A Novel Approach to Parameterized Verification of Cache Coherence Protocols

Yongjian Li[1], Kaiqiang Duan[1], Yi Lv[1,3], Jun Pang[2] and Shaowei Cai[1,3]
[1] The State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2] Computer Science and Communications, University of Luxembourg, Luxembourg
[3] University of Chinese Academy of Sciences, Beijing, China

***Abstract*—Parameterized verification of parameterized protocols like cache coherence protocols is an important but hard problem. Our tool paraVerifier handles this hard problem in a unified framework: (1) it automatically discovers auxiliary invariants and the corresponding causal relations from a small reference instance of the verified protocol; (2) the above invariants and causal relation information are automatically generalized into a parameterized form to construct a parameterized formal proof in a theorem prover (e.g., Isabelle). Our method is successfully applied to typical benchmarks including snooping and directory cache coherence protocol benchmarks. The correctness of these protocols is guaranteed by a formal and readable proof which is automatically generated. The notoriously hard FLASH protocol, which is at an industrial scale, is also verified.**

I. Introduction

Verification of parameterized concurrent systems plays an important role in the area of formal verifications, mainly due to the practical importance of such systems. Parameterized systems exist in many important application areas, including cache coherence, security, and network communication protocols. The hardness of parameterized verification is mainly due to the requirement of correctness that the desired properties should hold in any instance of the parameterized system.

The degree of rigorousness and automation are two critical aspects of approaches to parameterized verification. The verification of real-world parameterized systems is, however, rarely both rigorous and automatic. For instance, FLASH protocol is the cache coherence protocol of the Stanford FLASH mutlitprocessor [1]. This protocol is so complex that only a few approaches [2], [3], [4], [5] have successfully verified it so far. Furthermore, all existing successful verification approaches have their downsides. [2] is a theorem proving based approach which requires to construct inductive invariants by hand. The cases of [3] and [4] are similar to [2] that hand-crafted invariants are required to provide by human experts. As a contrast, [5] is a model checking based approach which can be carried out automatically. However, the formal proof can not be obtained from the work of [5]. In order to effectively verify complex parameterized protocols like FLASH protocol, there are two issues need to be addressed. The first one is how to find a set of sufficient and necessary invariants without (or with less) human intervention, which is a core problem in this field. The second one is the rigorousness of the verification. It is preferable to formulate all the verification in a publicly-recognized trust-worthy framework like a theorem prover [4].

In order to solve the parameterized verification in a both automatic and rigorous way, we design a tool called paraVerifier, which is based on a simple but elegant theory. Three kinds of causal relations are introduced, which are essentially special cases of the general induction rule. A so-called consistency lemma is then proposed, which is the cornerstone of our method. Notably, the theory foundation itself is verified as a formal theory in Isabelle, which is the formal library for verifying protocol case studies. The library provides basic types and constant definitions to model protocol cases and lemmas to prove properties.

Our tool paraVerifier is composed of two parts: an invariant finder `invFinder` and a proof generator `proofGen`. Given a protocol $\mathcal{P}$ and a property inv, `invFinder` tries to find useful auxiliary invariants and causal relations which are capable of proving inv. To construct auxiliary invariants and causal relations, we employ heuristics inspired by consistency relation. Also, when several candidate invariants are obtained using the heuristics, we use oracles such as a model checker and an SMT-solver to check each of them under a small reference model of $\mathcal{P}$, and chooses the one that has been verified. After `invFinder` finds the auxiliary invariants and causal relations, `proofGen` generalizes them into parameterized forms, which are then used to construct a completely parameterized formal proof in a theorem prover (e.g., Isabelle) to model $\mathcal{P}$ and to prove the property inv. After the base theory is imported, the generated proof is checked automatically. Usually, a proof is done interactively. Special efforts in the design of the proof generation are made in order to make the proof checking automatic.

It is noteworthy that we make efforts to illustrate the semantical intuition behinds these invariants. Thus our proof product is not only a certification of correctness, but also a comprehensive analysis report of these protocols.

