Proving Invariants via Rewriting and Abstraction [*]

Rob Sumners
Advanced Micro Devices, Inc.
Austin, TX 78741
robert.sumners@amd.com

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
sandip@cs.utexas.edu

July 2005

Abstract

We present a deductive method for proving invariants of reactive systems. Our approach uses term rewriting to reduce invariant proofs to reachability analysis on a finite graph. This substantially automates invariant proofs by obviating the need to define inductive invariants while still benefitting from the expressiveness of deductive methods. We implement a procedure supporting this approach which interfaces with the ACL2 theorem prover. The interface affords sound extension of our procedure with rewrite rules based on proven theorems. We demonstrate the method in the verification of cache coherence protocols.

1 Motivation

The goal of *invariant proving* is to show that a certain target property of a reactive system is an invariant. Invariant proving is a key problem in formal verification. Verification of safety properties can be reduced to the proof of an invariant. Even in the proofs of liveness properties, one typically needs to establish some auxiliary invariance condition.

Invariant proving is difficult for both model checking and theorem proving. The model checking approach involves a (symbolic or explicit) search to check if all the reachable system states satisfy the target property. If the number of states is tractable, the process is automatic usually with the additional benefit of counterexample generation when the verification fails. However, the method is limited in practice by state explosion. The theorem proving approach involves strengthening the target property to an *inductive invariant*. This approach is generally insensitive to state explosion but in practice can require significant user interaction for defining the inductive invariant. In addition, inductive invariants are brittle and often require extensive modification to match design changes. However, theorem provers support expressive logics which allow users to succinctly define systems, properties, and any additional functions and lemmas that enable efficient proofs.

We present a method to bridge the automation gap between theorem proving and model checking, while still preserving the expressiveness of theorem proving. We use *term rewriting* to reduce an invariant proof to the reachability analysis of a finite graph; the graph is a *predicate abstraction* [1] of the system. Rewriting is guided by *rewrite rules* that relate the different functions used to model the system. The rules are selected from theorems proven by a theorem prover.

Our approach transfers the user responsibility from defining inductive invariants to designing rules that manipulate functions used in system definitions. How does this reduce manual effort? Proving rewrite rules *does* require human interaction. However, while inductive invariants are defined by a user for a specific

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

system, rewrite rules are proven facts about the functions used to model the system and can be used in other systems using the same functions. It is customary for the users of a theorem prover to design rules that simplify terms which arise during proofs [2, 3]. We found that most of the rules necessary for our work are generic, and already available as proven theorems in a deductive setting. Note that since the logic of a theorem prover is undecidable, *any* collection of rules is incomplete and it may be necessary for the user to extend or refine the existing rules and definitions. However, feedback from our procedure assists in the development of these extensions and refinements which, in our experience, can be reused in the verification of similar systems.

Our procedure is interfaced with the ACL2 theorem prover [4]. ACL2 has been used to verify several commercial systems [5, 6], and we make use of rewrite rules that have been proven in these efforts. However, ACL2 is not critical to our method; it is used primarily as a mechanized logic with which we are familiar. We believe that our procedure can be easily ported to other theorem provers.

The remainder of the paper is organized as follows. In Section 2, we describe the ACL2 logic and illustrate our method with a simple example. We present our procedure in Section 3. In Section 4, we demonstrate the method in proving invariants of cache coherence protocols. In Section 5, we discuss related work. We conclude in Section 6. An implementation of the procedure, and the proofs described in this paper are available from the web page of the second author [7].

2 Background and Overview

In this section, we review the ACL2 logic, and illustrate our method to prove invariants of reactive systems modeled in ACL2. This paper is not about ACL2; our overview only provides a formal context for our work. Readers interested in ACL2 are referred to [4] for a comprehensive description.

2.1 The ACL2 Logic

ACL2 is a first-order logic of recursive functions with a syntax similar to Lisp. A *term* is a variable, a constant, or the application of an n-ary function `f` to n terms, written `(f` t_1 t_2 `...` t_n`)`. The set of *constants* is open but includes integers, strings, and symbols `T` and `NIL` denoting boolean true and false. Formulas in the logic are represented by terms. For example, the term (with variables x, y, z)

```
(implies (< x y) (< (+ x z) (+ y z)))
```

represents a formula about arithmetic. The syntax is quantifier-free, and variables in formulas are implicitly universally quantified. The term above can be read as follows in the logic: "For all x, y, z, if x is less than y, then $x + z$ is less than $y + z$."

ACL2 axiomatizes a subset of Common Lisp. An axiom relating functions `car` and `cons` is: `(equal (car (cons x y)) x)`. Theorems can be proven about axiomatized functions. The inference rules are propositional calculus with equality and instantiation, and well-founded induction up to ϵ_0. For example, instantiation of the above axiom yields the theorem: `(equal (car (cons 2 y)) 2)`.

We make special use of the ternary function `if`, which is axiomatized to be "if-then-else": `(if x y z)` is equal to `z` if `x` is equal to `NIL`, otherwise `y`. Since terms containing `if` are extensively used in ACL2 (and Lisp), there are constructs to structure such terms. For example, we use `(cond (a b) (c d) ... (x y))` to stand for `(if a b (if c d ... (if x y NIL) ... ))`. Boolean operations `and`, `or`, `implies`, etc. are axiomatized using `if`:

```
(equal (and x y)     (if x y NIL))
(equal (or x y)      (if x x y))
(equal (implies x y) (if x (if y T NIL) T))
(equal (iff x y)     (and (implies x y) (implies y x)))
```

In this paper, we use standard mathematical notations to represent certain functions. We use infix operators "=" for `equal`, "∧" for `and`, "∨" for `or`, "⇒" for `implies`, and "⇔" for `iff`. Instead of writing `(implies` γ `(equal` α β`))`, we will write $\gamma \Rightarrow (\alpha = \beta)$. We also write `(and` x_1 x_2 `...` x_n`)` to mean `(and` x_1 `(and` x_2 `... ))` and similarly for `(or` x_1 x_2 `...` x_n`)`. If S is a set $\{e_0, e_1, \ldots, e_n\}$ and M maps S to terms, then we write $\bigvee_{e\in S} M(e)$ and $\bigwedge_{e\in S} M(e)$ to denote (or $M(e_1)$ $M(e_2)$ ... $M(e_n)$) and (and $M(e_1)$ $M(e_2)$ ... $M(e_n)$) respectively.

