Combining Theorem Proving with Model Checking through Predicate Abstraction

Sandip Ray
University of Texas at Austin

Rob Sumners
Advanced Micro Devices

Editor's note:
Using theorem-based approaches to prove the invariants of infinite-state reactive systems often demands significant manual involvement. This article presents a new approach in which model checking complements theorem proving, reducing the manual effort involved by transferring user attention from defining inductive invariants to proving rewrite rules. The authors use this approach with ACL2 to verify cache coherence protocols.
—*Magdy S. Abadir, Freescale Semiconductor*

DESPITE SIGNIFICANT ADVANCES in formal-verification techniques during the past decade, a large gap in complexity still exists between practical verification problems and those that state-of-the-art verification tools can handle. Consequently, there has been increasing interest in combining different verification techniques to synergistically leverage their strengths.

In this article, we present a procedure for proving invariants of computing systems that uses a combination of theorem proving and model checking. Invariants are formulas (or predicates) defined on a system's state variables that hold for all the reachable states. Establishing invariants is a central activity in many formal-verification projects: Verifying safety properties is tantamount to proving an invariant, and proofs of liveness properties typically require auxiliary invariance conditions. On the other hand, invariant proving is difficult for theorem-proving and model-checking techniques individually. Theorem proving involves manually strengthening the formula to an inductive invariant—that is, an invariant preserved by every system transition. This requires significant user expertise.[1,2] Model checking involves symbolic or explicit exploration of the reachable states; the state explosion problem limits its scope in practice.

Our procedure automates invariant proofs while imposing no restriction on the expressiveness of the language used to define systems and their properties. The procedure includes lightweight theorem proving to generate a predicate abstraction,[3] which we then explore through model checking. Given system I and finite set P of predicates on the states of I, predicate abstraction constructs abstract system A, whose states correspond to valuations of predicates in P and whose transitions match the projected transitions of I. The reachable states of A define an inductive invariant of I. Our key observation is that we can mine set P from the definition of the transition relation of I by term simplification. Given the transition relation and a conjectured invariant formula Φ, we use term rewriting on their composition to determine the predicates relevant to the invariance of Φ. The rewrite rules specify relationships between the different functions used in the system definitions, and our procedure uses them to control rewriting. We collect such rules from the theorems proven by a theorem prover. The focused use of rewriting provides the primary connection between theorem proving and model checking, and we have developed methodologies and tools to exploit the connection.

In our approach, theorem-proving and model-checking techniques complement one another. We

0740-7475/07/$25.00 © 2007 IEEE Copublished by the IEEE CS and the IEEE CASS

reduce the manual effort in theorem proving by transferring user attention from defining inductive invariants to proving rewrite rules. An inductive invariant is unique to a specific system. On the other hand, rewrite rules are typically generic facts about functions in the system definition that can be reused in different systems (or different design iterations of the same system). By using different rules, we can configure the procedure for completely different systems. Furthermore, model checking can use the semantic information of the predicates to efficiently explore the abstraction. The result is a controllable procedure that provides substantial automation in invariant proofs. (The "Related work" sidebar summarizes some other approaches to predicate abstraction.)

We have implemented our procedure as a tool that interfaces with the ACL2 theorem prover.[4] ACL2 consists of an applicative programming language based on a subset of Common Lisp and a theorem prover for first-order logic with induction. Researchers and formal-verification engineers have used it to prove the correctness of a wide variety of systems, ranging from processor designs to Java programs. ACL2's key advantages for our work include the theorem prover's emphasis on rewriting and the availability of extensive libraries of theorems about data structures such as bit vectors, sets, records, and bags, which the tool can use as rewrite rules. We have used our procedure to verify several systems in ACL2.