Related work There have been a lot of studies in the field of parameterized verification [6], [7], [8], [9], [10], [3], [4], [11], [12], [13], [5]. Among them, the "invisible invariants" method [8] is an automatic technique for parameterized verification. In this method, auxiliary invariants are computed in a finite system instance to aid inductive invariant checking.

978-1-5090-5142-7/16/$31.00

The CMP method [4] adopts parameter abstraction and guard strengthening to verify a safety property inv of a parameterized system. An abstract instance of the parameterized protocol is constructed by a counter-example-guided refinement process in an informal way. Recently, in [5], a BRAB algorithm is implemented in an SMT-based model checker. It computes over-approximations of backward reachable states that are checked to be unreachable in the parameterized system.

II. PRELIMINARIES

There are three kinds of $variables$: 1) simple identifier, denoted by a string; 2) element of an array, denoted by a string followed by a natural inside a square bracket. E.g., $arr[i]$ indicates the ith element of the array arr; 3) filed of a record, denoted by a string followed by a dot and then another string. E.g., $rcd.f$ indicates the filed f of the record rcd. Each variable is associated with its $type$, which can be an enumeration, natural number, and Boolean.

$Expressions$ and $formulas$ are defined mutually recursively. $Expressions$ can be simple or compound. A simple expression is either a variable or a constant while a compound expression is constructed with the ite(if-then-else) form $f?e_1 : e_2$, where e_1 and e_2 are expressions, and f is a formula. A $formula$ can be an atomic formula or a compound formula. An atomic formula can be a boolean variable or constant, or in the equivalence form $e_1 \doteq e_2$, where e_1 and e_2 are two expressions. A $formula$ can also be constructed by using the logic connectives, including negation (!), conjunction ($\barwedge$), disjunction ($\veebar$), implication ($\dashrightarrow$).

An $assignment$ is a mapping from a variable to an expression, and is denoted with the assigning operation symbol ":=". A $statement$ α is a set of assignments which are executed in parallel, e.g., $x_1 := e_1; x_2 := e_2; ...; x_k := e_k$. If an assignment maps a variable to a (constant) value, then we say it is a $value\text{-}assignment$. We use $\alpha|_x$ to denote the expression assigned to x under the statement α. For example, let α be $\{arr[1] := C; x := false\}$, then $\alpha|_x$ returns $false$. A $state$ is an instantaneous snapshot of its behavior given by a set of value-assignments.

For every expression e and formula f, we denote the value of e (or f) under an state $s :: var \Rightarrow valueType$ as $\mathbb{A}[e, s]$ (or $\mathbb{B}[f, s]$). For the state s and a formula f, we write $s \models f$ to mean $\mathbb{B}[f, s] = true$. Formal semantics of expressions and formulas are given in HOL as usual, which is shown in [14].

For an expression e and a statement $\alpha = x_1 := e_1; x_2 := e_2; ...; x_k := e_k$, we use $\mathsf{vars}(\alpha)$ and e^α to denote the variables to be assigned $\{x_1, x_2, ...x_k\}$ and the expression transformed from e by substituting each x_i with e_i simultaneously. Similarly, for a formula f and a statement $\alpha = x_1 := e_1; x_2 := e_2; ...; x_k := e_k$, we use f^α to denote the formula transformed from f by substituting each x_i with e_i. Moreover, f^α can be regarded as the weakest precondition of formula f w.r.t. statement α, and we denote $\mathsf{preCond}(f, \alpha) \equiv f^\alpha$. Noting that a state transition is caused by an execution of the statement, formally, we define: $s \overset{\alpha}{\twoheadrightarrow} s' \equiv (\forall x \in \mathsf{vars}(\alpha). s'(x) = \mathbb{A}[\alpha|_x, s]) \wedge (\forall x \notin \mathsf{vars}(\alpha). s'(x) = s(x))$.