ACL2 provides facilities to consistently introduce new axioms. New total functions can be *defined* (or axiomatized), like the function `factorial` below

```
(factorial n) = (if (+ve n) (* n (factorial (- n 1))) 1)
```

where `(+ve n)` returns `T` if `n` is a positive natural number, else `NIL`. The logic also supports mutually recursive function definitions. ACL2 further allows the introduction of a function which is only known to satisfy some specified axioms. We can introduce a function `E` that only satisfies the axiom: `(+ve (E n)) = T`. Such axioms are called *constraints*, and `E` is then called a *constrained function*. A theorem about a constrained function f is provable for *any* function $\hat{f}$ satisfying the constraints. A constrained function with no constraint is termed *generic*. If ϕ is a theorem and $\hat{\phi}$ is obtained from ϕ by replacing occurrences of a generic function g with *any* function $\hat{g}$ of the same arity, then $\hat{\phi}$ is a theorem.

2.2 System Models and Invariants

Reactive systems consist of several *components* that perform on-going, non-terminating computations while interacting with an external environment. The "state" of the system at any time is given by the value of each component. For example, consider a trivial system with two components `C0` and `C1`. `C0` and `C1` initially have the value `1`. At each instant, they interact with an environment `E` and execute as follows.

- If `E` is `NIL` then `C0` gets the previous value of `C1`; otherwise `C0` is unchanged.
- If `E` is `NIL` then `C1` is assigned to the value `42`; otherwise `C1` is unchanged.

Such systems can be modeled [5] by specifying, for each component `C`, a function `(C n)` that returns the value of `C` at time `n`. The external stimuli are modeled by generic functions of `n`. We formalize *time* by two functions, a 0-ary function `t0` for "initial time", and a unary function `t+` for "next time". The value of a component at time `(t+ n)` can depend on other components at time `n` and the external stimuli at time `(t+ n)`. Equations 1-4 below define the system above, and can be specified using mutually recursive function definitions.[1] Here `E` is a generic unary function where `(E n)` is the value supplied by the environment at time `n`.

```
1. (C0 (t0))    = 1
2. (C1 (t0))    = 1
3. (C0 (t+ n)) = (if (E (t+ n)) (C0 n) (C1 n))
4. (C1 (t+ n)) = (if (E (t+ n)) (C0 n) 42)
```

We call a term Φ a *temporal term* if it has a single variable `n` representing time. A temporal term Φ is an *invariant* if it does not evaluate to `NIL` for any `n` (i.e. $\Phi \Leftrightarrow$ `T` is a theorem). The goal of *invariant proving* is to show that a temporal term Φ is an invariant. For the system above, an invariant is $\Phi_0 \triangleq$ `(+ve (C0 n))`.

A deductive method for invariant proving is to define an inductive invariant. A unary function `inv` is an *inductive invariant* strengthening Φ if **I1-I3** are theorems:

I1: `(inv (t0))` $\Leftrightarrow$ `T`

I2: `(inv n)` $\Rightarrow \Phi$

I3: `(inv n)` $\Rightarrow$ `(inv (t+ n))`

If *some* function `inv` is an inductive invariant strengthening Φ, then the invariance of Φ follows by induction on time `n`. For the example above, `(inv n) = (and (+ve (C0 n)) (+ve (C1 n)))` is an inductive invariant strengthening Φ_0.

[1]We actually need two other unary functions, namely `t-` for "previous time" and `tzp` to check if the "current time" is `t0`. We omit discussion of these functions for brevity.

2.3 Overview of Our Approach

Consider proving for the system above that Φ_0 is an invariant. Instead of manually defining an inductive invariant, our approach "discovers" the relevant terms by rewriting. The term $\mathtt{T}_0$ below is the result of rewriting the term Φ'_0 (which is the term Φ_0 with `n` replaced by `(t+ n)`) using equation 3 along with the following equation 5: `(+ve (if x y z))` = `(if x (+ve y) (+ve z))`.

$\mathtt{T}_0 \triangleq$ `(if (E (t+ n)) (+ve (C0 n)) (+ve (C1 n)))`

We treat $\mathtt{T}_0$ as a boolean combination of `(E (t+ n))`, `(+ve (C0 n))`, and `(+ve (C1 n))`, and classify `(+ve (C1 n))` as a new temporal term Φ_1. Using equations 4 and 5, and the computed fact `(+ve 42) = T`, we similarly rewrite Φ'_1 (that is, Φ_1 with `n` replaced by `(t+ n)`) to

$\mathtt{T}_1 \triangleq$ `(if (E (t+ n)) (+ve (C0 n)) T)`

$\mathtt{T}_0$ and $\mathtt{T}_1$ specify how Φ_0 and Φ_1 are "updated" at time `n`. We make this explicit by constructing the following mapping N from variables to terms:

$N(v_0) \triangleq$ `(if` e v_0 v_1`)`, $N(v_1) \triangleq$ `(if` e v_0 `T)`

N is obtained by replacing terms Φ_0, Φ_1, and `(E (t+ n))` in $\mathtt{T}_0$ and $\mathtt{T}_1$ with v_0, v_1, and e respectively. Informally, variables in the domain of N (namely, v_0 and v_1) "track" the temporal terms of interest (namely, Φ_0 and Φ_1), while other variables (namely, e) represent terms that are abstracted (namely, `(E (t+ n))`).