Introductory example

Consider trivial system T with two state components, **c0** and **c1**. The initial value of each of the two components is 0; and the following equations, in which $\mathbf{i}'$ is the external stimulus, give their updates at each transition:

$$\mathbf{c0}' = \mathit{if}(\mathbf{i}' \leq 42)\ \mathit{then}\ \mathbf{c1}\ \mathit{else}\ \mathbf{c0} \quad (1)$$

$$\mathbf{c1}' = \mathit{if}(\mathbf{i}' \leq 42)\ \mathit{then}\ \mathbf{c1}\ \mathit{else}\ 42 \quad (2)$$

We use primes to denote the next value of a state variable and for the input stimulus. In ACL2, we define the system in terms of mutually recursive functions (one for each state component), with argument **n** specifying the value at time **n**.[5] We formalize priming with a unary function **t+** (for next time), and we formalize inputs as uninterpreted functions. Thus, in ACL2's Lisp notation, Equations 1 and 2 are written as follows, where **i** is an uninterpreted function:

```
(=(c0 (t + n))
   (if(< = (i (t + n)) 42) (c1 n) (c0 n)))
(=(c1 (t + n))
   (if(< = (i (t + n)) 42) (c1 n) (c0 n) 42))
```

In this article, to make our presentation accessible to readers unfamiliar with Lisp and ACL2, we avoid using Lisp notation. Also, for many commonly used functions, we use self-explanatory, albeit informal names—for example, "*if x then y else z*" instead of "(if x y z)."

An invariant of system T is the formula $\mathrm{T}_0 \triangleq (\mathbf{c0} \leq 42)$; we can prove this by showing that the formula $\Phi \triangleq (\mathbf{c0} \leq 42) \wedge (\mathbf{c1} \leq 42)$ is an inductive invariant. However, instead of manually constructing Φ, we discover the relevant predicate $(\mathbf{c1} \leq 42)$ by rewriting. Assume the following proven theorem, which is a trivial fact about ***if-then-else***:

$$((\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) \leq w) = \mathit{if}\ x\ \mathit{then}\ (y \leq w)\ \mathit{else}\ (z \leq w) \quad (3)$$

Let $\mathbf{T}'_0$ be the term obtained by priming all system variables in $\mathbf{T}_0$. Using Equations 1 and 3 as rewrite rules oriented from left to right, we can rewrite $\mathbf{T}'_0$ as

$$\mathbf{T}'_0{*} \triangleq \mathit{if}(\mathbf{i}' \leq 42)\ \mathit{then}\ (\mathbf{c1} \leq 42)\ \mathit{else}\ (\mathbf{c0} \leq 42) \quad (4)$$

where $\mathbf{T}'_0{*}$ describes how each transition updates $\mathbf{T}_0$.

Analyzing the *if-then-else* structure of $\mathbf{T}'_0{*}$, we find two new predicates, $\mathbf{I_0} \triangleq (\mathbf{i}' \leq 42)$ and $\mathbf{T}_1 \triangleq (\mathbf{c1} \leq 42)$. We classify $\mathbf{I}_0$ as an input predicate and $\mathbf{T}_1$ as a new state predicate. Rewriting $\mathbf{T}'_0$, with Equations 2 and 3 together with the computed fact $(42 \leq 42)$, yields the following, where T is the constant that denotes Boolean truth:

$$\mathbf{T}'_1{*} \triangleq \mathit{if}(\mathbf{i}' \leq 42)\ \mathit{then}\ (\mathbf{c1} \leq 42)\ \mathit{else}\ \mathrm{T}$$

Our abstract system A_T is now defined with two Boolean state variables (for predicates $\mathbf{T}_0$ and $\mathbf{T}_1$), a free Boolean input (corresponding to predicate $\mathbf{I}_0$), and an initial abstract state defined by valuation of $\mathbf{T}_0$ and $\mathbf{T}_1$ at the initial state of T. Reachability analysis on A_T proves $\mathbf{T}_0$ is an invariant.