A $rule$ r is a pair $< g, \alpha >$, where g is a formula and is called the $guard$ of rule r, and α is a statement and is called the $action$ of rule r. For convenience, we denote a rule with the guard g and the statement α as $g \rhd \alpha$. Also, we denote $\mathsf{act}(g \rhd \alpha) \equiv \alpha$ and $\mathsf{pre}(g \rhd \alpha) \equiv g$. If the guard g is satisfied at the state s, then α can be executed, thus, a new state s' is derived, and we say the rule $g \rhd \alpha$ is triggered at s, and transited to s'. Formally, we define: $s \xrightarrow{r} s' \equiv s \models \mathsf{pre}(r) \wedge s \overset{\mathsf{act}(r)}{\twoheadrightarrow} s'$.

A $protocol$ $\mathcal{P}$ is a pair (I, R), where I is a set of $formulas$ and is called the initializing formula set, and R is a set of rules. As usual, the reachable state set of protocol $\mathcal{P} = (I, R)$, denoted as $\mathsf{reachableSet}(\mathcal{P})$, can be defined inductively: (1) a state s is in $\mathsf{reachableSet}(\mathcal{P})$ if there exists a formula $f \in I$, and $s \models f$; (2) a state s is in $\mathsf{reachableSet}(\mathcal{P})$ if there exist a state s_0 and a rule $r \in R$ such that $s_0 \in \mathsf{reachableSet}(\mathcal{P})$ and $s_0 \xrightarrow{r} s$.

A parameterized object(T) is simple a function from a natural number to T, namely of type $nat \Rightarrow T$. For instance, a parameterized formula pf is of type $nat \Rightarrow formula$, and we define $\mathsf{forallForm}(1, pf) \equiv pf(1)$, and $\mathsf{forallForm}((n + 1), pf) \equiv \mathsf{forallForm}(n, pf) \barwedge pf(n + 1)$. $\mathsf{existsForm}(1, pf) \equiv pf(1)$, and $\mathsf{existsForm}((n + 1), pf) \equiv \mathsf{existsForm}(n, pf) \veebar pf(n + 1)$.

Now we illustrate the above definitions by using a simple mutual exclusion protocol with N nodes. Let I, T, C, and E be enumerating values, x, n are simple and array variables, N a natural number, $\mathsf{pini}(N)$ the predicate to specify the inial state, prules(N) a HOL-notation to denote a set of the four rules of the protocol, $\mathsf{mutualInv}(i, j)$ a property that $n[i]$ and $n[j]$ cannot be C at the same time. We want to verify that $\mathsf{mutualInv}(i, j)$ holds for any $i \leq N$, $j \leq N$ s.t. $i \neq j$. Example 1 will be used throughout the paper.

Example 1 *Mutual-exclusion example.*

```
assignN(i)≡n[i]≐I
pini(N) ≡ x≐true ∧ forallForm(N,assignN )
try(i) ≡ n[i] ≐ I ▷ n[i] := T
crit(i) ≡ n[i] ≐ T∧ x ≐ true ▷ n[i] := C; x := false
exit(i) ≡ n[i] ≐ C ▷ n[i] := E
idle(i) ≡ n[i] ≐ E ▷ n[i] := I; x := true
prules(N) ≡ {r. ∃ i. i ≤ N ∧( r=crit(i) ∨ r=exit(i)
∨ r=idle(i) ∨ r=try(i)}
mutualEx(N)≡ (pIni(N), prules(N))
mutualInv(i,j) ≡ ! (n[i]≐ C ⊼ n[j]≐ C)
```

III. CAUSAL RELATIONS AND CONSISTENCY LEMMA

A novel feature of our work lies in that three kinds of causal relations are exploited, which capture whether and how the execution of a particular protocol rule changes the protocol state variables appearing in an invariant. Consider a rule r, a formula f, and a formula set fs, three kinds of causal relations are defined as follows:

