-invariant` (which is automatically proven by ACL2) ensures that `(ok n)` is true for all `n`.

We now provide a brief overview of how the invariant prover functions. Essentially, the prover consists of two main phases: first, the prover constructs a finite set of state predicates and second, the prover explores the finite state graph defined by this finite set of predicates to determine if the invariant is proven. The first phase computes a set of predicates beginning with the singleton set consisting of the predicate `(ok n)` and iteratively extending this set with predicates encountered in the rewriting of each predicate in the set with `n` replaced by `(1+ n)` – we call this rewritten `(1+ n)` term, the *next-state term*. So, we begin by rewriting `(ok (1+ n))` which reduces to the term `(if (equal (status (a) n) :critical) (if (equal (status (b) n) :critical) nil t) t)` and from this term, two new predicates `(equal (status (a) n) :critical)`

```
(encapsulate (((a) => *) ((b) => *))
  (local (defun a () 1)) (local (defun b () 2))
  (defthm a-/=-b (not (equal (a) (b))))
  (defthm a-non-nil (not (equal (a) nil)))
  (defthm b-non-nil (not (equal (b) nil))))

(define-system critical-spec
 (ok (n) t
   (not (and (= (status (a) n-) :critical)
             (= (status (b) n-) :critical)))))
```

Figure 2: Specification for Simple Critical Section system

```
(defun ii-ok-for1 (n i)
  (iff (= (status i n) :critical)
       (and (in-critical n) (= (critical-id n) i))))

(defun ii-ok (n)
  (and (ii-ok-for1 n (a)) (ii-ok-for1 n (b))))

(defthm ok-is-invariant
  (and (ii-ok 0)
       (implies (ii-ok n)
                (and (ok n) (ii-ok (1+ n))))))
```

Figure 3: Inductive Invariant for Critical Section system

```
(ok n)
(equal (status (a) n) ':critical)
(equal (status (b) n) ':critical)
(equal (status (a) n) ':try)
(equal (status (b) n) ':try)
(in-critical n)
(equal (critical-id n) (a))
(equal (critical-id n) (b))
```

Figure 4: Predicates computed for Simple Critical Section system

and `(equal (status (b) n) :critical)` are added to the set of predicates. This process iterates until no more new predicates are added to the set and in this case, when conclude with the set of predicates in Figure 4.

After computing this set of predicates, the invariant prover explores the state graph defined by this predicate set. A node in this state graph is defined by a valuation of a set of boolean variables corresponding to the predicates (one boolean variable for each state predicate). An arc in the state graph is defined by evaluating the next-state transition functions for each boolean variable which are created from a simple replacement of variables for subterms in the computed next-state terms for each predicate. The prover will explore this state graph starting from a suitable initial node in search of a node where the boolean variable corresponding to the target invariant – `(ok n)` in our case – is equal to `nil`. In our simple example, the prover explores the generated graph (with 20 nodes) and does not find a state where the variable for `(ok n)` is `nil` and thus concludes that the original invariant `(ok n)` was true at all times `n`.

It is useful to compare the predicate set which was computed in this case with the subterms in the inductive invariant `ii-ok`. The inductive invariant is a boolean combination of the computed predicates, but the predicate set also includes the predicates `(equal (status (a) n) ':try)` and `(equal (status (b) n) ':try)`. This would suggest that it does not matter what value these predicates might take, and so they could be safely *abstracted* by coercing them from state predicates into input predicates (which are free to take on any value at each step). This abstraction is achieved by the user providing a rewrite rule which tags these terms such that the prover will coerce them to be inputs. In this case, a single rewrite rule will suffice:

```
(defthm coerce-try-status-to-input
  (equal (equal (status p n) ':try)
         (hide (equal (status p n) ':try))))
```

This rewrite rule will fire on both of the predicates we wish to abstract and leave the `hide` operator tagging the terms. The prover will see the `hide` operator in the term and will not add these terms to the set of computed predicates and instead add these terms to the set of input predicates which will be free inputs at each step in the state graph exploration. This removes two state boolean variables and adds two input boolean variables and in our example, reduces the number of nodes in the state graph from 20 nodes to 6 nodes. The 6 nodes in this reduced system are comprised of a node for `(A)` in the critical state, a node for `(B)` in the critical state, a node for some process other than `(A)` or `(B)` in the critical state, and three nodes for the case where no process is in the critical state (these last three nodes are differentiated by whether `critical-id` is equal to `(A)` or `(B)` or neither). The number of input variables increases by 2 from 4 input variables to 6 input variables, but the prover prunes the input valuations to only those valuations which are relevant and thus, the total number of relevant input valuations only increases from 6 to 8.