Procedure

The preceding example, though trivial, introduces the high-level steps in our procedure:

Related work

Predicate abstraction, a method introduced by Graf and Saidi,[1] has been successfully used in verification tools such as Slam[2] and Uclid.[3] Constructing an exact predicate abstraction for a given set of predicates requires an exponential number of validity checks to determine the abstract transition relation. Work in predicate abstraction has thus focused on constructing a sufficient conservative upper bound and investigating ways to make validity checks efficient, using satisfiability- and binary decision diagram (BDD)-based techniques.[2,4–7] Researchers have also attacked a related problem—discovering the relevant set of predicates for producing a sufficient yet tractable abstraction—through refinement guided by counterexamples.[8–10] This technique iteratively refines an abstraction by adding new predicates to eliminate spurious counterexamples generated by model checking. It is effective when the relevant predicates are quantifier free. Recent work has extended predicate discovery to handle quantified predicates, which appear in proofs of infinite-state systems. One method allows predicates to contain quantified variables over a fixed index set.[6,7,11]

A predicate discovery technique proposed by Namjoshi and Kurshan iteratively applies syntactic transformation on the weakest liberal preconditions of the transition relation.[12] The `Discover` procedure in our method is a focused implementation of this idea, with rewriting for syntactic transformation. Our key contributions involve determining how to achieve the appropriate transformations with simplification procedures available in theorem provers, how to scale predicate discovery with domain insights and user guidance, and how to use the semantic information in predicates for abstract state exploration. Namjoshi and Kurshan's method is also the basis for indexed predicate discovery in Uclid.[11] However, whereas Uclid uses syntactic transformation as a heuristic and relies on automatic predicate abstraction based on quantifier instantiation to compute an approximation of the abstract state space, our method focuses on deductive control to generate and explore the abstraction.

References

1. S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," *Proc. 9th Int'l Conf. Computer-Aided Verification* (CAV 97), LNCS 1254, Springer-Verlag, 1997, pp. 72–83.
2. T. Ball and S.K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proc. 8th Int'l SPIN Workshop on Model Checking of Software*, LNCS 2057, Springer-Verlag, 2001, pp. 103–122.
3. R.E. Bryant, S.K. Lahiri, and S.A. Seshia, "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," *Proc. 14th Int'l Conf. Computer-Aided Verification* (CAV 02), LNCS 2404, Springer-Verlag, 2002, pp. 78–92.
4. H. Saidi and N. Shankar, "Abstract and Model Check while You Prove," *Proc. 11th Int'l Conf. Computer-Aided Verification* (CAV 99), LNCS 1633, Springer-Verlag, 1999, pp. 443–453.
5. S.K. Lahiri and R.E. Bryant, "Constructing Quantified Invariants via Predicate Abstraction," *Proc. 5th Int'l Conf. Verification, Model Checking and Abstract Interpretation* (VMCAI 04), LNCS 2937, Springer-Verlag, 2004, pp. 267–281.
6. S.K. Lahiri, R.E. Bryant, and B. Cook, "A Symbolic Approach to Predicate Abstraction," *Proc. 15th Int'l Conf. Computer-Aided Verification* (CAV 03), LNCS 2275, Springer-Verlag, 2003, pp. 141–153.
7. C. Flanagan and S. Qadeer, "Predicate Abstraction for Software Verification," *Proc. 29th ACM SIGPLAN SIGACT Symp. Principles of Programming Languages* (POPL 02), ACM Press, 2002, pp. 191–202.
8. T. Ball et al., "Automatic Predicate Abstraction of C Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 01), ACM Press, 2001, pp. 201–213.
9. S. Chaki et al., "Modular Verification of Software Components in C," *IEEE Trans. Software Engineering*, vol. 30, no. 6, June 2004, pp. 388–402.
10. S. Das and D.L. Dill, "Counter-Example Based Predicate Discovery in Predicate Abstraction," *Proc. 4th Int'l Conf. Formal Methods in Computer-Aided Design* (FMCAD 02), LNCS 2517, Springer-Verlag, 2002, pp. 19–32.
11. S.K. Lahiri and R.E. Bryant, "Indexed Predicate Discovery for Unbounded System Verification," *Proc. 16th Int'l Conf. Computer-Aided Verification* (CAV 04), LNCS 3117, Springer-Verlag, 2004, pp. 135–147.
12. K.S. Namjoshi and R.P. Kurshan, "Syntactic Program Transformations for Automatic Abstraction," *Proc. 12th Int'l Conf. Computer-Aided Verification* (CAV 00), LNCS 1855, Springer-Verlag, 2000, pp. 435–449.