N specifies a directed graph G as follows. The nodes are mappings from $\{v_0, v_1\}$ to the set $\{\mathtt{T}, \mathtt{NIL}\}$. The mapping $Z \triangleq [v_0 \mapsto \mathtt{T}, v_1 \mapsto \mathtt{T}]$, corresponding to values of Φ_0 and Φ_1 at time `(t0)`, is the *initial node*. For nodes $p \triangleq [v_0 \mapsto x, v_1 \mapsto y]$ and $p' \triangleq [v_0 \mapsto x', v_1 \mapsto y']$, there is an edge from p to p' if for some $e_b \in \{\mathtt{T}, \mathtt{NIL}\}$, $x' =$ `(if` e_b x y`)` and $y' =$ `(if` e_b x `T)`.

We then prove that Φ_0 is an invariant by checking that v_0 is mapped to `T` in each node p reachable from node Z. Notice that G is a predicate abstraction of this example system.

3 Procedure

We introduce some notations before describing our procedure. We use $[a \mapsto \alpha, b \mapsto \beta]$, where a and b are distinct, to denote a finite mapping η with domain $\{a, b\}$ and range $\{\alpha, \beta\}$ so that $\eta(a) = \alpha$ and $\eta(b) = \beta$. We use $dom(\eta)$ to denote the domain of η. Given mappings η_1 and η_2 on disjoint domains, $\eta_1 \cup \eta_2$ denotes their union: if $\eta_1 \triangleq [a \mapsto \alpha, b \mapsto \beta]$ and $\eta_2 \triangleq [c \mapsto \gamma]$, then $\eta_1 \cup \eta_2 \triangleq [a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma]$.

Let $\nu(\tau)$ be the set of variables in term τ. If $\nu(\tau)$ is empty, τ is called a *ground term*. For term τ and mapping σ from variables to terms, τ/σ is the term obtained by replacing every variable $v \in dom(\sigma)$ in τ with $\sigma(v)$. Term τ is called a *boolean term* if it is (i) a variable, or (ii) one of `T` or `NIL`, or (iii) a term `(if` α β γ`)` where α, β, and γ are boolean terms. For a given set V of variables, $B(V)$ is the set of mappings from V to $\{\mathtt{T}, \mathtt{NIL}\}$.

Given a temporal term Φ_0, our procedure first returns three mappings Γ, N, and Z from variables to terms such that the following hold:

C1: $dom(N) \subseteq dom(\Gamma)$; $Z \in B(dom(N))$.

C2: $\Gamma(v_0) = \Phi_0$ for some $v_0 \in dom(N)$.

C3: For each $v \in dom(N)$, $N(v)$ is a boolean term.

C4: For each $v \in dom(N)$, $\nu(N(v)) \subseteq dom(\Gamma)$.

C5: For each $v \in dom(N)$, $\Gamma(v)/[\mathtt{n} \mapsto \mathtt{(t+\ n)}] \Leftrightarrow N(v)/\Gamma$ is a theorem.

C6: For each $v \in dom(N)$, $\Gamma(v)/[\mathtt{n} \mapsto \mathtt{(t0)}] \Leftrightarrow Z(v)$ is a theorem.

In our example, $\Gamma \triangleq [v_0 \mapsto$ `(+ve (C0 n))`$, v_1 \mapsto$ `(+ve (C1 n))`$, e \mapsto$ `(E (t+ n))`$]$. We then construct a directed *abstraction graph* G as follows:

G1: The set of nodes in G is the set $B(dom(N))$.

G2: The mapping Z is the *initial node*.

G3: Let $V \triangleq dom(\Gamma) \backslash dom(N)$. There is an edge from node p to node q if there exists $i \in B(V)$ such that for all $v \in dom(N)$, $N(v)/[p \cup i] \Leftrightarrow q(v)$ is a theorem.

C3 and **C4** imply that $N(v)/[p \cup i]$ is a ground boolean term; thus the edge relation from **G3** can be determined by evaluation. We now show how the invariance of Φ_0 reduces to reachability in G and how we compute Γ, N, and Z.

For node p of G, we define the *minterm* of p, denoted by $M(p)$, as:

$$M(p) \triangleq \bigwedge_{v \in dom(N)} (\Gamma(v) \Leftrightarrow p(v))$$

Let $nbrs(p)$ denote the set of all nodes q such that there is an edge from p to q. From **G3** and **C5** the following can be shown to be a theorem.

$$M(p) \Rightarrow (\bigvee_{q \in nbrs(p)} M(q))/[\mathtt{n} \mapsto \mathtt{(t+\ n)}] \qquad \textbf{(1)}$$

Let $R(p)$ be the set of all nodes reachable from p. Since for any node $q \in R(p)$, $R(q) \subseteq R(p)$, it follows from **(1)** that the following is a theorem.

$$(\bigvee_{q \in R(p)} M(q)) \Rightarrow (\bigvee_{q \in R(p)} M(q))/[\mathtt{n} \mapsto \mathtt{(t+\ n)}] \qquad \textbf{(2)}$$

Claim 1 below is well-known, and follows from the definition of inductive invariants using **(2)** and **C6**, and lets us conclude the invariance of Φ_0 by checking if $p(v_0) = \mathtt{T}$ for each $p \in R(Z)$.