Definition 1 *We define the following relations:*

- $\mathsf{invHoldRule_1}(s, f, r) \equiv s \models \mathsf{pre}(r) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;[1]
- $\mathsf{invHoldRule_2}(s, f, r) \equiv s \models f \longleftrightarrow s \models \mathsf{preCond}(f, (\mathsf{act}(r))$;
- $\mathsf{invHoldRule_3}(s, f, r, fs) \equiv \exists f' \in fs.\ (s \models (f' \overline{\wedge}\, \mathsf{pre}(r)) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r)))$;
- $\mathsf{invHoldRule}(s, f, r, fs) \equiv s \models \mathsf{invHoldRule_1}(s, f, r) \vee s \models \mathsf{invHoldRule_2}(s, f, r) \vee s \models \mathsf{invHoldRule_3}(s, f, r, fs)$.

The relation $\mathsf{invHoldRule}(s, f, r, fs)$ defines a causality relation between f, r, and fs, which guarantees that if each formula in fs holds before the execution of rule r, then f holds after the execution of rule r. This includes three cases. 1) $\mathsf{invHoldRule_1}(s, f, r)$ means that after rule r is executed, f becomes true immediately; 2) $\mathsf{invHoldRule_2}(s, f, r)$ states that $\mathsf{preCond}(S, f)$ is equivalent to f, which intuitively means that none of the state variables in f is changed, and the execution of statement S does not affect the evaluation of f; 3) $\mathsf{invHoldRule_3}(s, f, r, fs)$ states that there exist another invariant $f' \in fs$ such that the conjunction of the guard of r and f' implies the precondition $\mathsf{preCond}(S, f)$. We can also view $\mathsf{invHoldRule_{1\_3}}$ as three special kinds of inductive tactics, which can be applied to prove each formula in fs holds at each inductive protocol rule cases. Note that the three kinds of inductive tactics can be applied in a theorem prover, which is the cornerstone of our work. Only after the theorem prover is told which one among the three kinds of tactics is to be used, it can prove automatically. Without the fine-grained classification, the theorem prover cannot solve the proof tasks. In the procedure of automatic proof generation, proofGen generates proof scripts which contain enough application of the three kinds of tactics and guide the theorem prover to finish the proof.

With the invHoldRule relation, we define a consistency relation $\mathsf{consistent}(invs, inis, rs)$ between a protocol $(inis, rs)$ and a set of invariants $invs = \{inv_1, \ldots, inv_n\}$.

Definition 2 *A relation $consistent(invs, inis, rs)$ holds if the following conditions hold: (1) for all formulas $inv \in invs$ and $ini \in inis$ and all states s, $s \models ini$ implies $s \models inv$; (2) for all formulas $inv \in invs$ and rules $r \in rs$ and all states s,* $\mathsf{invHoldRule}(s, inv, r, invs)$.

Example 2 *Let us define a set of auxiliary invariants:*

```
invOnXC(i) ≡ !(x ≐ true ⊼ n[i]≐ C)
invOnXE(i) ≡! (x ≐ true⊼ n[i] ≐ E)
aux1(i,j) ≡! ( n[i]≐ C⊼n[j] ≐ E)
aux2 (i,j) ≡! ( n[i]≐ E⊼n[j]≐ E)
pinvs(N)≡ {f. ∃ i1 i2. i1 ≤ N ∧ i2 ≤ N ∧
i1 ≠ i2 ∧ f =mutualInv i1 i2
∨(∃ i1. i1 ≤ N ∧ f =invOnXC i1)
∨(∃ i1. i1 ≤ N ∧ f= invOnXE i1)
∨(∃ i1 i2. i1 ≤ N ∧ i2 ≤ N ∧ i1 ≠ i2
∧ f = aux1 i1 i2)
∨(∃ i1 i2. i1 ≤ N ∧ i2 ≤ N ∧ i1 ≠ i2
∧ f = aux2 i1 i2) }
```