For the sake of demonstration, let us consider the modification of the `<body>` of the function `in-critical` where the else case of the `if` term is changed from `(= (status (i n) n-) :try)` to `(= (status (i n) n-) :idle)`. When the invariant prover is run on this modified definition with the same target invariant of `(ok n)` (in particular, `(ok 6)`), the prover will report a failure and generate a sequence of states and inputs which illuminate a behavior of the system which leads to an invalidation of `(ok n)`. The reported failure in

```
((:STEP 1
  :STATE (AND (EQUAL (STATUS (A) 0) ':IDLE)
              (EQUAL (STATUS (B) 0) ':IDLE)
              (NOT (EQUAL (CRITICAL-ID 0) (B)))
              (NOT (EQUAL (CRITICAL-ID 0) (A))))
  :INPUT (EQUAL (I 1) (A)))
 (:STEP 2
  :STATE (AND (EQUAL (STATUS (A) 1) ':TRY)
              (EQUAL (STATUS (B) 1) ':IDLE)
              (NOT (EQUAL (CRITICAL-ID 1) (B)))
              (NOT (EQUAL (CRITICAL-ID 1) (A))))
  :INPUT (EQUAL (I 2) (B)))
 (:STEP 3
  :STATE (AND (EQUAL (STATUS (A) 2) ':TRY)
              (EQUAL (STATUS (B) 2) ':TRY)
              (NOT (EQUAL (CRITICAL-ID 2) (B)))
              (NOT (EQUAL (CRITICAL-ID 2) (A))))
  :INPUT (EQUAL (I 3) (CRITICAL-ID 2)))
 (:STEP 4
  :STATE (AND (EQUAL (STATUS (A) 3) ':TRY)
              (EQUAL (STATUS (B) 3) ':TRY)
              (NOT (IN-CRITICAL 3)))
  :INPUT (EQUAL (I 4) (B)))
 (:STEP 5
  :STATE (AND (EQUAL (STATUS (A) 4) ':TRY)
              (EQUAL (STATUS (B) 4) ':CRITICAL)
              (NOT (IN-CRITICAL 4)))
  :INPUT (EQUAL (I 5) (A)))
 (:STEP 6
  :STATE (AND (EQUAL (STATUS (A) 5) ':CRITICAL)
              (EQUAL (STATUS (B) 5) ':CRITICAL))
  :INPUT T)
```

Figure 5: Pruned Failure Reported by Prover for Critical Section Example with fault injected

this case is provided in Figure 5. The failure reported may or may not correspond with a counterexample demonstrating failure of the target invariant. The prover prunes the abstract failure to include only the valuations of relevant and determining state and input predicates at each step in order to facilitate and focus the user's subsequent analysis of the root cause of the failed proof attempt.

3 Invariant Prover Description

We now present an abbreviated description of the invariant prover. The invariant prover takes a target invariant predicate α (which must only reference a single free variable `n`) and either returns `:proven` or `:failure`. If the prover returns `:proven`, then the target invariant α is true for all values of `n`. If the prover returns `:failure`, then the prover was simply unable to prove the target invariant to be true.

As we mentioned before, the prover consists primarily of two phases. The first phase computes a finite set of state predicates (including the target invariant α) which intuitively represents the requisite finite information required at each time `n` to ensure that α is true and will remain true. The second phase of the prover explores the finite state graph defined on the valuations of the predicates computed in the first

```
(defun find-stateps (trm)
  (cond ((atom trm) (throw-error-for-illegal-term))
        ((quotep trm) ())
        ((eq (first trm) 'if)
         (union-trms (find-stateps (second trm))
                     (find-stateps (third trm))
                     (find-stateps (fourth trm))))
        (t (and (state-predp trm) (list trm)))))
```

Figure 6: `find-stateps` function for computing state predicates

phase and reports either success or failure. We will provide an overview of each phase and present several elaborations needed to make the procedure effective and efficient, while providing appropriate feedback to the user.