Claim 1 *If for every node $p \in R(Z)$, $p(v_0) = \mathtt{T}$ then* `(inv n)` $= (\bigvee_{q \in R(Z)} M(q))$ *is an inductive invariant strengthening Φ_0.*

To compute Γ, N, and Z which satisfy **C1-C6**, we will define procedures *rewrt* and *chop* with the following properties. Given term τ, *rewrt*(τ) returns term τ^* such that $\tau \Leftrightarrow \tau^*$ is a theorem. Given term τ and mapping η from $\nu(\tau)$ to terms, *chop*(τ, η) returns a pair $\langle \eta', \rho \rangle$ where ρ is a boolean term and η' is a mapping from variables to terms such that the following properties hold: (a) ρ/η' is syntactically equal to τ, and (b) For $v \in dom(\eta)$, $\eta'(v) = \eta(v)$. We then compute Γ and N as follows.

```
Initially $\Gamma := [v_0 \mapsto \Phi_0]$; $N := []$;
while $\exists v \in dom(\Gamma) \backslash dom(N)$ such that $(statep(\Gamma(v)) = \mathtt{T})$
    let $v \in dom(\Gamma) \backslash dom(N)$ such that $(statep(\Gamma(v)) = \mathtt{T})$
        $\langle \Gamma, \rho \rangle$  :=  $chop(rewrt(\Gamma(v)/[\mathtt{n} \mapsto \mathtt{(t+\ n)}]), \Gamma)$
        $N$       :=  $N \cup [v \mapsto \rho]$
end while
Return $\langle \Gamma, N \rangle$
```

Recall that the value of a component at time `(t+ n)` depends on the components at time `n` and the external stimuli at time `(t+ n)`. Hence if `(t+ n)` occurs as a subterm of a term τ, then τ involves an external stimulus. Call a temporal term τ a *state predicate* if it does not contain `(t+ n)` or the application of the special function `hide` in any subterm; otherwise we call τ an *input predicate*. We use `hide` for user-guided abstractions, and we will discuss it in Section 3.1. Procedure *statep*(τ) above returns `T` if τ is a state predicate, otherwise it returns `NIL`. We compute Z as follows: for a ground term τ, let $val(\tau) \in \{\mathtt{T}, \mathtt{NIL}\}$ where $\tau \Leftrightarrow val(\tau)$ is a theorem, then $Z(v) \triangleq val(\Gamma(v)/[n \mapsto \mathtt{(t0)}])$ for each $v \in dom(N)$.

It remains for us to describe *rewrt* and *chop*. Procedure *rewrt* is a *term rewriter*. It transforms a term τ into another term τ^* using the system definitions and theorems as follows. A definition or theorem of the form $\gamma \Rightarrow (\alpha = \beta)$ where α, β, and γ are terms, is treated as a *rewrite rule*.[2] The rule is *applicable* to term σ if there is a mapping b from variables to terms such that $(\gamma/b \Leftrightarrow \mathtt{T})$ is a theorem and α/b is syntactically equal to σ; β/b is the *result* of the application. Since inference rules of the logic include *equality* and *instantiation*, if σ^* is a result of rewriting σ, then $\sigma = \sigma^*$ (and hence $\sigma \Leftrightarrow \sigma^*$) is a theorem. A *rewriter* applies rules to a term until no rule is applicable. The resulting term is a *normal form*. In general rewriting is a non-deterministic process, but *rewrt* implements rewriting that is principally inside-out (arguments of a term are rewritten before the term), and ordered (rules are applied in a fixed total order). The procedure *rewrt* also incorporates some congruence-based reasoning and gives special treatment to the function `hide`. Terms T_0 and T_1 in Section 2.3 are normal forms.

We now simply define *chop* as the following recursive procedure which traverses the applications of `if` in a term and replaces the non-`if` subterms with new variables while updating the mapping η accordingly.

[2]A definition is of the form $\alpha = \beta$ and is treated as the rewrite rule: $\mathtt{T} \Rightarrow (\alpha = \beta)$.

```
chop(τ,η) ≜ If $\tau$ = (if $\alpha$ $\beta$ $\gamma$)
                let $\langle \eta, \rho_1 \rangle := chop(\alpha, \eta)$
                    $\langle \eta, \rho_2 \rangle := chop(\beta, \eta)$
                    $\langle \eta, \rho_3 \rangle := chop(\gamma, \eta)$
                 Return $\langle \eta,$ (if $\rho_1$ $\rho_2$ $\rho_3$)$\rangle$
            Else If $(\exists v \in dom(\eta) : \eta(v) = \tau)$ Return $\langle \eta, v \rangle$
                Else let $u \notin dom(\eta)$ Return $\langle \eta \cup [u \mapsto \tau], u \rangle$
```

We conclude this description with a note on convergence. The computation of Γ and N need not converge. In practice, we attempt to reach convergence within a user-specified bound. Why not coerce terms on which convergence has not been reached to input predicates? We have found that such coercions typically result in coarse abstraction graphs and spurious failures. We prefer to rely on user control and perform such coercions only via user-guided abstractions.

3.1 Observations and Extensions

Our method primarily relies on rewrite rules to simplify terms. Even in our example in Section 2, equation 5 is critical to rewrite Φ'_0 to T_0. Otherwise, the normal form $\mathsf{T}'_0 \triangleq$ `(+ve (if (E (t+ n)) (C0 n) (C1 n)))` would be classified as an *input predicate* which leads to a spurious failure.

This trivial example illustrates an important aspect of our approach. Equation 5 is a critical but generic "fact" about `+ve` and `if`, independent of the system analyzed. Equation 5, known as an *if-lifting* rule, would usually be stored in a library of common rules. While generic rules can normalize most terms, it is important for scalability that the procedure provide control to facilitate generation of manageable graphs. We now discuss one feature, *user-guided abstraction*, that affords control by coercing terms to input predicates. We omit other features that our implementation supports, such as use of rewriting for assume-guarantee reasoning and case splitting, since we do not use them in the examples in this paper.