In the following discussion, we assume that $inv =$ $\mathsf{mutual}(i_1, i_2)$, $r = \mathsf{crit}(iR_1)$, $rs = \mathsf{pinvs}(N)$, *and assumptions $i_1 \leq N$, $i_2 \leq N$, $i_1 \neq i_2$, and $iR_1 \leq N$ hold.*

- $\mathsf{invHoldRule_3}(s, inv, r, invs)$, *where* $i_1 = iR_1$. *Since* $\mathsf{invOnXC}(i_2) \in invs$, $\mathsf{preCond}\ (\mathsf{act}(r), inv) =!(\mathsf{C} \doteq \mathsf{C} \,\overline{\wedge}\, n[i_2] \doteq \mathsf{C})$, *and* $s \models (\mathsf{invOnXC}(i_2) \,\overline{\wedge}\, \mathsf{pre}(\mathsf{crit}(iR_1)))$ *implies* $s \models!(\mathsf{C} \doteq \mathsf{C} \,\overline{\wedge}\, n[i_2] \doteq \mathsf{C})$.
- $\mathsf{invHoldRule_3}(s, inv, r, invs)$, *where* $i_2 = iR_1$. *Since* $\mathsf{invOnXC}(i_1) \in invs$, $\mathsf{preCond}\ (\mathsf{act}(r), inv) =!(n[i_1] \doteq \mathsf{C} \overline{\wedge} \mathsf{C} \doteq \mathsf{C})$, *and* $s \models (\mathsf{invOnXC}(i_2) \overline{\wedge} \mathsf{pre}(\mathsf{crit}(iR_1)))$ *implies* $s \models!(n[i_1] \doteq \mathsf{C} \,\overline{\wedge}\, \mathsf{C} \doteq \mathsf{C})$.
- $\mathsf{invHoldRule_2}(s, inv, r)$, *where* $i_1 \neq iR_1$, *and* $i_2 \neq iR_1$, *since* $\mathsf{preCond}(\mathsf{act}(r), inv) = inv$.

For any invariant $inv \in invs$, inv holds at a reachable state s of a protocol $P = (ini, rs)$ if the consistency relation $\mathsf{consistent}(invs, inis, rs)$ holds. The following lemma formalizes the essence of the aforementioned causal relation, and is called consistency lemma.

Theorem 1 *If $P = (ini, rs)$,* $\mathsf{consistent}(invs, ini, rs)$, *and* $s \in \mathsf{reachableSet}(P)$, *then for any $inv \in invs$, $s \models inv$.*

Theorem 1 is our main tool to apply to prove. Let us recall the proof goal set in Example 1: the mutual exclusion property holds for each reachable state of the mutual-exclusion protocol. In order to prove the goal, we prove a more general result:

Lemma 2 *If $P = (pini(N), prules(N))$ is the protocol listed in example 1, $s \in$* $\mathsf{reachableSet}(P)$, *and $0 < N$, and $pinvs(N)$ is the set of formulas in example 2, then for any inv s.t. $inv \in pinvs(N)$, $s \models inv$.*

Proof: By theorem 1, we only need to prove that parts (1) and (2) of the relation $\mathsf{consistent}(pinvs(N), pini(N), prules(N))$ hold. Part (1) can be checked routinely. Part (2) can be proved by case analysis on a formula $f \in invs$ and a rule $r \in rs$. Example 2 has checked one case: $f = \mathsf{mutual}(i_1, i_2)$, $r = \mathsf{crit}(iR_1)$. Other cases can be analyzed similarly. ■

In order to apply the consistency lemma to prove that a given property inv (e.g., the mutual exclusion property) holds for each reachable state of a protocol $P = (inis, rs)$ (e.g., mutual-exclusion protocol), we need to solve two problems. First, we need to construct a set of auxiliary invariants $invs$ which contains inv and satisfies $\mathsf{consistent}(invs, inis, rs)$. By applying the consistency lemma, we decompose the original problem of invariant checking into that of checking the causal relation between some $f \in invs$ and $r \in rs$. The latter needs case analysis on the form of f and r. Only if a proof script contains sufficient information on the case splitting and the kind of causal relation to be checked in each subcase, Isabelle can help us to automatically check it. How to generate automatically such a proof, which can be run in Isabelle, is the second problem.

