Counterexample Guided Invariant Discovery for Parameterized Cache Coherence Verification*

Sudhindra Pandav, Konrad Slind, and Ganesh Gopalakrishnan

School of Computing, University of Utah
{sudhindr, slind, ganesh}@cs.utah.edu

Abstract. We propose a heuristic-based method for discovering inductive invariants in the parameterized verification of safety properties. The promise of the method stems from powerful heuristics we have identified for verifying the cache coherence of directory based protocols. The heuristics are based on syntactic analysis of counterexamples generated during verification, combined with simple static analysis of the predicates involved in the counterexamples to construct and refine inductive invariants. The heuristics were effective in filtering irrelevant predicates as well as keeping the sizes of the generated inductive invariants small. Contributions are: (i) the method is an efficient strategy for discovering inductive invariants for practical verification; (ii) the heuristics scaled smoothly from two small to one large cache coherence protocol (of complexity similar to commercial cache coherence protocols); (iii) the heuristics generate relevant auxiliary invariants which are easily verifiable in few seconds; and (iv) the method does not depend on special verification frameworks and so can be adapted for other verification tools. The case studies include German, FLASH, and a new protocol called German-Ring. The properties verified include mutual exclusion and data consistency.

1 Introduction

Parameterized verification methods—which verify systems comprised of multiple identical components for an arbitrary number of these components—are of growing importance in formal verification. Most parameterized verification techniques for safety properties (such as cache coherence) are based on discovering inductive invariants. Despite the large amount of research conducted in this area, there is no general-purpose inductive invariant discovery method that has been shown to be uniformly good across a spectrum of examples. High-level descriptions of large systems contain enough state variables that even after applying common reduction strategies, such as symmetry reduction, abstraction, and efficient fixpoint computation algorithms, the system is far too large for automated verification methods—let alone parameterized methods. Practical verification therefore demands some kind of symbiotic interaction between the user and the automated verification machinery to construct invariants that imply the safety

* Supported by NSF Grant CCR-0219805 and SRC Contract 1031.001.

D. Borrione and W. Paul (Eds.): CHARME 2005, LNCS 3725, pp. 317–331, 2005.

property. Such a verification method should not only help solve the verification problem but also help open a dialog between verification engineers and system designers who may exchange their knowledge about important system invariants.

In this paper, we discuss heuristics that have allowed us to generate invariants that are just strong enough to verify safety properties of cache coherence protocols. We build our heuristics in the context of a decision procedure for the equality fragment of first order logic with uninterpreted functions (**EUF**) [1]. The goal of these heuristics is to (i) cut down the number of invariants that are needed for verifying the proof goal, and (ii) filter out irrelevant facts (predicates) in the formation of inductive invariants. Our starting point is a concrete model of the system and a safety property to be verified. We start the system from an unconstrained state and symbolically simulate it for a single step. We then use an **EUF** decision procedure to check that the next state obtained from symbolic simulation satisfies the safety property, assuming the hypothesis that the start state satisfies it. Naturally, we are bound to get a failure case as we started from an unconstrained start state. We then construct invariants based on syntactic analysis of such failure cases obtained during the verification process. The syntactic analysis of the counterexamples is conceptually simple and can be easily automated. We deploy efficient filtering heuristics to minimize the predicates that make up the invariants. These heuristics, although context-dependent, are a kind of static analysis and may be done (only once) before the verification process starts. The heuristics are intuitive from a designer's point of view and can be automated for any cache coherence protocol. The idea behind the generated invariants is to constrain the start state to be within the set of reachable states such that the safety property holds. The process stops when the safety property and all the invariants are proved. Note that *our method is primarily intended* for verifying the safety property with respect to a model that has been thoroughly debugged through simulation as well as perhaps even formally verified for small non-parametric instances of, say, 3-4 nodes. This fact justifies why a user would react to a counterexample by strengthening the invariant—and not suspecting that the model is incoherent. This mindset as well as division of labor in achieving parametric verification is nothing new.

On simple but realistic examples, our heuristics worked without *any* adaptations; in other cases, the method still offered a structured approach to invariant discovery that had to be adapted only to a mild degree in an example-specific manner. In all three of our case studies[1]—namely the original German protocol [2], the FLASH protocol, and the high-level version of a completely new industrial protocol (which we call *German-Ring*) used in the IBM z990 multibook microprocessor complex [3]—our approach resulted in modestly sized inductive invariants.

We used the UCLID tool [4] for our experiments. UCLID provides a reasonably efficient collection of decision procedures for the logic of *Equality with Uninter-*

[1] The proof scripts, UCLID reference models, and the first author's MS thesis are available at `http://www.cs.utah.edu/formal_verification/charme05_pandav`. Please contact the first author for details.

preted Functions (**EUF**). On our examples, UCLID's runtime was under a few seconds. Our method relies on UCLID's ability to generate concrete counterexamples. These counterexamples are analyzed in order to come up with invariant strengthenings. *Our key contributions are in terms of the manner in which we analyze counterexamples and discover invariant strengthenings.* We believe our methods can be based on other counterexample-generating decision procedures for sufficiently expressive fragments of first-order logic.