User-guided abstraction is achieved via a special function `hide`. In the logic, `hide` is the identity function: `(hide x) = x`. However, the *rewrt* procedure will immediately return any term `(hide` τ`)` as a normal form. To see how this affords coercion, consider a system with components A0, A1, A2, etc., where A0 is specified as follows:

```
1. (A0 (t0))     = 1
2. (A0 (t+ n)) = (if (+ve (A1 n)) (A0 n) 42)
```

A0 is assigned 42 if the previous value of A1 is not a positive integer, and otherwise is unchanged. Consider proving that $P_0 \triangleq$ `(+ve (A0 n))` is an invariant. Our procedure will discover the term $P_1 \triangleq$ `(+ve (A1 n))` and attempt to rewrite `(+ve (A1 (t+ n)))`, thereby possibly exploring other components. But P_1 is irrelevant to the invariance of P_0. This irrelevance can be suggested by the user with the rule: `(+ve (A1 n)) = (hide (+ve (A1 n)))`. Since `hide` is the identity function, proving this rule is trivial. The rule has the effect of "wrapping" `hide` around `(+ve (A1 n))` to create a normal form which is coerced as an input predicate `(hide (+ve (A1 n)))` producing a trivial abstraction graph.

3.2 Reachability Checking

The abstraction graph is checked by reachability analysis. Our reachability implementation is an on-the-fly, breadth-first search. While less efficient than commercial model checkers, our simple checker has been sufficient to verify the examples in Section 4. Note that *any* model checker can be interfaced with our work by translating the abstraction graph to a program understandable by the checker. We have implemented interfaces for VIS [8], Cadence SMV [9], and NuSMV [10]. Our checker also contains additional features to provide user feedback, such as pruning counterexamples to only report predicates that are relevant to the failures in the reachability check.

We have also found that it is important to leverage the predicate discovery procedure to limit exploration of irrelevant paths during search. Recall that user-guided abstraction can reduce nodes in the graph by coercing temporal terms to input predicates. However, the process can increase the number of edges in the graph. To combat this, the abstraction procedure computes for each node p (on-the-fly) a set of *representative input valuations*, that is, valuations of input predicates that are relevant in determining $nbrs(p)$. If τ is coerced to an input using `hide`, it contributes to an edge from p only if some $q \in nbrs(p)$ depends on the input variable corresponding to `(hide` τ`)`. In addition, we *filter* exploration of spurious paths by using *rewrt* to determine provably inconsistent combinations of state and input predicates. For example, assume that for some $s \in dom(N)$, $\Gamma(s) \triangleq$ `(equal (f n) (g n))`, and for $i_0, i_1 \in dom(\Gamma)\backslash dom(N)$, $\Gamma(i_0) \triangleq$ `(equal (f n) (i (t+ n)))` and $\Gamma(i_1) \triangleq$ `(equal (g n) (i (t+ n)))`. Then for node p such that $p(s) =$ `NIL`, filtering avoids exploration of edges in which both i_0 and i_1 are mapped to `T`.

4 Demonstration

In this section, we demonstrate the use of our approach to verify cache coherence protocols. For didactic reasons, we first consider a simple ESI protocol, and show in some detail the rewrite rules used to generate the abstraction graph. We then discuss how the same approach is used to verify a more complicated protocol.

4.1 A Simple ESI Protocol

In our ESI model, an unbounded number of *client* processes communicate with a single controller process to access memory blocks (or cache *lines*). Cache lines consist of addressable data. A client can *read* the data from an address if its cache contains the corresponding line. A client acquires a cache line by sending a *fill* request to the controller; such requests are tagged for Exclusive or Shared access. A client with shared access can only *load* data in the cache line. A client with exclusive access can also *store* data. The controller can request a client to Invalidate or *flush* a cache line and if the line was exclusive then its contents are copied back to memory. The key equations in the ESI model definition are shown in Fig. 1. Functions `mem`, `cache`, `excl`, and `valid` model the following components:

- `(mem c n)` is the content of line `c` in the memory at time `n`.
- `(cache p c n)` is the content of line `c` in the cache of process `p` at time `n`.
- `(valid c n)` is the set of processes having a copy of line `c` at time `n`.
- `(excl c n)` is the set of processes having an exclusive copy of `c` at time `n`.

We model external stimuli with generic unary functions `p`, `op`, `addr`, and `data`:

- `(p n)` is the index of the process scheduled at time `n`
- `(op n)` is the action taken by `(p n)`. It can be `"load"`, `"store"`, `"fille"`, `"fills"`, or `"flush"`; `"fille"` and `"fills"` represent exclusive and shared fill requests.
- If `(op n)` = `"store"`, then `(p n)` writes `(data n)` at `(addr n)` in its cached block.

Notice that we use set and record operations `insert`, `drop`, `get`, `put`, etc., to define the model. This emphasizes the importance of rewrite rules to normalize terms built out of the functions used in the system models. For these operations, such rules are available in ACL2 [2] and the following are some useful rules:

```
(in e (insert a s)) = (or (in e s) (equal e a))
(in e (drop a s))   = (and (in e s) (/= e a))
(get a (put b v r)) = (if (equal a b) v (get a r))
```

The property we verify is *coherence*: reading from an address returns the value most recently written. We specify coherence as an invariant as follows. Let `R` and `A` be generic 0-ary functions representing an arbitrary reading process and an arbitrary address. We then define unary functions `D` and `coherent` in Fig. 2. `(D n)` is the last value that was stored to address `(A)` at time `n`, and `(coherent n)` remains true as long as a load by `(R)` from `(A)` returns `(D n)`. Thus, coherence follows from the proof that `(coherent n)` is an invariant.