1. Rewrite a primed formula.
2. Mine predicates from the rewritten term's *if-then-else* structure.
3. Replace predicates with Boolean variables to define the abstraction.
4. Apply reachability analysis to this abstraction.

Predicate discovery

The main routines for predicate discovery are **Rewrt**, which rewrites a term, and **Chop**, which casts subterms from the rewritten formula into predicates. **Rewrt** is an equality-based, conditional term rewriter. It takes a term τ and a sequence ε of conditional equations constituting the system definition and theorems and applies these equations, reducing τ to a simpler (but provably equivalent) normal form τ^*. Rewriting is inside-out (the routine rewrites the arguments of τ before rewriting τ), and ordered (the routine applies equations in reverse order of their position in ε). Most general-purpose theorem provers implement a conditional rewriter such as **Rewrt** to simplify terms using user-proven rewrite rules. However, our particular implementation of **Rewrt** differs from a traditional rewriter in that it has features to efficiently support predicate discovery. In particular, **Rewrt** provides fine-grained control mechanisms by giving special treatment to certain functions discussed later.

Chop extracts predicates from rewritten term τ^* produced by **Rewrt**. It recursively explores the top-level *if-then-else* structure of τ^*, collecting the non-*if* subterms. The subterms are classified as either state predicates (which correspond to state variables in the abstract system) or input predicates (which are replaced with free input). A term is an input predicate if it contains a primed system variable. In addition, the user can instruct **Chop** to abstract certain terms to input predicates through the **Rewrt** control mechanisms.

The top-level predicate discovery procedure is called **Discover**. Given a formula Φ, conjectured to be an invariant of a concrete system C, **Discover** iteratively generates the set of state and input predicates S and I as follows: Initially, $S := \{\Phi\}$ and $I := \emptyset$. At any iteration, we

- pick a predicate $\alpha \in S$,
- prime the concrete state variables in α to obtain α',
- apply Rewrt to α' to obtain reduced term α'^*, and
- use chop on α'^* to augment sets S and I.

We iterate until no new predicates are added to S. We then define our abstract system A_C over set $V_s \triangleq \{v_\alpha : \alpha \in S\}$ of Boolean state variables and $V_i \triangleq \{v_\beta : \beta \in I\}$ of input variables. The next-state value for v_α (where $\alpha \in S$) is defined by the term α'^*, with each subterm $\gamma \in S \cup I$ in α'^* replaced by the corresponding variable v_γ. Because all non-*if* subterms of α are replaced by a variable, a Boolean combination of variables of $V_s \cup V_i$ can represent the relation. System A_C is a conservative abstraction of C with respect to the predicates in S.

Despite its simplicity, **Discover** can automate a large class of invariant proofs. The mechanism derives its power from rewrite rules. In general, these rules are conditional equalities that encode user insight about how to simplify terms arising in the domain. Encoding predicate discovery heuristics as rewrite rules gives our approach flexibility and allows sound user extensions.

Earlier, we stated that predicate discovery iterations must converge on a fixed point before we construct the abstract transition relation. However, the iterations might not always converge; **Discover** attempts to achieve convergence within a user-specified upper bound. We could convert the predicates on which **Discover** has not reached convergence to free inputs without affecting soundness. However, because the predicates are arbitrary (possibly quantified) first-order formulas, a concretization function operating on the individual abstract states is not possible.[6] This rules out automatic counterexample-guided predicate discovery, and we depend on user control to determine the generated predicates' usefulness. Because of this, we have found that indiscriminately abstracting terms often leads to coarse abstractions and thus to spurious counterexamples. Instead, we prefer to perform abstractions via the following user-guided control mechanisms.