1.1 Related Work

Since the work of German [5], if not before, there has been a significant amount of research on automating the discovery of invariants, see [6,7,8,9] for a (non-exhaustive) list of efforts. In spite of the sophistication of these techniques, the process of finding invariants is still mostly manual. Also these methods tend to discover far too many invariants (equivalent to one large invariant with many conjuncts), and there is currently no good way of deciding which ones are useful.

Predicate abstraction based methods [10,11] to construct inductive invariants automatically require complex quantified predicates. Das used predicate abstraction for verifying mutual exclusion for FLASH [12], albeit on a simpler model. Automated predicate discovery [10] tends to discover large predicates, and so cannot be applied for verifying large protocols like FLASH. Lahiri [13] developed a theory of automatically discovering indexed predicates to be used to construct inductive invariants; predicates are iteratively discovered by computing weakest preconditions, which can generate many superfluous predicates at each stage. It requires manual filtering to get rid of useless predicates (which needs human expertise); also, for large protocols like FLASH, the iteration may fail to converge to a fixpoint. The method of invisible invariants [14] is a collection of automated heuristics to construct auxiliary invariants. The heuristics compute the reachable set of states for a finite instance of the system and then generalize to construct an assertion, which is checked for inductiveness. However, the method is only known to work on a restricted class of systems, to which protocols like FLASH do not belong.

For the FLASH protocol, there have been few previous attempts at discovering inductive invariants for the *data consistency property*; namely, Park [15] in the setting of the PVS theorem prover and Chou et.al. [16] in the setting of Murphi. Park also proved *sequential consistency property* for FLASH (delayed mode). Efficient abstraction-based techniques for parameterized verification have been proposed in [16]. These techniques are suggested by a theory based on simulation proofs, by which one can justifiably use "non-interference lemmas", generated from counter examples, to refine the abstract model and prove the safety property. The lemmas are generated from counter example analysis, but the analysis is not syntax-driven, as in our approach. McMillan used compositional model checking for the safety and liveness property verification of the FLASH protocol [17]. The Cadence SMV tool has various built-in abstractions and symmetry reductions to reduce an infinite state system to finite state, which is then model checked. The user has to provide auxiliary lemmas, though few,

and has to decompose the proof to be discharged by symbolic model checking. This requires significant human skill and knowledge for proving conjectures and driving the tool. In our method, we do not need such human intervention in using the tool. Rather, expertise is needed in picking relevant predicates for our filtering heuristics. Fortunately, such intervention occurs at the higher level of protocol design, which can help designers in not only understanding their protocols better, but also in communicating insights at that level to designers. In contrast to proofs done in the context of specialized tools such as Cadence SMV, our method can be employed in the context of more general-purpose tools such as UCLID or CVC-Lite that have **EUF** decision procedures which generate concrete counterexamples. Emerson and Kahlon [18] verified the German protocol by reducing it to a snoopy protocol and then invoking their proposition to automatically verify the reduced snoopy protocol. The reduction is manually performed and requires expertise. It is not clear whether such a method can be applied to FLASH. Recently, Bingham and Hu [19] proposed a new finite-state symbolic model checking algorithm for safety property verification on a broad class of infinite-state transition systems. They presented a method to reduce a conjunctively guarded protocol to a broadcast protocol on which their algorithm can be applied. They automatically verified German's protocol for data consistency within a minute. It is not clear, however, whether such a method can be scaled to work on large protocols like FLASH.

2 Overview of the Invariant Discovery Process

We model a protocol with a set of *state variables* $\mathcal{V}$. The values assigned to state variables characterize the state of the system. We also use a set of *input variables* $\mathcal{I}$, which can be set to arbitrary values on each step of operation. The value assigned to each input variable is nondeterministically chosen from the domain, thus modeling the concurrent nature of the protocol.

A protocol is formalized by $\mathcal{M} = \langle \mathcal{V}, \theta, \Delta \rangle$, a rule-based state machine, where

- $\mathcal{V}$ is a set of *state variables*. A *state* of the system M provides a type-consistent interpretation of the system variables $\mathcal{V}$. Let Σ denote the set of states over $\mathcal{V}$.
- θ is an boolean **EUF** formula describing the set of initial states $I \subseteq \Sigma$.
- Δ is a set of nondeterministic *rules* describing the transition relation $R \subseteq \Sigma^2$. Syntactically, each rule $\delta \in \Delta$ can be expressed as: $g \rightarrow a$, where g is a predicate on state variables and input variables and a is a *next state function (action)* expression. If g holds, a is executed: this assigns next state values to a subset $\mathcal{W}$ of state variables; any other state variables are unchanged when the transition is taken. If the guards of multiple rules hold at the same time, just one of the rules is picked up nondeterministically for execution.