The careful reader will notice that in Fig. 1, membership in the set `(excl c n)` is tested using the function `in1`. In the logic, `in1` is simply set membership: `(in1 e s) = (in e s)`, where `(in e s)` returns `T` if `e` is a member of set `s`, else `NIL`. The function `in1` is expected to apply to sets that are either empty or singleton. We utilize this expectation with the following rule:

```
 (mem c (t+ n))
=
 (cond
  ((/= (cline (addr (t+ n)) c)
   (mem c n))
  ((and (equal (op (t+ n)) "flush")
        (in1 (p (t+ n)) (excl c n)))
   (cache (p (t+ n)) n))
  (T (mem c n))))

 (cache p c (t+ n))
=
 (cond
  ((/= (cline (addr (t+ n))) c)
   (cache c n))
  ((/= (p (t+ n)) p)
   (cache p c n))
  ((or (and (equal (op (t+ n)) "fills")
            (empty (excl c n)))
       (and (equal (op (t+ n)) "fille")
            (empty (valid c n))))
   (mem c n))
  ((and (equal (op (t+ n)) "store")
        (in1 p (excl c n)))
   (put (addr (t+ n)) (data (t+ n))
        (cache p c n)))
  (T (cache p c n)))
```

```
 (valid c (t+ n))
=
 (cond
  ((/= (cline (addr (t+ n)) c))
   (valid c n))
  ((and (equal (op (t+ n)) "flush")
        (e-in1 (p (t+ n)) (excl c n)))
   (drop (p (t+ n)) (valid c n)))
  ((or (and (equal (op (t+ n)) "fills")
            (empty (excl c n)))
       (and (equal (op (t+ n)) "fille")
            (empty (valid c n))))
   (insert (p (t+ n)) (valid c n)))
  (T (valid c n)))

 (excl c (t+ n))
=
 (cond
  ((/= (cline (addr (t+ n))) c)
   (excl c n))
  ((and (equal (op (t+ n)) "flush")
        (e-in1 (p (t+ n)) (excl c n)))
   (drop (p (t+ n)) (excl c n)))
  ((and (equal (op (t+ n)) "fille")
        (empty (valid c n)))
   (insert (p (t+ n)) (excl c n)))
  (T (excl c n)))
```

Figure 1: A model of the ESI protocol. Function `cline` is generic; `(cline a)` is assumed to return the index of the cache line containing address `a`. Functions `insert` and `drop` are defined to be set insertion and deletion, `in` and `in1` check set membership, and `empty` is a test for emptyset. Function `put` models "record update", so that `(put a v r)` is record `r` changed to map key `a` to value `v`. `(e-in1 e s)` is defined to be `(or (empty s) (in1 e s))`, and `(/= x y)` is defined to be `(not (equal x y))`.

```
(in1 e s) = (cond ((empty s) nil)
                  ((singleton s) (equal e (choose s)))
                  (T (hide (in1 e s))))
```

Here `(choose s)` returns *some* member `s` if `s` is a non-empty set, and `(singleton s)` checks if `s` is a singleton. This rule shows how rewrite rules and structured definitions can convey protocol-level assumptions (namely, that `(excl c n)` is always empty or singleton) to the abstraction process without limiting expressiveness. Application of the rule causes terms involving `in1` to be rewritten to introduce a case-split for the cases where the set is empty, singleton, or otherwise, and coerces the third case to an input predicate.

With the rules above, our procedure proves that `(coherent n)` is an invariant. The abstraction graph is defined on 9 state predicates (Fig. 3) and 25 input predicates. The search traverses 133 edges exploring 11 nodes and the proof takes a couple of seconds. Without edge pruning, the search explores 48 nodes. Notice that the rule about `in1` is crucial not only to abstract the irrelevant case, but also to *introduce* the relevant state predicate 9; this predicate "tracks" the fact that the value stored in address `(A)` at the local cache of an arbitrary processor (not necessarily `(R)`) at time `n` is equal to `(D n)`. Factors like this have made it difficult for fully automatic decision procedures to abstract "processor indices" in past work in abstraction, and underline the importance of using an expressive logic to define the necessary functions for modeling

```
(D (t0))           = (get (A) (mem (t0)))
(coherent (t0))    = T
(D (t+ n))         = (if (and (equal (addr (t+ n)) (A))
                              (equal (op (t+ n)) "store")
                              (in1 (p (t+ n)) (excl (cline (addr (t+ n))) n)))
                         (data n)
                       (D n))
(coherent (t+ n)) = (if (and (equal (p (t+ n)) (R))
                              (equal (addr (t+ n)) (A))
                              (equal (op (t+ n)) "load")
                              (in (R) (valid (cline (addr (t+ n))) n)))
                         (equal (get (A) (cache (R) (cline (A)) n)) (D n))
                       (coherent n))
```

Figure 2: Definition of functions `D` and `coherent` for the ESI model. Function `get` is the "record access" operation; `(get k r)` returns the value stored with key `k` in record `r`

```
1. (coherent n)
2. (valid (cline (A)) n)
3. (in (R) (valid (cline (A)) n))
4. (excl (cline (A)) n)
5. (singleton (excl (cline (A)) n))
6. (equal (choose (excl (cline (A)) n)) (R))
7. (equal (D n) (get (A) (mem (cline (A)) n)))
8. (equal (D n) (get (A) (cache (R) (cline (A)) n)))
9. (equal (D n) (get (A) (cache (choose (excl (cline (A)) n))
                               (cline (A)) n)))
```

Figure 3: State Predicates Discovered for the ESI Model

target systems.