Fine-grained control

Although **Discover** relies primarily on rewrite rules, it is important to control predicate classification to limit the abstract system's size. For this, we use two control mechanisms: user-guided abstraction and a simple form of assume-guarantee reasoning. We implemented both mechanisms through rewriting and integrated them with **Discover**.

User-guided abstraction lets **Discover** abstract predicates to free inputs, using a special function called ***hide***. The logic defines ***hide*** as the identity function: ***hide***$(x) = x$. However, **Rewrt** and **Chop** give ***hide*** special treatment: Applications of ***hide*** are

not rewritten, and **Chop** classifies a term containing ***hide*** as an input predicate. This allows **Discover** to converge with a small set of state predicates. For instance, consider system H with components **h0**, **h1**, **h2**, and so on, where the update of **h0** is given by

$$\mathbf{h0}' = \mathit{if}\ (\mathbf{h1} \leq \mathbf{h2})\ \mathit{then}\ 42\ \mathit{else}\ \mathbf{h0}$$

Here, subterm ($\mathbf{h1} \leq \mathbf{h2}$) is irrelevant to the invariance of ($\mathbf{h0} \leq 42$). We convey this insight by the following rule, which produces a trivial abstraction with one predicate:

$$\mathbf{h1} = \mathit{hide}(\mathbf{h1})$$

We use hiding not only to abstract irrelevant terms but also to introduce relevant ones.

Our procedure also uses rewriting to emulate limited assume-guarantee reasoning, implemented through another special function, ***force***. Like ***hide***, ***force*** is an identity function. During the invariance proof of Φ, **Rewrt** ignores any term τ containing ***force***, and **Chop** replaces ***force***(τ) with T, thereby assuming the invariance of τ. To complete the proof, we recursively apply the procedure to show that each predicate τ containing ***force*** is an invariant; during the latter proof, we assume the invariance of Φ and rewrite instances of Φ to T. The apparent circularity resolves itself through induction on the sequence of transitions, as is common in assume-guarantee reasoning.

Edge pruning and reachability analysis

We use reachability analysis to explore system A_C generated by **Discover**. In principle, any model checker can solve the reachability problem. However, we leverage our theorem-proving environment to efficiently explore the abstraction.

Recall that ***hide*** abstracts terms involving state variables of C. Although this reduces the abstract states, it increases the number of abstract edges. Most of these abstract edges are false, meaning they correspond to either an inconsistent combination of state and input predicates or an irrelevant combination of input predicates (that is, another combination creates the same transition). For instance, consider state predicate $S_0 \triangleq (\mathbf{C0} = \mathbf{C1})$ and two input predicates, $I_0 \triangleq (\mathbf{C0} = \mathbf{i}')$ and $I_1 \triangleq (\mathbf{C1} = \mathbf{i}')$. For any state of A_C in which S_0 is mapped to nil, the edge E in which both I_0 and I_1 are mapped to T is an inconsistent predicate combination.

Pruning a false edge requires resolving the consistency of Boolean combinations of all state and input predicates produced by **Discover**. To this end, we generate constraints to account for the predicates' semantic information. To generate the constraints, we apply **Rewrt** to each predicate in $S \cup I$ conditionally by assuming Boolean combinations of relevant predicates. Rewriting $\neg S_0$ under the hypothesis I_0 and I_1 exposes the falsity of the edge E. We use the following heuristic to determine relevance: If rewriting does not simplify τ under hypotheses Φ and $\neg\Phi$, then Φ is not relevant to τ. As in predicate discovery, the user can extend constraint generation with rewrite rules. This facility is critical to our procedure's scalability to large systems.