2.1 Syntax Based Heuristics

For all cache coherence protocols that we are aware of—at least a dozen, including industrial ones—cache coherence can be stated as the safety property

$$\forall i, j. ((i \neq j) \wedge cache(i) = exclusive) \Rightarrow cache(j) \neq exclusive$$

The data consistency property of coherence protocols and the invariants we generate also enjoy a syntactically similar shape. Thus our method focuses on properties of the form

$$P : \forall \mathcal{X}. \mathcal{A}(\mathcal{X}) \Rightarrow \mathcal{C}(\mathcal{X}) \tag{2.1}$$

where $\mathcal{X}$ is the set of *index variables* and $\mathcal{A}$ and $\mathcal{C}$ are the antecedent and consequent of the formula, expressed using boolean connectives.

Let $\mathcal{P} = SP \wedge \bigwedge_i Q_i$ be the conjunction of the safety property SP and the invariants Q_i we generate. We can also treat $\mathcal{P}$ as a set of candidate invariants. Initially $\mathcal{P} = \mathcal{SP}$, as we start with empty set of auxiliary invariants. Let D be the decision procedure for the logic of **EUF**. Our method of inductive invariant checking works as follows:

1. Pick a property P from the set $\mathcal{P}$ for verification [2]. Use the decision procedure D to verify that P holds for the initial state of the system.
2. Perform a one-step symbolic simulation of the system, moving from a general symbolic state s to a successor state t according to the transition relation. Use the decision procedure D to verify that the property P holds in the successor state t, assuming the conjunction of invariants $\mathcal{P}$ holds in start state s. We verify a formula of the form $\mathcal{P}(s) \Rightarrow P(t)$. If the result is **true**, we are done with the verification of property P. Otherwise, there are three possible failure cases, determined by the way in which the property can hold in the first state s and not hold in the second state t. The failure case is selected arbitrarily by the decision procedure.
3. Synthesize new formula Q from *syntactic analysis* and heuristics for the corresponding failure case. Add it to the system i.e., $\mathcal{P}' = \mathcal{P} \wedge \mathcal{Q}$; go to (2). The intuition behind the new formula is to introduce a constraint that would not only get rid of the absurd failure (typically a scenario from an unreachable state space), but also trim the search space just enough to prove the property.

We iterate till all the properties in $\mathcal{P}$ are proved to be inductive.

A failure (or a counterexample) is a tuple $\langle \sigma^s, \delta', \sigma^t \rangle$ where σ^s, σ^t gives the start and next state interpretation for the system variables in the start and the next states respectively, and δ' is the (instantiated) transition rule. We say an interpretation σ satisfies a boolean formula F (denoted as $\sigma \models F$) if F is *true* under the interpretation σ. The syntactic evaluation of a formula F under an interpretation σ is denoted by $\langle F \rangle_\sigma$. Before we discuss the analysis of each failure case, a few definitions that we will need in the discussion:

Given an interpretation σ and a boolean formula F, the *satisfying core* of F under interpretation σ ($SC(F, \sigma)$) returns a maximal subformula, F', of F such that $\langle F' \rangle_\sigma \wedge (F' \Rightarrow F)$. The maximal subformula can be easily computed by traversing the syntax tree of F in a top-down manner. For example, if $F =$

[2] We start with the safety property SP. Then select the property in the order in which it is generated to be a potential invariant.

$a_1 \vee a_2 \ldots \vee a_n$ then $SC(F, \sigma) = \bigvee_i \{a_i | \langle a_i \rangle_\sigma = \mathbf{true}\}$. The intuition is to capture as much information from the formula F provided by the interpretation σ that satisfies F.

Similarly, we define the *violating core* of a formula F under interpretation σ to be a maximal subformula, F' such that $\neg\langle F' \rangle_\sigma \wedge (\neg F' \Rightarrow \neg F)$.

The *action core* of a variable v for the transition rule $\delta : g \rightarrow a$ under the interpretation σ is the conjunction of the cores of the guard and the conditions in the nested ITE expression that assigns the next state value in the action a. Before we formally define the action core, we first define the set of boolean conditions in the nested ITE expression that leads to the next state assignment of v. Let

$$C(a(v)) = \begin{cases} \{c\} \cup C(t) \cup C(e) & \text{if } a(v) = ITE(c, t, e) \\ \{\} & \text{otherwise} \end{cases}$$

We divide the above set into two, one set contains conditions that are satisfied in the ITE expression ("then conditions") and other that are not ("else conditions"). Let

$$I(a(v)) = \{c \in C(a(v)) | \langle c \rangle_\sigma = \mathbf{true}\}$$
$$J(a(v)) = \{c \in C(a(v)) | \langle c \rangle_\sigma = \mathbf{false}\}$$

Finally, the action core of a variable v for the rule $\delta : g \rightarrow a$ under the interpretation σ is given by:

$$\begin{aligned} AC(v, \delta, \sigma) = \quad & SC(g, \sigma) \\ & \wedge \textstyle\bigwedge_{c \in I(a(v))} SC(c, \sigma) \\ & \wedge \textstyle\bigwedge_{c \in J(a(v))} \neg VC(c, \sigma) \end{aligned}$$

The action core helps determine the predicates that were responsible for the next state assignment to state variable v by executing the transition rule δ under the interpretation σ. Since the guard g of the rule δ executed has to be satisfied, the satisfying core $SC(g, \sigma)$ is always included in the action core computation. Then, if the assignment expression for state variable v is a nested ITE we also conjunct the satisfying or the violating core of the boolean conditions in the nested ITE that were satisfied or violated respectively for reaching the assignment.