3.1 Computing State Predicates

The first phase consists of iteratively computing a set Γ of predicates which is initially set to $\Gamma = \{\alpha\}$. A *predicate* is simply an ACL2 term which we interpret in a boolean `iff` context and which references a single free variable `n`. When we refer to terms, we refer to ACL2 terms which are either quoted constants, variable symbols, or the application of a function (either a function symbol or a lambda construct) to a list of argument terms. For a predicate $\gamma \in \Gamma$, the *next-state term* γ' is computed by applying a term rewrite function `rewrt` to the term γ with `n` substituted by `(1+ n)` (i.e. γ' = `(rewrt '((lambda (n) ,`γ`) (1+ n)))`). The function `rewrt` is the term rewriter which we have implemented for the invariant prover and we will provide more details on this term rewriter subsequently. For each *next-state term* γ', we need to compute a set of candidate state predicates which are subterms of γ' and which will be added to the set Γ. The set of candidates is computed by the function `find-stateps` which is defined in Figure 3.1.

The function `state-predp` takes a term and returns true if the term references `n` and has no subterms `(1+ n)` and no calls of the special operator `hide`. Thus, the goal of the first phase is to compute the least set of predicates Γ which includes α and satisfies the closure property that for each $\gamma \in \Gamma$, the set `(find-stateps` γ'`)` is a subset of Γ.

The function `rewrt` takes an input term (along with the current ACL2 `world` object) and performs inside-out ordered conditional rewriting. The rewriter is *inside-out* in that the arguments of a term are rewritten before the term is rewritten (the same as the rewriter in ACL2). The rewriter is *ordered* in that it attempts to apply rewrite rules in a specific order (the same as the rewriter in ACL2). The rewriter is *conditional* in that it will rewrite the hypothesis of a rule before applying the rule itself (the same as the rewriter in ACL2). The `rewrt` function also extends and utilizes a *context* as it rewrites the true and false branches of an `if` term. The `rewrt` function also supports equality reasoning by maintaining a representative structure used for substitutions in each context.

It is reasonable to ask why we chose to write our own term rewriter when ACL2 has its own built-in term `rewrite` function. The simple answer is that the ACL2 rewriter was designed for a different purpose. In the context of the invariant prover, it is important that the user has precise control over the rewriter and that the behavior of the rewriter is predictable. Further, we wanted the invariant prover to be "self-contained" with the defining functions as simple as possible in the hope that at some point in the future we will complete a formal soundness proof of the invariant prover. In order to ensure these properties now and moving forward, we decided it was appropriate to write our own term rewriter.

The term rewriter does interface with ACL2 by querying the current ACL2 world for determining candidate rewrite rules to apply (and the order in which to apply them). Thus, the user can prove theorems in ACL2 and have the corresponding rewrite rule applied in the invariant prover. Currently, only rules of class `:rewrite` are used by the invariant prover, but the `rewrt` function does include support for several "meta"

features of term rewriting in ACL2. In particular, the `rewrt` function has support for `syntaxp`, `bind-free`, `hide`, and `force`. It also supports the use of `executable-counterparts` for efficient reduction of ground terms. The function `rewrt` also has minimal support for congruence rewriting for `iff` and `equal` contexts. The rewriter is designed to be simple, predictable, and sufficient for the purpose of computing the set of predicates Γ.

Limited assume-guarantee reasoning. As we mentioned before, the `rewrt` function supports the use of `forced` hypothesis. In the invariant prover, when a rule is applied with a forced hypothesis, the forced hypothesis is assumed to be true and added to a list of *forced assumptions*. For each assumption which is a predicate, we later call the invariant prover again in order to prove that the assumption is true for all values of `n`. During these subsequent calls of the invariant prover, we will assume any previously proven predicates to simply be true. The apparent circularity in these crossing assumptions is resolved by an induction over time (the predicates are assumed true only at previous values of `n`). This limited assume-guarantee reasoning allows the division of invariant proofs of concurrent systems involving multiple components into proofs of invariants separately for each of the components.

3.2 Exploring the Predicate State Graph

The first phase of the prover computes a finite set of predicates Γ. The second phase of the prover consists of exploring the finite-state graph defined on the valuations of the predicates in Γ. The built-in "model checker" we have implemented is an efficient explicit-state breadth-first search for invariant failure. We are currently working to provide interfaces to other external model checkers. The finite-state exploration consists of the following steps: (a) compute the next-state function and constraint function for the state graph (which are defined in a manner to afford the exploration of only those arcs which are possible or relevant), (b) compile the next-state and constraint functions, (c) perform a breadth-first search through the state graph, and (d) if a failure is detected, prune the failure witness.