We briefly remark on reachability analysis: The transition relation of A_C is usually a complex Boolean expression (with a large number of input variables introduced by ***hide***) and normally lacks any regularity or structure, making it unsuitable for symbolic model-checking techniques. Furthermore, the reduction by **Discover** of the abstract state space via rewriting factors out the benefits of symmetry or partial-order reduction. We therefore focus on explicit-state model checking of A_C. Our implementation is essentially an efficient, on-the-fly, explicit-state, breadth-first search. In ACL2, we use the Lisp interface to dynamically generate the reachability code (and the edge-pruning constraints) at runtime from the transition relation produced by **Discover**.

Applications

We have used our tool in the verification of cache coherence protocols for unbounded processes. We chose these protocols because they have been widely used as benchmarks for automated abstraction tools. To demonstrate our approach's robustness, we consider two different cache systems: a simple ESI (exclusive, shared, invalidate) system and a model of the German cache protocol.

ESI protocol

In the ESI system, an unbounded number of client processes communicate with a single controller process to access cache lines. A client acquires a cache line by sending a fill request; the requests are tagged as exclusive or shared. A client with shared access can read the cache line's contents; a client with exclusive access can also update the cache line. The controller can request a client to invalidate, or **flush**, a cache line; if the line was exclusive, its contents are copied to memory.

We call our model of the system **esi**. Figure 1 shows the definition of the transition relation of **esi**. The system has four state components—**valid**, **excl**, **cache**, and **mem**—which we model with the following set and record operations: For cache line **c**, **valid(c)** is a set of processes with access to **c**, **excl(c)** is the set of processes with exclusive access, **mem(c)** is a record that maps the addresses in **c** to the data in memory, and **cache(p, c)** returns the contents of cache line **c** in the cache of process **p**.

Our desired property is coherence, meaning that reading by any process from an arbitrary valid address in its cache returns the last value written. This notion involves universal quantification over addresses and process indices. To formalize this quantification in ACL2, we introduce two state variables, **data** and **coherent**, and two uninterpreted Skolem constants, **a** and **p**. Figure 2 shows our specification of coherence, which can be read as follows: "Let **a** be an arbitrary address and **p** be an arbitrary process. State variable **data** stores the most recent value written to **a**, and **coherent** checks that whenever **p** has **a** in its cache and the current action is **load**, the value read is the same as the content of **data**." Thus, we reduce coherence to the invariant that **coherent** always returns T.

To use our tool on the **esi** system, we apply rules about set and record operations from the current ACL2 library.[7] Figure 3 shows three of these rules.

The tool requires the following additional rule to successfully prove the invariant:

in(*e*, **excl**) = *if*(**excl** = ∅) *then* NIL
else if singleton(**excl**) *then*
(*e* = *choose*(**excl**))
else hide(*in*(*e*, **excl**))

The rule encodes the key domain insight about coherence—that **excl** is either empty or singleton. The rule causes membership tests on **excl** to be rewritten to a case split for whether the set is empty, singleton, or otherwise, and abstracts the third uninteresting case to a free input. With this rule, the tool proves coherence by generating an abstract system as defined by predicates, and the search traverses 11 nodes and 133 abstract arcs, completing in seconds. The following are the nine coherence proof predicates for the esi system:

1. **coherent**
2. **valid**(ā) ≠ ∅
3. *in*(p, **valid**(ā))
4. **excl**(ā) ≠ ∅
5. *singleton*(**excl**(ā))
6. *choose*(**excl**(ā)) = p
7. **data** = *rget*(a, **mem**(ā))
8. **data** = *rget*(a, **cache**(p,ā))
9. **data** = *rget*(a, **cache**(*choose*(**excl**(ā)),ā))

```
if (I' = flush) then
        valid'(Ā') = valid(Ā') \ {P'}
        excl'(Ā') = excl(Ā') \ {P'}
        if in(P',excl(Ā')) then
            mem'(Ā') = cache(P',Ā')
(a)

if (I' = fills) ∧ (excl(Ā') = ∅) then
        cache'(P',Ā') = mem(Ā')
        valid'(Ā') = valid(Ā') ∪ {P'}
(b)

if (I'= fille) ∧ (valid'(Ā') = ∅) then
        cache'(P',Ā') = mem'(Ā')
        valid'(Ā') = valid'(Ā') ∪ {P'}
        excl'(Ā') = excl(Ā') ∪ {P'}
(c)

if (I' = store) ∧ in(P',excl(Ā')) then
        cache'(P',Ā') = rset(cache(P',Ā'),A',D')
(d)
```