Now we discuss each failure case analysis:

Failure case I $(\sigma^s \models \mathcal{A} \wedge \sigma^s \models \mathcal{C}), (\sigma^t \models \mathcal{A} \wedge \sigma^t \not\models \mathcal{C})$

For this case, it is clear that the state transition rule δ' in question has assigned some of the variables in the consequent $\mathcal{C}$ leading to the failure. Let $S_\mathcal{C}$ be the set of such state variables. For each state variable $v \in S_\mathcal{C}$, we compute the *action core*, $AC(v, \delta', \sigma^s)$. Conjoin these action cores to obtain a formula $\mathcal{G}' = \bigwedge_{v \in S_\mathcal{C}} AC(v, \delta', \sigma^s)$. Let $\mathcal{A}' = SC(\mathcal{A}, \sigma^s)$ be the satisfying core of the antecedent. The idea behind the various *cores* is to minimize the predicates that make up our assertions. At the end of this process, we generate the following assertion

$$\mathcal{A}' \Rightarrow \neg\mathcal{G}' \tag{2.2}$$

The idea behind this formula is to disallow the conditions that lead to the violation of the consequent, if an over-approximation of the antecedent holds.

Failure case II $(\sigma^s \not\models \mathcal{A} \wedge \sigma^s \not\models \mathcal{C})$, $(\sigma^t \models \mathcal{A} \wedge \sigma^t \not\models \mathcal{C})$

In this case, the transition rule has assigned some variable in $\mathcal{A}$, since the truth value of $\mathcal{A}$ went from false to true when going from σ^s to σ^t. However, the failed consequent is just propagated from one state to other. Thus, we seek to suppress those conditions in the guard and action expressions of the rule δ' that led to the next state assignment satisfying the antecedent. We first determine the violating subformula of $\mathcal{C}$, $\mathcal{C}' = VC(\mathcal{C}, \sigma^s)$ (note that $\sigma^s \not\models \mathcal{C}'$, means $\sigma^s \models \neg\mathcal{C}'$). Let $S_{\mathcal{A}}$ be the set of variables in the antecedent that got assigned. Again as in failure-case **I**, for each variable $v \in S_{\mathcal{A}}$ we compute the action core $AC(v, \delta', \sigma^s)$. We then compute the *precondition* $\mathcal{G}' = \bigwedge_{v \in S_{\mathcal{A}}} AC(v, \delta', \sigma^s)$. This was the condition that fired the counterexample rule δ' and led to the next state assignment violating the property of interest. We therefore generate the following assertion to deal with failure case **II**:

$$\neg\mathcal{C}' \Rightarrow \neg\mathcal{G}' \tag{2.3}$$

The basic idea is to not allow a rule propagate the failed consequent to the next state.

Failure case III $(\sigma^s \not\models \mathcal{A} \wedge \sigma^s \models \mathcal{C})$, $(\sigma^t \models \mathcal{A} \wedge \sigma^t \not\models \mathcal{C})$

This case is the rarest, the main reason being that it arises for protocols that are buggy.[3] The transition rule δ' has assigned values to state variables present in both the antecedent and consequent, leading to violation. Under no circumstances, should any transition rule assign values conflicting with the invariance property. This failure case helped us identify modeling errors in our experimental studies.

2.2 Filtering Heuristics

In contrast to the failure analysis above, the heuristics we now discuss are context-dependent and can be applied only on cache coherence protocols. The motivation for them is that the major component of $\mathcal{G}'$ in the assertions 2.2, 2.3, consists of predicates from the guard g'. Large cache coherence protocols like FLASH have guards with many predicates: retaining all predicates from the guard g' in the assertion would be impractical. To remedy this, we filter irrelevant predicates from a guard. We came up with the filtering heuristics based on the empirical observations we made from our case studies.

Rules in cache coherence protocols can be categorized into two classes: *P-rules*, which are initiated by the requesting processor (*home* or *remote*); and *N-rules*, which are initiated by a message from the network. Messages in the network can be classified, as either **request**s or **grant**s. A request message typically is from a caching node to the home node (such as *Get* and *GetX* in FLASH

[3] Parameterized verification is an expensive process and typically should be attempted only after finite-state model-checking has extensively ferreted out bugs.

Table 1. Filtering Heuristics: The numbers in the last column refers to the order in which the predicates must be picked. For example, if the counterexample has a N-rule of request msg type being processed by the home, then we construct assertion by picking predicates on directory variables first. If we are not able to prove this assertion inductive, then we *add* the predicates on environment variables to the assertion and check for inductiveness.