4.2 A More Elaborate Cache Coherence Protocol

We now consider a more elaborate system and observe how concepts from the ESI model are reused with little "overhead". The system is based on the protocol defined by S. German. In this system, the controller (named *home*), communicates with clients via three channels 1, 2, and 3. Clients make cache requests (*fill requests*) on channel 1. Home grants cache access (*fill responses*) to clients on channel 2; it also uses channel 2 to send invalidation (*flush*) requests. Clients send flush responses on channel 3, sometimes with data.

The *German protocol* has been studied extensively by the formal verification community [11, 12, 13]. The original implementation has single-entry channels. In UCLID, *indexed predicates* were used [14] to verify a version in which channels are modeled as unbounded FIFOs. Our system is inspired by the version with unbounded FIFOs. However, since we have not built rules to reason directly about unbounded FIFOs, we modify the protocol to use channels of bounded size, and prove, in addition to coherence, that the imposed channel bounds are never exceeded in our model. As in our ESI model, we also model the memory.

Our model is roughly divided into three sets of functions specifying the state of the clients, the *home* controller, and the channels. The state of the clients is defined by the following functions:

- `(cache p c n)` is the content of line `c` in the cache of client `p` at time `n`.
- `(valid c n)` is the set of clients having a copy of line `c` at time `n`.
- `(excl c n)` is the set of clients which have exclusive access of `c` at time `n`.

Home maintains a central directory which enables it to "decide" whether it can safely grant exclusive or shared access to a cache line. It also maintains a list of pending invalidate requests it must send, and the state of the memory. The state of *home* is specified by the following functions:

- `(h-valid c n)` is the set of clients which have access to line `c` at time `n`.
- `(h-excl c n)` is the client which has exclusive access to line `c` at time `n`.
- `(curr-cmd c n)` is the pending request for line `c` at time `n`.
- `(curr-client c n)` is the most recent client requesting for line `c` at `n`.
- `(mem c n)` is the value of line `c` in the memory at time `n`.
- `(invalid c n)` is a record mapping client identifiers to the state of a pending invalidate request at time `n`. It can be "`none pending`", or "`pending and not sent`", or "`invalidate request sent`", or "`invalidate response sent`". This function models part of the state of *home* and part of the state of the channels 2 and 3 (namely, invalidate requests and responses).

Finally, the states of the three channels are specified by the following functions (in addition to `invalid` above):

- `(ch1 p c n)` is the requests sent from client `p` for line `c` at time `n`.
- `(ch2-sh c n)` is the set of clients with a shared fill response in channel 2.
- `(ch2-ex c n)` is the set of clients with an exclusive fill response in channel 2.
- `(ch2-data p c n)` is the data sent to client `p` with fill responses.
- `(ch3-data p c n)` is the data sent from client `p` with the invalidate responses.

At any time `n`, one of the following 12 actions is selected to execute nondeterministically: (1) a client sends a shared fill request on channel 1, (2) a client sends an exclusive fill request on channel 1, (3) *home* picks a fill request from channel 1, (4) *home* sends an invalidate request on channel 2, (5) a client sends an invalidate response on channel 3, (6) *home* receives an invalidate response on channel 3, (7) *home* sends an exclusive fill response on channel 2, (8) *home* sends a shared response on channel 2, (9) a client receives a shared fill response from channel 2, (10) a client receives a shared exclusive response from channel 2, (11) a client performs a store, and (12) a client performs a load.

The coherence property we proved for this system is the same as that for ESI (Fig. 2). Although this system is more elaborate than ESI (and hence an inductive invariant, if manually constructed, is very different), the rules from our libraries (including the set and record rules mentioned in Section 4.1) are directly applicable. Further, a lesson learned from the ESI model is reused and we test membership in sets `(ch2-ex c n)` and `(excl c n)` using `in1`. A similar rule is used to cause a case split on the record access operations for `(invalid c n)`. With these rules, our procedure can prove coherence along with the bounded channel invariant. The abstraction graph for coherence is defined by 46 state and 117 input predicates. The reachability check explores 7000 nodes and about 300 thousand edges, and the proof is completed in less than 2 minutes on a modern desktop machine running Linux. The proof of the bounded channel invariant completed in less time on a smaller abstraction graph.

5 Comparison with Related Work

Our method generates *predicate abstractions* using rewriting. Predicate abstraction involves creating an abstract model whose state variables correspond to predicates in the concrete system. The idea is derived from the more general notion of *abstract interpretations* [15]. Graf and Saidi [1] made the idea explicit and used it to verify communication protocols in PVS. Predicate abstractions have been used recently in SLAM [16] and BLAST [17] to verify device drivers and C programs, and in UCLID [18, 14] to verify unbounded state systems.

The key difference between these approaches and ours is in the method employed for *predicate discovery*, that is, computation of the predicates necessary for construction of the abstract model. Predicate discovery in PVS [1, 19] involves on-the-fly validity checks using the theorem prover. While this allows specification of arbitrary formulas as predicates, it can be prohibitively expensive. Other predicate abstraction methods employ a more computational approach. SLAM and BLAST use *boolean programs* with a control-flow skeleton similar to the original system, UCLID uses weakest liberal preconditions and *index variables*, and Das and Dill [20] use counterexample analysis. To our knowledge, all these methods enforce some restriction on the language to express systems and target properties. Our method, on the other hand, is motivated to exploit the expressiveness afforded by allowing predicates to be arbitrary first-order formulas, while still being efficient in practice. In our method, predicate discovery is based on instantiation of *previously proven* rewrite rules. Proving rewrite rules involves human effort; but such proofs are done "off line" and do not contribute to the cost of predicate computation. With an effective library of rules, our method is efficient in practice. Our approach also disentangles heuristics for predicate discovery from the predicate abstraction process. However, the process might need user interaction to determine the necessity of a new rule or definition. We note that while predicate abstractions have been used both in our work and UCLID to verify versions of the German protocol, the difference in expressive power of the two logics makes it difficult to compare them directly. Our method requires user-provided rewrite rules, but also affords greater control over the structure and form of the system definition and the efficiency of predicate discovery.