The next-state function of the state graph is computed by first computing the set of *input predicates* Φ as the union of the term sets `(find-inputps` γ'`)` for each $\gamma \in \Gamma$. The function `find-inputps` is the same as `find-stateps` except that it replaces `(state-predp trm)` with `(not (state-predp trm))`. After computing Φ, we assign a boolean input variable $iv(\phi)$ for each $\phi \in \Phi$ and a boolean state variable $sv(\gamma)$ for each $\gamma \in \Gamma$. We then compute a next-state function for each boolean state variable $sv(\gamma)$ by taking the term γ' and replacing every subterm in Γ with the corresponding boolean state variable and every subterm in Φ with the corresponding boolean input variable. We also generate a constraint function which determines whether an input valuation is not possible in a given state or if a state valuation is not possible. The next-step and constraint functions are defined in a manner which allows the state exploration to only explore input valuations which are relevant at a given state and exclude any input or state valuations which can be proven to be impossible. An unfortunate side effect of our methodology is the creation of a large number of input predicates which were abstracted by the user. Attempting every boolean combination of free inputs would be expensive and thus, we prune the set of input valuations to only include the set of values which could have any unique effect at a given state on the behavior of the system.

The next step of the second phase involves installing the generated functions for the next-state function of the graph and the parent functions defining the breadth-first search through the state graph. These functions are installed using the ACL2 `ld` command and then compiled using the ACL2 `comp` command (which calls the Common Lisp compiler). We pack the boolean variables for the states and inputs into bitvectors stored in Common Lisp natural numbers and we use hash-tables implemented with stobjs to provide fast lookup for previously reached states. The resulting compiled breadth-first search top-level loop function is then called directly with any results passed back through a top-level single-threaded object or stobj (which we freely admit is a hack). We chose to perform a breadth-first search because we want to have minimal-length failure witnesses. Further, when the path to a failing state is computed, we prune the corresponding predicate valuations which are reported to the user to only include the predicate valuations which were necessary for the failure to occur. This pruning is important to assist in focusing the attention of the user to the key points where the exploration of the generated abstract graph was ineffective.

4 ESI Cache Coherence Protocol Example System

As a further example usage of the invariant prover, we consider a simplified version of the protocol or algorithm-level definition of a memory system with caches utilizing the ESI cache coherence protocol. In this protocol, each processor will access a central memory divided into so-called *cache lines* or simply *lines* which are further divided into atomic addressable data. When a processor needs to read from an address, the processor first loads the cache-line for the address from central memory into its local cache (if it has not done so already) by performing a `:fill` operation. If the processor wants to store a value to an address in the cache-line, it will need to get exclusive access to the cache-line and in our version of ESI, this will correspond to performing a `:fille` operation. At any time, for a variety of reasons, a cache-line from a local cache may be evicted from the local cache via a `:flush` operation. If the cache-line in the local cache was modified, then it would need to be copied back to main memory.

The definition of our example ESI system is provided in Figure 7. The constrained functions `(proc n)`, `(addr n)`, `(data n)`, and `(op n)` define a sequence of input requests from the "environment". The function `(op n)` defines the operation to be performed at time `n`, where the relevant operations are `:load`, `:store`, `:fill`, `:fille`, and `:flush`. The `(proc n)`, `(addr n)`, and `(data n)` functions define the processor identifier, address, and data value at each time `n`. The constrained function `c-l` maps atomic data addresses to cache-line addresses which models the relationship between addresses and cache-lines in a memory system. The function `(mem c n)` defines the value of a cache-line in main shared memory at cache-line address `c` at time `n`. The function `(cache p c n)` takes a processor identifier `p`, a cache-line address `c`, and a time value `n` and returns the value of the cache-line at address `c` in the local cache of `p` at time `n`. The functions `(excl c n)` and `(valid c n)` define sets of processor identifiers for a cache-line address `c` at time `n`. The set `(valid c n)` is the set of processors which have the cache-line `c` in their local cache. The set `(excl c n)` is the set of processors which have the cache-line in an exclusive state. We do not model the distinction between the modified and exclusive cache states and simply assume for simplicity that a cache-line in an exclusive state has modified data, but this would be a fairly straightforward extension of our example.