Rule (R)	Msg Type (m)	Client Type (c)	**Filter:** pick predicates on
P-rule	request	home	*local variables*
		remote	*directory variables*
N-rule	request	home	(1) *directory variables*, (2) *environment variables*
		remote	*channel variables describing the* (1) *type* (2) *sender of the msg*
	grant	—	*channel variables describing the msg type*

or *req_shared* and *req_exclusive* in German). A grant message is a message typically sent by *home* node to a *remote* node (such as *Put* and *PutX* in FLASH or *grant_shared* and *grant_exclusive* in German). All non-request messages, which are part of a pending transaction, such as invalidations, invalidation acknowledgments, *etc.* can be regarded as grants.

We also classify the state variables of cache coherence protocols in four types: *local* variables — describing the state of a caching agent such as `cache_state`, `cache_data`, ...; *directory* variables — such as `dir_dirty, excl_granted,...`; *channel* variables — describing the shared communication channels, such as `ch2`, `unet_src`, ...; and *environment* variables — explaining the state of the transaction or global state. For example, the variable `current_command` in the German protocol explains the command that is currently being processed, and the variable `some_others_left` in FLASH which determines whether there are any *shared* copies.

Our filtering heuristics are based on the above classifications, and are summarized in Table 1. The predicates filtered by the heuristics are characterized by the *type* of the state variables on which they are expressed. We tabulate these context-dependent filtering heuristics based on our empirical observations. We found them to be very efficient in constructing invariants. Let us look at an instance how we apply the filtering heuristics. In German, `rule5` treats what happens when the home nodes receives a `inv_ack` message from a remote node. The guard of the rule is:

$$(\texttt{home\_current\_command} \neq \texttt{empty}) \wedge (\texttt{ch2(i)} = \texttt{invalidate\_ack})$$

This rule is a N-rule with message type **grant**. According to Table 1 one must pick predicates on channel variables describing the message type. Thus the relevant predicate from this guard is ($\texttt{ch2}(i) = \texttt{invalidate\_ack}$) and we need not consider the predicate ($\texttt{home\_current\_command} \neq \texttt{empty}$).

As can be seen, the filtering heuristics are a kind of static analysis. The tabular form of filtering heuristics (see Table 1) has resemblance to the tables

that designers use for design cache coherence protocols. Those tables explain the action taken by a processing node for different protocol scenarios. We just order the state variables involved and choose predicates on them from the guard of the counterexample rule. So, these heuristics can be easily developed upon even by the designer which can not only aid the verification process but also encourage co-ordination between a verification expert and a designer in industrial setting.

Other Heuristics. Apart from the above heuristics for filtering predicates from the guard, other simple techniques can be useful:

Specialization: In cache coherence protocols, the home node has a distinguished status; therefore, if the counterexample deals with the home node, then the new invariant should not be generalized for all nodes and is applicable only for the home node.

Consistency Requirement: Sometimes, the right hand side of an assignment to a state variable is another variable. Imagine the property to be verified has a predicate $p = r$ in the consequent, where p, r are term variables. This is common in data consistency properties. Suppose also that $a(p) = q$ where a is the action function for the counterexample rule δ and q is a variable. In such cases, we cannot rely solely on boolean conditions in the guard and ITEs of the action to construct invariants, as the problem lies in the requirement that the state variable q has to be consistent too. The invariant should include a predicate on the consistency of this value. For example, if p and q are term variables and the consequent of the property has the predicate $p = i$, then we construct the invariant of the form $g' \Rightarrow (q = i)$.

3 A Detailed Illustration on the German Protocol

The 'German' directory based protocol was proposed as a verification benchmark by Steven German [2], and it provides a good illustration of our method. Our UCLID model of the protocol extends that developed by Lahiri [20] with a data-path description obtained from the Murphi model in [16]; the model is available from our website. For lack of space, and since the German protocol has been a popular example [16,14,13,18,11], we do not seek to explain the protocol here.

Coherence Property Verification. To start, let the coherence property

$$P : \forall i, j. ((i \neq j) \wedge \mathit{cache}(i) = \mathit{exclusive}) \Rightarrow \mathit{cache}(j) = \mathit{invalid}$$

be symbolically simulated for one step as described in the previous section.

Counterexample 1: The decision procedure returns a counterexample in which the start state satisfies coherence (node i is *invalid* while j is *exclusive*). The client id `cid` chosen for execution is the node i, which receives a `grant_exclusive` message from the `home` node ("home" hereafter). The rule chosen for execution is `rule8`, which changes the cache state of `cid` to exclusive upon receiving this message. This violates coherence after `rule8` is executed.