Our solutions to the two problems are as follows: Given a protocol, `invFinder` finds all the necessary ground auxiliary invariants from a small instance of the protocol in Murphi.

[1]Here $\longrightarrow$ and $\longleftrightarrow$ are HOL connectives. Throughout this work, we use HOL as our meta-logic, and embed our protocol description in HOL including descriptions of rules and properties.

This step solves the first problem. A table `protocol.tbl` is worked out to store the set of ground invariants and causal relations, which are then used by `proofGen` to create an Isabelle proof script which models and verifies the protocol in a parameterized form. In this step, ground invariants are generalized into a parameterized form, and accordingly ground causal relations are adopted to create parameterized proof commands which essentially proves the existence of the parameterized causal relations. This solves the second problem. At last, the Isabelle proof script is fed into Isabelle to check the correctness of the protocol.

IV. Searching Auxiliary Invariants

Algorithm 1: Algorithm: $invFinder$

Input: Initially given invariants F, a protocol $\mathcal{P} =< I, R >$
Output: A set of tuples which represent causal relations between concrete rules and invariants:

$A \leftarrow F$;
$tuples \leftarrow []$;
$newInvs \leftarrow F$;
while $newInvs$ *is not empty* **do**
  $f \leftarrow newInvs.dequeue$;
  for $r \in R$ **do**
    $paras \leftarrow \mathsf{Policy}(r, f)$;
    for $para \in paras$ **do**
      $cr \leftarrow \mathsf{apply}(r, para)$;
      $newInvOpt, rel \leftarrow \mathsf{coreFinder}(cr, f, A)$;
      $tuples \leftarrow tuples@[< r, para, f, rel >]$;
      if $newInvOpt \neq NONE$ **then**
        $newInv \leftarrow \mathsf{get}(newInvOpt)$;
        newInvs.enqueue(newInv);
        $A \leftarrow A \cup \{newInv\}$;
return $tuples$;

Given a protocol $\mathcal{P}$ and a property set F containing invariant formulas we want to verify, `invFinder` aims to find useful auxiliary invariants and causal relations which are capable of proving any element in F. A set A is used to store all the invariants found up to now, and is initialized as F. A queue $newInvs$ is used to store new invariants which have not been checked, and is initialized as F. A relation table $tuples$ is used to record the causal relation between a parameterized rule in some parameter setting and a concrete invariant. Initially, $tuples$ are set as NULL. invFinder works iteratively in a semi-proving and semi-searching way. In each iteration, the head element f of $newInvs$ is popped, then $\mathsf{Policy}(r, f)$ generates groups of parameters $paras$ according to r and f by some policy. For each parameter $para$ in $paras$, it is applied to instantiate r into a concrete rule cr. Here $\mathsf{apply}(r, para) = r$ if r contains no array-variables and $para = []$; otherwise $\mathsf{apply}(r, para) = r(para_{[1]}, ..., para_{[|para|]})$. Then $coreFinder(cr, f, A)$ is called to check whether a causal relation exists between cr and f; if there is such one relation item, the relation item rel and a formula option $newInvOpt$ is returned; otherwise a run-time error occurs in $coreFinder$, which indicates no proof can be found. In the first case, a tuple $< r, para, f, rel >$ will be inserted into $tuples$; If the formula option $newInvOpt$ is NONE, then no new invariant formula is generated; otherwise $newInvOpt = \mathsf{Some}(f')$ for some formula f', then $\mathsf{get}(newInvOpt)$ returns f', and the new invariant formula f' will be pushed into the queue $newInvs$ and inserted into the invariant set A. The above searching process is executed until $newInvs$ becomes empty. At last, the table $tuples$ is returned.