The definitions of the ESI cache system functions in Figure 7 use operations on sets and records derived from the books presented in [8]. The operator `(g a r)` returns the data value associated with address `a` in record `r`. The operator `(s a v r)` returns the record `r` changed to associate address `a` with value `v`. The value `nil` is the unique value representing the empty set and the empty record (and `(equal (g a nil) nil)` is a theorem). The operator `(in e s)` tests whether the element `e` is in the set `s`. The operators `(sadd e s)` and `(sdrop e s)` return the set `s` where, respectively, the element `e` has been added and removed. The operator `(c1 s)` returns true iff the set `s` is a singleton. The operator `(scar s)` returns the least element (in the ACL2 total order `lexorder`) from the set `s` – `scar` stands for "set `car`". The operators `c1` and `scar` are not used directly to define the ESI cache system functions, but instead arise from the application of rewrite rules during the first phase of the invariant prover.

In Figure 4 we provide the target invariant `(ok n)` we wish to prove. The constrained functions `(p)` and `(a)` define an arbitrary process identifier and address respectively. The function `(a-dat n)` is an auxiliary variable which records the last data value which was stored to address `(a)`. The value of `(a-dat n)` is `nil` if no store to `(a)` has been performed. The function `(ok n)` remains true as long as the value which is returned on a successful `:load` operation is equal to `(a-dat n)`. Effectively, the proof of `(ok n)` as invariant ensures that the value returned by a load of address `(a)` is the value most recently stored to address `(a)` and this is the basic property of coherence which any shared memory system should satisfy.[1]

We first note that the rewrite rules which were already proven about the sets and records operators were immediately and directly useful in the predicate computation phase of the invariant prover. Even though we can leverage these existing books of rewrite rules for operators on records and sets, it is still the case that running the invariant prover on this ESI cache example is a little more subtle than the earlier critical section example. In particular, blind application of the invariant prover will either produce a model which

[1] Note that there are many other properties which a shared memory system should satisfy such as ordering, consistency, liveness, and fairness. We only consider coherence here as an example, but we can imagine the possibility to reduce other desired properties of a memory system to the proof of suitable invariants using the invariant prover.

```
(encapsulate (((proc *) => *) ((op *) => *)
              ((addr *) => *) ((data *) => *))
 (local (defun proc (n) n)) (local (defun op (n) n))
 (local (defun addr (n) n)) (local (defun data (n) n)))

(encapsulate (((c-l *) => *)) (local (defun c-l (a) a)))

(defun in1 (e s) (in e s))

(define-system mesi-cache
 (mem (c n) nil
   (cond ((/= (c-l (addr n)) c) (mem c n-))
         ((and (= (op n) :flush)
               (in1 (proc n) (excl c n-)))
          (cache (proc n) c n-))
         (t (mem c n-))))

 (cache (p c n) nil
   (cond ((/= (c-l (addr n)) c) (cache p c n-))
         ((/= (proc n) p) (cache p c n-))
         ((or (and (= (op n) :fill) (not (excl c n-)))
              (and (= (op n) :fille) (not (valid c n-))))
          (mem c n-))
         ((and (= (op n) :store) (in1 p (excl c n-)))
          (s (addr n) (data n) (cache p c n-)))
         (t (cache p c n-))))

 (excl (c n) nil
   (cond ((/= (c-l (addr n)) c) (excl c n-))
         ((and (= (op n) :flush)
               (implies (excl c n-)
                        (in1 (proc n) (excl c n-))))
          (sdrop (proc n) (excl c n-)))
         ((and (= (op n) :fille) (not (valid c n-)))
          (sadd (proc n) (excl c n-)))
         (t (excl c n-))))

 (valid (c n) nil
   (cond ((/= (c-l (addr n)) c) (valid c n-))
         ((and (= (op n) :flush)
               (implies (excl c n-)
                        (in1 (proc n) (excl c n-))))
          (sdrop (proc n) (valid c n-)))
         ((or (and (= (op n) :fill) (not (excl c n-)))
              (and (= (op n) :fille) (not (valid c n-))))
          (sadd (proc n) (valid c n-)))
         (t (valid c n-)))))
```