Analysis: The start state doesn't satisfy both the antecedent of P (since $cache(i) = invalid$) and the consequent (since $cache(j) = exclusive$): thus P is vacuously satisfied. The rule assigns next state value to `cache(cid)` such that the antecedent holds in the next state and the violated consequent just propagates itself from start state to next state. Thus this is a class **II** counterexample as defined in Section 2.1. The boolean guard of the rule (obtained after beta-reduction) is `ch2(cid)` = `grant_ex`. We now let the syntax guide us in constructing a new assertion. First, we compute the violated core of the consequent, which in this case is the consequent itself. So $\mathcal{C}' = (cache(j) = exclusive)$. Then we compute the action core for the state variable, `cache`, which is the only state variable in the antecedent updated in the action of the counterexample rule. Thus $\mathcal{G}' = (ch2(cid) = grant\_ex)$. We now need to eliminate the input variable `cid` from $\mathcal{G}'$. Since the counterexample gives the same interpretation to both i and `cid`, `cid` may be replaced by i. Thus the constructed auxiliary assertion is, according to Formula 2.3:

$$I_1 : \forall\, i, j.\, cache(j) \neq invalid \Rightarrow ch2(i) \neq grant\_ex$$

Filtering heuristics do not apply since $\mathcal{G}'$ has just a single predicate. With I_1 in the system to prune the search space, we again check P for correctness.

Counterexample 2: We now obtain a new counterexample: node i is in exclusive state in the start state (thus satisfying the antecedent of P), while node j is in invalid state (thus satisfying the consequent of P). Thus P holds in the start state. We also have node j receiving a *grant_sh* message from `home`. The client id `cid` chosen for execution is the node j, and the rule is `rule7`. This rule changes the cache state of the client to *shared*, if the client has received a shared grant from home. Thus we have node j in shared state while node i has exclusive rights in the next state, which violates P.

Analysis: This counterexample is of type **I**. The state variable `cache` appears in the consequent and gets updated by the action. We compute the action core for `cache`, which is the guard $ch2(cid) = grant\_sh$. The assertion is built according to the formula 2.2; replacing the input variable `cid` by its corresponding index variable i. The constructed auxiliary assertion is

$$I_2 : \forall\, i, j.\, cache(i) = exclusive \Rightarrow ch2(j) \neq grant\_sh$$

With the auxiliary assertions I_1 and I_2 in the system, the property P is successfully proved. Note that both invariants I_1 and I_2 were constructed by following the recipe suggested in the analysis. We did not need any protocol dependent heuristics or filterings, as the involved guards were of small sizes. Of course, the auxiliary assertions remain to be proved.

Filtering Heuristics. Now we discuss an application of the filtering heuristics. While following our approach in verifying assertion I_2, we obtained a counterexample in an application of `rule9`. The start state has node i in exclusive state and node j is the `current_client`, satisfying the guard of the transition rule. In

the next state the client j has been granted `grant_shared` message by the home node, as mandated by `rule9`, but node i is still in exclusive state, thus violating assertion I_2. This counterexample is of type **I**. The rule describes home granting shared access to a client, if the client has requested shared access, home has not granted exclusive access to any other node, and the response message channel is empty. The calculated precondition $\mathcal{G}'$

$$\texttt{current\_command} = \texttt{req\_sh} \wedge \neg \texttt{exclusive\_granted} \wedge \texttt{ch2}(\texttt{current\_client}) = \texttt{empty}$$

has three boolean predicates. Having all of them in the refined assertion would perhaps be more than needed to construct an inductive version of I_2. Therefore, we use our filtering heuristics to prune $\mathcal{G}'$. The counterexample rule, `rule9`, is an **N-rule** of **request** type being processed by the *home* node. According to the heuristics suggested for **N-rule request** (see Table 1), therefore, the predicate on the directory variable, `exclusive_granted` is chosen, as it is the most crucial one in decision making. The predicate `current_command=req_sh`, which explains the request message, is irrelevant since the concurrent nature of a cache coherence protocol should allow request messages any time while the system is running. Also, the predicate checking the emptiness of the shared channel, `ch2(current_client)=empty`, doesn't yield a global constraint. Therefore,the strengthened assertion I_2 is:

$$I_{2.1} : \forall\, i, j. cache(i) = exclusive \Rightarrow ch2(j) \neq grant\_sh \wedge exclusive\_granted$$

After a few further steps, we arrive at the final version of $I_{2.1}$ (call it $I_{2.n}$):

$$\begin{aligned} I_{2.n} : \forall\, i, j.\ & ((i \neq j) \wedge cache(i) = exclusive) \Rightarrow \\ & \big(ch2(j) \neq grant\_sh \wedge exclusive\_granted \wedge \\ & ch3(j) \neq inv\_ack \wedge ch2(j) \neq inv \wedge \neg inv\_list(j) \wedge \neg sh\_list(j)\big) \end{aligned}$$

The structure of the formula to be synthesized into an inductive invariant can be easily mechanized based on the case analysis of counterexamples. We blindly followed the above counterexample based analysis and the filtering heuristics to construct all the auxiliary invariants for verifying the coherence property.