Figure 1. Transition relation of the esi system. Constants flush (a), fills (b), fille (c), and store (d) represent actions. For address A, Ā represents the cache line containing A. Function rset is the record update operator. The environmental stimulus consists of current operation I′, address A′, and process index P′; if the operation is store, the stimulus additionally includes data D′.

```
if (I' = store) ∧ in(P',excl(Ā')) ∧ (A' = a) then
        data' = D'
if (I' = load) ∧ in(P',valid(Ā')) ∧ (A' = a) ∧ (P' = p) then
        coherent' = (rget(cache(p,ā), a) = data)
```

Figure 2. Coherence specification. Here, p and a are uninterpreted Skolem constants representing an arbitrary process and address, rget is the record access operator, and in tests set membership.

```
in(e, insert(a, s)) = in(e, s) ∨ (e = a)
in(e, drop(a, s)) = in(e, s) ∧ (e  a)
rget(a, rset(b, v, r)) = if(a = b) then v else rget(a, r)
```

Figure 3. Rewrite rules for set and record operations.

The rule just described is instrumental in introducing the relevant state predicate, predicate 9, which checks that the value stored in address **a** of arbitrary process $q \triangleq choose(\text{excl}(\bar{a}))$ is the same as **data**. Discovering the relevance of process q is necessary to relate the **excl** set with the desired coherence property. Such requirements have made it difficult for fully automatic abstraction procedures to abstract process indices, demonstrating the importance of using expressive logic and supporting user extensions.

German protocol

The **esi** system is illustrative but trivial. To demonstrate our tool's scalability, we report results on its application to a more elaborate cache protocol, devised by Steven German. In this protocol, clients communicate with the controller (called *home*) using three channels as follows:

- Clients send cache requests in channel 1.
- The controller (home) grants access and sends invalidate requests in channel 2.
- Channel 3 carries the invalidate acknowledgments.

The German protocol is a more elaborate implementation of ESI. In the German protocol's original version, each channel is single entry.[8] Recent verification projects have extended it with channels modeled as unbounded FIFO buffers.[9] Our model of the German protocol (which we call **german**), is inspired by the unbounded channel version. However, instead of modeling unbounded FIFO buffers, we restrict the channels to be bounded, and we prove, in addition to coherence, an invariant stating that the channel bound is never exceeded by the implementation. We also model the data path and memory.

We prove the same coherence property for **german** as for **esi**. Note that **german** is more elaborate than **esi** (hence, an inductive invariant, if manually constructed, would be very different). However, our tool incurs little extra overhead. The rules in Figure 3 are directly applicable, and the system-specific rules for testing membership of singleton sets carry over to this system. The tool proves the coherence property as an invariant of the protocol in about 2 minutes on a 1.8-GHz Pentium IV desktop machine running GNU/Linux. The abstract system has 46 state predicates and 117 input predicates, and reachability analysis explores 7,000 nodes and traverses about 300,000 arcs.

Our method preserves the expressive power and control afforded by deductive reasoning while benefiting from the automation provided by model-checking approaches. By reducing invariant proofs of (possibly infinite-state) system designs to model checking on a finite abstraction, we avoid the manual effort involved in defining inductive invariants. Furthermore, the use of rewrite rules enables the procedure to be flexible for reasoning about different systems. Admittedly, the benefits depend on the quality of the manually supplied rewrite rules. However, most general-purpose theorem provers contain effective libraries to assist in the process, and we can reuse domain-specific rules. This reusability makes the method robust in iteratively refining a system design, compared with defining inductive invariants, which are extremely sensitive to design changes.