Figure 7: Definition of ESI Cache System example

```
(encapsulate (((p) => *) ((a) => *))
  (local (defun p () t)) (local (defun a () t)))

(define-system mesi-specification
 (a-dat (n) nil
   (if (and (= (addr n) (a))
            (= (op n) :store)
            (in1 (proc n) (excl (c-l (addr n)) n-)))
       (data n)
     (a-dat n-)))

 (ok (n) t
   (if (and (= (proc n) (p))
            (= (addr n) (a))
            (= (op n) :load)
            (in (p) (valid (c-l (addr n)) n-)))
       (= (g (a) (cache (p) (c-l (a)) n-)) (a-dat n))
     (ok n-))))
```

Figure 8: Target Invariant for ESI Cache System example

```
(defthm in1-case-split
  (equal (in1 e s)
         (cond ((not s) nil)
               ((c1 s)  (equal e (scar s)))
               (t       (hide (in e s))))))
```

Figure 9: Main Rewrite Rule needed in ESI Cache System verification

is too abstract, or does not converge in the computation of Γ. Thus, the user will need to introduce a few rewrite rules to introduce certain "case splits" and abstract certain state predicates.

The main rewrite rule which we needed to add centers from a key concept in the validity of the invariant: the exclusive set `(excl c n)` at any point in time is either empty or only has a single element. In order to capture this intuition, we first "tag" all membership tests of the `excl` set with the `in1` operator. The `in1` function is defined to be `in`, but it's use denotes that the second parameter to `in1` is either the empty set or a singleton set. We make use of this meta-information about `in1` applications by introducing the rewrite rule in Figure 9. The rule `in1-case-split` will force any application of `in1` to be rewritten to introduce a case-split for the cases where the set is empty, the set is a singleton set, or otherwise. In the case of an empty set `s`, `(in1 e s)` simply reduces to `nil`. In the case of a singleton set, we choose to rewrite `(in1 e s)` into `(equal e (scar s))` in order to cause the term bound to `e` to be replaced with the term `(scar s)`. Since we expect `s` in an `(in1 e s)` application to either be empty or a singleton, in the final case, we will rewrite `(in1 e s)` to be `(hide (in e s))` in order to coerce it to be a free input.

After introducing the rewrite rule `in1-case-split` and adding a few additional rules to abstract state predicates which are introduced during the first phase of the invariant prover, we end up with the following set of computed predicates provided in Figure 10. The model built on these predicates is checked (with only 11 states) and verifies that `(ok n)` is indeed an invariant of the system – the invariant prover completes the entire process in a couple of seconds. Note the importance of the `in1-case-split` rule since it introduces `(scar (excl (c-l (a)) n))` as a relevant process identifier value to select as a replacement for `(proc (1+ n))`. This eventually causes the last predicate in Figure 10 to be generated and added to the set. This predicate is important since it tracks the fact that the value stored for address `(a)` at an arbitrary

```
(ok n)
(valid (c-l (a)) n)
(in (p) (valid (c-l (a)) n))
(excl (c-l (a)) n)
(c1 (excl (c-l (a)) n))
(equal (scar (excl (c-l (a)) n)) (p))
(equal (a-dat n) (g (a) (mem (c-l (a)) n)))
(equal (a-dat n) (g (a) (cache (p) (c-l (a)) n)))
(equal (a-dat n) (g (a) (cache (scar (excl (c-l (a)) n)) (c-l (a)) n)))
```

Figure 10: Predicate Set in ESI Cache System example

processor's (not necessarily `(p)`) local cache is equal to `(car (a-dat n))`. Factors like this have made it difficult to effectively abstract the "processor index" dimension in model checking real-world concurrent systems in previous research efforts. This places greater onus on having an expressive language for defining the necessary operators to effectively represent the key components of the state of a system.

5 Conclusions and Future Work

In this paper we have presented a tool-supported methodology for proving invariants about reactive systems defined in ACL2. We wish to stress that even though the methodology may appear at one level to be automatic or push-button, it is not. The user will (at the least) be required to carefully analyze the predicates generated during the first phase of the prover to consider the effectiveness of the current rewrite rules and determine which predicates can be safely abstracted. This analysis requires an understanding or intuition about which facets of the state of the system affect the continued validity of the invariant. If the user abstracts too many details, then the user will end up debugging false failures in the resulting state graph explorations. If the user does not abstract enough details, then the predicate computation or subsequent model check may never successfully complete. As it stands now, the user must understand which predicates are generated and which of these generated predicates are relevant. This is also why we maintain the requirement that the first phase of the prover must reach a fixpoint in the computed predicate set – this forces the user to examine the predicates to determine which predicates to include, exclude, or avoid.