Data Consistency Verification. The datapath property is

$$\begin{aligned} \forall i.\ & ((\neg exclusive\_granted \Rightarrow (memdata = auxdata)) \wedge \\ & ((cache(i) \neq invalid) \Rightarrow (cache\_data(i) = auxdata))) \end{aligned} \tag{3.1}$$

To verify data consistency, we needed just two additional invariants beyond those discovered to verify the coherence property. Both the invariants were generated from the counterexamples that violated the consistency requirement. We will examine one such invariant. When we ran UCLID to check 3.1, we obtained a counterexample where the start state has node i receiving *grant_ex* message from *home*, but the data variable of the channel `ch2` carrying the message had a value different from `auxdata`. The transition rule was `rule8`. The action of the rule assigns `cache_data` for node i the value possessed by `ch2_data`, which is not `auxdata`, thus violating data consistency. So, following the consistency requirement heuristic (see Section 2.2), we invent the following auxiliary invariant:

$$D_1 := \forall\, i.\, (ch2(i) = grant\_ex) \Rightarrow (ch2\_data(i) = auxdata)\ .$$

4 Summary of Verifications

Besides the German protocol, we have also applied our method to the datapath and controlpath verification of FLASH and the controlpath verification of German-Ring. We now briefly summarize how our method performed on all the verifications.

German. We needed a total of 9 invariants to completely verify the coherence property of German. It took us a day to come up with the invariants. The total time taken by UCLID to prove the properties was 2.16s.

The earlier manual proof by Lahiri needed 29 invariants and took 8 hours for UCLID to finish the verification. Lahiri also applied an indexed predicate discovery method [13] to construct inductive invariants for the German protocol. He derived a single indexed invariant, which required a manually provided predicate on the auxiliary variable `last_granted`. Note that auxiliary variables do not participate in decision making and so such predicates cannot be discovered, unless they are part of the property to be proven. For that reason, our invariants do not depend on auxiliary variables. Lahiri also generated a dual indexed inductive invariant automatically. However, this invariant had 28 predicates, against just 13 needed for constructing our invariants (most of them dual indexed), and took 2813 seconds of UCLID time, as against 2.16 seconds needed for ours.

We also verified data consistency for German; it required two additional invariants. It took couple of hours to modify the model to include datapath variables and finish the verification.

FLASH. The FLASH model was translated from the SMV model of McMillan [17]. We first verified coherence: no two nodes can be in the exclusive state. Surprisingly, no predicates on the directory were needed to prove the safety property except `dir_dirty`; this contrasts with the German coherence property verification which pulled out almost the entire logic of the protocol. This clearly points out that it is a waste of time and effort to generate invariants irrelevant to the proof of the safety property. We also verified data consistency for FLASH. New data variables for the cache, history variables, and auxiliary variables were introduced. These variables do not appear in the guards of rules; however, the data consistency property had predicates on these variables, so our method was effective. Certain counterexamples showed scenarios that seem hard to humanly imagine. For example, FLASH allows parallel operations like *replacement* to occur while another critical transaction is pending. These operations affect important directory variables, and so invariants involving these directory variables had to be strengthened. The filtering heuristics were very highly used in constructing the invariants. Many of the counterexamples had rules of N-rule grant type processed by a $r$emote node, especially involving the scenario where invalidation acknowledgements are pending. Invariants involving directory variables such as `shlist` (keeps track of the nodes having a shared copy of cacheline) and

`real` (keeps track of number of invalidations sent in the system) were difficult to construct as they needed to be precisely strengthened.

It took just 7 invariants [21] to prove the mutex property for FLASH, containing just 9 predicates and UCLID took 4.71s to complete the verification. Surprisingly, none of these invariants needed predicates on directory variables other than `dir_dirty`, thus explaining the fact that we use only the information that would be just enough to imply the safety property. An additional 15 invariants were required to prove the consistency property and UCLID took 18.68s to automatically verify them. This shows the difference in efforts and time needed to verify different safety properties, and how our method efficiently adapts to such verifications by saving tool processing time and human effort. These invariants had predicates on almost all directory variables. Overall, it took us 3 days to discover all the invariants needed to imply the data consistency property from the counterexample guided discovery process.

German-Ring. We applied our method to verify a high-level description of the protocol used in the IBM z990 superscalar multiprocessor [3], provided to us by Steven German. This is an unconventional protocol, where caches communicate by sending messages on a bidirectional ring. The destination node for a message in the ring is computed by arithmetic calculations using `mod`,$\times$ and $\div$.

The invariants were constructed using just the counterexample analysis explained in subsection 2.1, without the need of filtering heuristics. Since the UCLID language doesn't support arithmetic operators like `mod`,$\times, \div$ where the arguments are variables, we could not model the ring topology of the protocol. Instead, we modeled an approximation in which nodes can send messages arbitrarily to any node in the system. However, the rules behind message passing/processing and all state changes were completely modeled as in the high-level specification of the German-Ring protocol. We were able to prove the coherency property, no matter how the caches are arranged.