Here we still use the mutual exclusion protocol to illustrate the main ideas of invFinder. Let $P = (pini(N), prules(N))$ is the protocol listed in example 1, $f = \mathsf{mutualInv}(1, 2)$, and $F = \{f\}$. The output of Algorithm 1 is to construct useful auxiliary invariants in example 2 and causal relations used in Lemma 2. By this example, the parameter generation policy Policy and the core invariant searching function coreFinder will be illustrated in Section IV-A and IV-B.

A. Parameter Generation Policy

Let $r = \mathsf{crit}(i)$ be a parameterized rule. An important research question is: How many groups of rule parameters are needed to instantiate r into concrete rules? The answer will determine how to compute the auxiliary invariants and causal relations between these concrete rules and f for generating a proof. For instance, [1], [2], and [3] are three groups to instantiate r into $\mathsf{crit}(1)$, $\mathsf{crit}(2)$, and $\mathsf{crit}(3)$. However, we need to know, are these three groups of concrete rules sufficient to compute the necessary auxiliary invariants and causal relations, and do we need another group parameter [4] to instantiate r. Roughly speaking, after the generation of concrete rules according to the policy, enough auxiliary invariants and causal relations should be computed to generate a proof a shown in Lemma 2. In detail, through the computation of $\mathsf{coreFinder}(cr, f, A)$ by using $\mathsf{crit}(1)$, $\mathsf{crit}(2)$ and $\mathsf{crit}(3)$ with f, adopting the information generated from the generated auxiliary invariants and causal relations should derive a proof of case on f and crit in Example 2 which also involves three subcases. Here [4] is not necessary because [3] and [4] are "equivalent" by our Policy. Let us explain the reason as follows.

In order to formulate the main ideas of our parameter generation policy, we introduce the concept of permutation modulo to symmetry relation $\simeq^n_m$, and a quotient set of perms^n_m (the set of all n-permutations of m) under the relation. Here an n-permutation of m is ordered arrangement of an n–element subset of an m-element set $I = \{i.0 < i \leq m\}$. We use a list xs with size n to stand for an n-permutation of m. For instance, $[1, 2]$ is a 2-permutation of 3. $xs_{[i]}$ and $|xs|$ denote the $i-$th element and the length of xs respectively. If $xs_{[i]} = i$ for all $i \leq |xs|$, we call it identical permutation.

Definition 3 *Let m and n be two natural numbers, where $n \leq m$, L and L' are two n-permutations of m,*

1) $L \sim^n_m L' \equiv (|L| = |L'| = n) \wedge (\forall i. i < |L| \wedge L_{[i]} \leq m - n \longrightarrow L_{[i]} = L'_{[i]})$.
2) $L \simeq^n_m L' \equiv L \sim^n_m L' \wedge L' \sim^n_m L$.
3) $\mathsf{semiP}(m, n, S) \equiv (\forall L \in \mathsf{perms}^n_m \exists L' \in S. L \simeq^n_m L') \wedge (\forall L \in S. \forall L' \in S. L \neq L' \longrightarrow \neg(L \simeq^n_m L'))$.
4) *A set S is called a quotient of the set perms^n_m under the relation $\simeq^n_m$ if $\mathsf{semiP}(m, n, S)$.*

The definition of relation $\simeq^n_m$ (item 1 and 2 in Definition 3) directly leads to the following lemma.

Lemma 3 *If $L \simeq^n_{m+n} L'$, then for any $0 < i \leq |L|$, any $0 < j \leq m$, $L_{[i]} = j$ if and only if $L'_{[i]} = j$.*

For instance, let $L = [2, 3]$ and $L' = [2, 4]$, then $L \simeq^2_4 L'$. Due to Lemma 3, we can analyze a group of concrete parameters by analyzing only one of them as a representative. Keeping this in mind, let us look at the following lemma, which together with Lemma 3 is the theoretical basis of our policy.