Independent of the need to examine and control the computed predicate set, the user may also be required to introduce non-obvious "case splits" and rewrites. This was required in the ESI cache system where the key rewrite rule `in1-case-split` required some user insight into the nature of the validity of the invariant. While some of these insights may transfer from one system definition to another, with each new system, some insight will be required to properly reduce the state information to a manageable finite set of predicates. In the worst case, the user may be required to restructure the definition of the system or introduce entirely new operators in order for the rewriting during the predicate computation phase to be effective. For example, if one alternatively defined the ESI cache system where the cache state information stored in the `(excl c n)` and `(valid c n)` sets were stored as an extra state component within the local cache for each processor (e.g. turn each set of process identifiers into a record associating process id.s with boolean flags), then the existing set operators `scar` and `c1` would not be useful and new operators which iterate through records would need to be defined along with the proofs of the theorems providing the necessary rewrite rules.

Given these limitations of the presented methodology, it is fair to consider the presented methodology as a variant of theorem proving as opposed to an extension of model checking. Further, the reader may be inclined to believe that with the example systems we have presented, that the prover is only effective in cases where it is "easy" to define an inductive invariant – certainly a sufficient inductive invariant for each of the systems presented in this paper would be easy to define. But this is also an inappropriate conclusion because the example systems we presented were chosen to convey the ability of the methodology to "automatically" abstract large state components (like processor state and caches) which tend to cause

difficulty for model checking approaches. As these example systems are further elaborated (more protocol states, operation pipelines, etc.) the difficulty and complexity in defining and proving an inductive invariant can quickly become overwhelming. At the same time, the key rewrite rules needed to reduce components of the state such as larger caches may apply directly to the elaborated systems. Further, the elaborations (e.g. more intermediate protocol states) may immediately transfer to the state predicates defining the abstract state graph, and thus the requisite case analysis can be performed implicitly in the efficient exploration of the abstract state graph rather than being codified directly in the definition of a complex inductive invariant.

In future work, we plan to apply the invariant prover to several new example systems which further identify limitations in the approach. We eventually hope to provide an interface for operating the invariant prover on top of an effective library of rewrite rules encoding common RTL-level operators found in digital hardware description. We are currently working on providing backend interfaces to traditional model checkers and bounded model checkers (for finding failing executions). We also are exploring improvements to the rewriting phase of the tool to provide more early reduction of the generated terms to avoid certain predicates and further reduce the next-state functions in the abstract models. We are also looking into improving certain heuristics for introducing case splits automatically.

References

[1] Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In Dwyer, M.B., ed.: 8th International SPIN Workshop on Model Checking of Software. Volume 2057 of LNCS., Springer-Verlag (2001) 103–122

[2] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counter-example Guided Abstraction Refinement. In Emerson, E.A., Sistla, A.P., eds.: 12th International Conference on Computer-aided Verification. Volume 1855 of LNCS. (2000) 154–169

[3] Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In Grumberg, O., ed.: Computer-Aided Verification (CAV). Volume 1254 of LNCS. (1997) 72–83

[4] Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Approximation or Analysis of Fixpoints. In: Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1997) 238–252

[5] Emerson, E., Sistla A.: Symmetry and Model Checking. In C. Courcoubetis, ed.: 5th International Conference on Computer-aided Verification. Volume 697 of LNCS. (1993)

[6] Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)

[7] Kaufmann, M., Manolios, P., Moore, J.S., eds.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)

[8] Kaufmann, M., Sumners, R.: Efficient Rewriting of Data Structures in ACL2. In: 3rd International Workshop on ACL2 Theorem Prover and Its Applications. (2002)

[9] Manolios, P., Namjoshi, K., Sumners R.: Linking Model-checking and Theorem-proving with Well-founded Bisimulations. In: N. Halbwachs and D. Peled, eds: 11th International Conference on Computer-aided Verification. Volume 1633 of LNCS. (1999)

[10] Sumners, R.: An Incremental Stuttering Refinement Proof of a Concurrent Program in ACL2. In Kaufmann, M., Moore, J.S., eds.: Second International Workshop on ACL2 Theorem Prover and Its Applications, Austin, TX (2000)