In addition to being flexible, our approach is very efficient in practice when given an appropriate set of rewrite rules. The reader might be surprised by the ability of our tool to efficiently compute invariant proofs with a large number of predicates. For instance, whereas our proof of **german** completes in a couple of minutes with 46 predicates, the Uclid proof of the German protocol generates 29 predicates but requires about 3 hours.[9] Our method's efficiency comes from the carefully controlled use of rewrite rules for discovering predicates and pruning edges.

We are applying the method to prove multithreaded Java Virtual Machine bytecode programs in ACL2. We are also investigating ways to improve the content and detail of feedback provided by our implementation and the abstract counterexample it generates. ■

Acknowledgments

This material is based on work supported by DARPA and the National Science Foundation under grant CNS-0429591, and by the Semiconductor

Research Consortium under grant 02-TJ-1032. We thank John Matthews, J Strother Moore, Vinod Vishwanath, and Thomas Wahl for their many comments, suggestions, and insights.

References

1. R. Joshi et al., "Checking Cache-Coherence Protocols in TLA+," *Formal Methods in Systems Design*, Mar. 2003, vol. 22, no. 2, pp. 125-131.
2. N. Shankar, "Machine-Assisted Verification Using Theorem Proving and Model Checking," *Mathematical Methods in Program Development*, M. Broy, and B. Schieder, eds., *NATO ASI Series F: Computer and Systems Science*, Springer, 1997, vol. 158, pp. 499-528.
3. S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," *Proc. 9th Int'l Conf. Computer-Aided Verification* (CAV 97), LNCS 1254, Springer-Verlag, 1997, pp. 72-83.
4. M. Kaufmann, P. Manolios, and J.S. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic, 2000.
5. D.M. Russinoff, "A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions," *LMS J. Computation and Mathematics*, Dec. 1998, vol. 1, pp. 148-200.
6. S.K. Lahiri and R.E. Bryant, "Constructing Quantified Invariants via Predicate Abstraction," *Proc. 5th Int'l Conf. Verification, Model Checking and Abstract Interpretation* (VMCAI 04), LNCS 2937, Springer-Verlag, 2004, pp. 267-281.
7. M. Kaufmann and R. Sumners, "Efficient Rewriting of Data Structures in ACL2," *Proc. 3rd Int'l Workshop ACL2 Theorem Prover and Its Applications* (ACL2 02), D. Borrione, M. Kaufmann, and J.S. Moore, eds., TIMA Laboratory, 2002, pp. 141-150.
8. A. Pnueli, S. Ruah, and L. Zuck, "Automatic Deductive Verification with Invisible Invariants," *Proc. 7th Int'l Conf. Tools and Algorithms for Construction and Analysis of Systems* (TACAS 01), LNCS 2031, Springer-Verlag, 2001, pp. 82-97.
9. S.K. Lahiri and R.E. Bryant, "Indexed Predicate Discovery for Unbounded System Verification," *Proc. 16th Int'l Conf. Computer-Aided Verification* (CAV 04), LNCS 3117, Springer-Verlag, 2004, pp. 135-147.

The biography of **Sandip Ray** is on p. 122 of this issue.

Rob Sumners is a research engineer at Advanced Micro Devices in Austin, Texas. His research interests include developing algorithms for improving theorem prover efficiency, using theorem proving as a basis of system analysis, and employing formal methods to facilitate functional verification. Sumners has a BS, an MS, and a PhD in electrical and computer engineering from the University of Texas at Austin.

■ Direct questions and comments about this article to Sandip Ray, Dept. of Computer Sciences, University of Texas at Austin, Austin, TX 78712; sandip@cs.utexas.edu.