Our approach is similar in concept to the work of Namjoshi and Khurshan [21]. This method computes predicates by applying syntactic transformations to a formula that represents weakest liberal preconditions; it is also the basis of *indexed predicate discovery* in UCLID [14]. Our approach can be viewed as a focused and scalable implementation of this method using term rewriting for syntactic transformation, with extensions and heuristics to facilitate generation of effective abstractions as required for practical application.

6 Conclusion

We have presented a method for automating deductive proofs of invariants for reactive systems. Our method reduces an invariant proof to the reachability analysis of an abstraction graph which is a form of predicate abstraction of the original system. Manual definition of inductive invariants is not necessary. This makes invariant proofs robust against changes arising from design evolution. The novelty of our method is in the use of term rewriting to *discover* relevant predicates for the construction of the abstraction graph. Since the method is based on symbolic manipulation of terms, it is relatively insensitive to state explosion. Further, our implementation provides features to facilitate control over the search cost of the abstraction graph.

Our approach affords flexibility in predicate discovery by allowing the user to "plug in" different libraries of rewrite rules. This makes it suitable for the verification of a large class of systems, without imposing restrictions on the language used. A key advantage of using deductive reasoning over automatic decision procedures is the expressiveness of the logic. Expressiveness affords succinct system definitions and powerful proof techniques. Our work is geared towards exploiting this advantage, while still providing substantial automation in practice. In our work, we found that most of the rules necessary for effective application of our tool are generic theorems about functions used in modeling the target system, and, in the context of ACL2 proofs, available in existing libraries. Further, the concepts behind necessary "system-specific" abstractions can also be reused for similar systems. Note, however, that the target system must be modeled with some discipline so that rules can be designed to normalize terms built out of functions used in the model. If a component is modeled at a "low level" with functions for which rules are difficult to design, then it might be necessary to provide a more disciplined alternative definition. However, such low-level models are rarely designed manually, but rather are generated by compilers for higher-level languages. Our method is applicable to systems at the level at which they are modeled.

In future work, we plan to apply this method to verify more detailed systems. In particular, we are working on applying the method to verify invariants of a pipelined implementation of the Y86 processor [22] developed at CMU.

Acknowledgments

We thank Jared Davis, Robert Krug, Wilfred Legato, J Strother Moore, Erik Reeber, and Thomas Wahl for several comments and suggestions.

References

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

[2] Kaufmann, M., Sumners, R.: Efficient Rewriting of Data Structures in ACL2. In Borrione, D., kaufmann, M., Moore, J.S., eds.: 3rd ACL2 Workshop. (2002)

[3] Davis, J.: Finite Set Theory based on Fully Ordered Lists. In Kaufmann, M., Moore, J.S., eds.: 5th ACL2 Workshop. (2004)

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

[5] Russinoff, D.: A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon Processor. In: Formal Methods in Computer-Aided Design. Springer LNCS 1954 (2000) 3–36

[6] Brock, B., Kaufmann, M., Moore, J.S.: ACL2 Theorems about Commercial Microprocessors. In: FMCAD 1996. Springer LNCS 1166 (1996) 275–293

[7] (`http://www.cs.utexas.edu/users/sandip/`)

[8] The VIS Group: VIS: A system for Verification and Synthesis. In Alur, R., Henzinger, T., eds.: Computer Aided Verification. Springer LNCS 1102 (1996)

[9] McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)

[10] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Verifier. In: Computer-Aided Verification. LNCS 1633 (1999) 495–499

[11] Pnueli, A., Ruah, S., Zuck, L.: Automatic Deductive Verification with Invisible Invariants. In Margaria, T., Yi, W., eds.: Tools and Algorithms for Construction and Analysis of Systems. Volume Springer LNCS 2031. (2001) 82–97

[12] Emerson, E.A., Kahlon, V.: Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In: Correct Hardware Design and Verification Methods. Volume Springer LNCS 2860. (2002) 247–262

[13] Lahiri, S.K., Bryant, R.E.: Constructing Quantified Invariants via Predicate Abstraction. In Stefen, B., Levi, G., eds.: Verification, Model Checking and Abstraction. Springer LNCS 2937 (2004) 267–281

[14] Lahiri, S.K., Bryant, R.E.: Indexed Predicate Discovery for Unbounded System Verification. In: Computer-Aided Verification. Springer LNCS 3117 (2004) 135–147

[15] Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Approximation or Analysis of Fixpoints. In: Principles of Programming Languages, ACM Press (1977) 238–252

[16] 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. Springer LNCS 2057 (2001) 103–122

[17] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Principles of Programming Languages, ACM Press (2002) 58–70

[18] Lahiri, S.K., Bryant, R.E., Cook, B.: A Symbolic Approach to Predicate Abstraction. In: Computer-Aided Verification. Springer LNCS 2275 (2003) 141–153

[19] Saidi, H., Shankar, N.: Abstract and model check while you prove. In: Computer-Aided Verification. Springer LNCS 1633 (1999) 443–453

[20] Das, S., Dill, D.L.: Counter-example Based Predicate Discovery in Predicate Abstraction. In: FMCAD 2002. Springer LNCS 2517 (2002) 19–32

[21] Namjoshi, K.S., Khurshan, R.P.: Syntactic Program Transformations for Automatic Abstraction. In: Computer-Aided Verification. Springer LNCS 1855 (2000) 435–449

[22] Bryant, R.E., O'Hallaron, D.R.: Computer Systems: A Programmer's Perspective. Prentice Hall (2003)