In the verification, our heuristics generated two invariants sufficient to verify the safety property. It took us two days to complete the entire verification process including modeling of the protocol and generating the invariants.

5 Automation

We now briefly explain how the syntactic analysis of counterexample and the heuristics can be automated.

Automation. Given an interpretation, the computation of satisfiability, violating and action cores can be easily automated. When a property fails, the counterexample returned by a decision procedure is an assignment to variables that are used in the system and property description. This assignment is the interpretation that is used to decide to which failure case the counterexample belongs. The corresponding transition rule is determined and the satisfiability core of the guard is computed for the interpretation. Then we use our *filtering heuristics* (this can be a manual process too) to filter the predicates from

the satisfying core formula of the guard. We would use this filtered formula to construct the invariant. Depending on the failure case, the corresponding core for the antecedent and consequent of the property is also computed. The action core computation for the variables in the property that are assigned in the action of the rule is also computed. Finally, the appropriate invariant is generated by applying the formulas 2.2,2.3. All the steps in the computation, except perhaps filtering heuristics, can be easily automated as they perform basic extraction and manipulation of boolean formulas. Providing a system that automates these steps and also provides a good interface for applying heuristics and backtracking is useful future work.

How to Detect Over-strengthening? A crucial issue is how do we detect whether we are over-strengthening the invariant or not. At present, we do not have a concrete solution to this problem. We detect this in a very crude way when we learn that we are picking the same predicates from the guard of the transition rule involved in the counterexample that has already been used in the invariant constructed so far. This signals that we are moving in circles and should backtrack to the point where we can pick some other predicate suggested by the priority ordering in filtering heuristics.

6 Conclusions

We have discussed new invariant generation techniques for the safety property verification of cache coherence protocols, using a simple counterexample based analysis. Our heuristics have been successfully applied to verify the mutual exclusion and data consistency properties of the German and FLASH cache coherence protocols. We were also pleasantly surprised at how effective they were on the new German-Ring protocol. The invariants that our method generates are sufficient but *lean*: just sufficient to prove the desired properties. Such invariants typically offer sharper insights into the behavior of a system compared to "flooding" the scene with too many invariants.

Industry level cache coherence protocols are too complicated for any current formal verification system to handle automatically. Our heuristics can help tackle this important problem by guiding manual deductive verification of such protocols, and by being able to generate simple auxiliary invariants easily from the counter example analysis. Our method is more general than previous approaches that were often pursued in the context of special verification frameworks. In contrast, our method can be applied in the context of any decision procedure for **EUF** logics that generate concrete counterexamples.

We have focused on constructing auxiliary invariants for safety property verification. We do not know whether such counterexample based analysis can be adapted for liveness property verification. Some of the issues that could be explored in future work are: (1) almost all the steps in the counterexample analysis and the heuristics can be automated; (2) it would be interesting to adapt our methods to *k-step* inductive invariant checking of safety properties for cache coherence protocols.

References

1. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
2. S. German. Personal Communication.
3. T. J. Siegel, E. Pfeffer, and J. A. Magee. The IBM eServer z990 microprocessor. *IBM J. Res. Dev.*, 48(3-4):295–309, 2004.
4. 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. In *CAV'02*, volume 2404 of *LNCS*, pages 78–92. Springer, 2002.
5. S. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Trans. Software Eng.*, 1(1):68–75, 1975.
6. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: Safety.* Springer-Verlag, 1995.
7. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *TACAS'01*, volume 2031 of *LNCS*, pages 113–127. Springer-Verlag, 2001.
8. N.Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, 1997.
9. S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In *CAV'96*, volume 1102 of *LNCS*, pages 323–335. Springer, 1996.
10. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD'02*, volume 2517 of *LNCS*, pages 19–32. Springer, 2002.
11. K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *VMCAI'02*, volume 2294 of *LNCS*, pages 317–330. Springer, 2002.
12. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99*, volume 1633 of *LNCS*, pages 160–171. Springer, 1999.
13. S.K. Lahiri and R. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV'04*, volume 3114 of *LNCS*, pages 135–147. Springer, 2004.
14. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS'01*, volume 2031 of *LNCS*, pages 82–97. Springer, 2001.
15. S. Park and D. L. Dill. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *SPAA'96*, pages 288–296. ACM Press, 1996.
16. P.K. Mannava C.T. Chou and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD'04*, volume 3312 of *LNCS*, pages 382–398. Springer, 2004.
17. K. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME'01*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.
18. E.A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME'03*, volume 2860 of *LNCS*, pages 247–262. Springer, 2003.
19. J. D. Bingham and A. J. Hu. Empirically efficient verification for a class of infinite-state systems. In *TACAS'05*, volume 3440 of *LNCS*, pages 77–92. Springer, 2005.
20. S. Lahiri. Personal Communication.
21. S. Pandav, K. Slind, and G. Gopalakrishnan. Mutual exclusion property verification of FLASH cache coherence protocol. Technical Report UUCS-04-010, School of Computing, University of Utah, 2004.