Lemma 4 *Let S be a set s.t. $\mathsf{semiP}(m, n, S)$,*

1) for any $L \in \mathsf{perms}^n_m$, there exists a $L' \in S$ s.t. $L \simeq^n_m L'$.
2) let $L \in S$, $L' \in S$, if $L \neq L'$, then there exists two indices $i \leq m$ and $j \leq n$ such that $L_{[i]} = j$ and $L'_{[i]} \neq j$.

Lemma 4 shows 1) completeness of S w.r.t. the set perms^n_m under the relation $\simeq$, 2) the distinction between two different elements in S. Therefore, S has covered all analysing patterns according to the aforementioned comparing scheme between elements of L with numbers $j < n - m$. Moreover, the case patterns represented by different elements in S are different from each other. This fact can be illustrated by the following example.

Example 3 *Let $m = 3$, $n = 1$, $S = \{[1], [2], [3]\}$ and $\mathsf{semiP}(m, n, S)$, let LR be an element in S, there are three cases:*

1) $LR = [1]$: it is a special case where $LR_{[1]} = 1$;
2) $LR = [2]$: it is a special case where $LR_{[1]} = 2$;
3) $LR = [3]$: it is a special case where $LR_{[1]} \neq 1$ and $LR_{[1]} \neq 2$;

Notice that the above cases are mutually disjoint and their disjunction is a tautology. Besides, [3] and [4] and both special cases where $LR_{[1]} \neq 1$ and $LR_{[1]} \neq 2$, and $[3] \simeq^1_3 [4]$. This is the reason why [4] is not needed to be chosen to instantiate crit.

In Algorithm 1, a concrete formula $cinv$ is popped from the queue $newInvs$, which can be seen as a normalized instantiation of some parameterized formula $pinv$.

Definition 4 *A concrete invariant formula $cinv$ is normalized w.r.t a parameterized invariant $pinv$ if there exists no array variable in $cinv$ and $pinv = cinv$ or there exists an identical permutation LI with $|LI| > 0$ such that $cinv = pinv(1, \ldots |LI|)$;*

For instance, the concrete formula $!(n[1] \doteq C \,\bar{\wedge}\, n[2] \doteq C)$ is obtained by instantiating $\mathsf{mutualInv}(i_1, i_2)$ with $[1, 2]$. Let $cinv$ be a normalized concrete invariant w.r.t. a parameterized invariant $pinv$, pr be a parameterized rule, m be the number of actual parameters occurring in $cinv$, and n be the number of formal parameters occurring in pr, our policy is to compute a quotient of perms^n_m, denoted as $\mathsf{cmpSemiperm}(m + n, n)$, and use its elements as a group of parameters to instantiate pr into a set crs of concrete rules.[2] For instance, for the invariant $!(n[1] \doteq C \,\bar{\wedge}\, n[2] \doteq C)$ (or $\mathsf{mutualInv}(1, 2)$), three groups of parameters [1], [2], [3] are used to instantiate crit respectively. Each of the instantiation results will be used to check which kind of causal relation exists between the invariant and each one of the resulting concrete rules. The checking work is accomplished by $\mathsf{coreFinder}$, which is illustrated in the following subsection.

[2]the details of computing $\mathsf{cmpSemiperm}(m + n, n)$ can be found in [14].

B. Core Searching Algorithm

For a $cinv$ and a rule $r \in crs$, the core part of the $\mathsf{invFinder}$ tool is shown in Algorithm 2. It needs to call two oracles. The first one, denoted by `chk`, checks whether a ground formula is an invariant. Such an oracle can be implemented by translating the formula into a formula in NuSMV, and calling NuSMV as the model checking engine to check whether it is an invariant in a given small reference model of the protocol. If the reference model is too small to check the invariant, then the formula will be checked by Murphi in a big reference model. The second oracle, denoted by `tautChk`, checks whether a formula is a tautology. Such a tautology checker is implemented by translating the formula into a form in the SMT (SAT Modulo Theories) format, and checking it by an SMT solver such as Z3.

Algorithm 2: Core Searching Algorithm: $coreFinder$

Input: r, inv, $invs$
Output: A formula option f, a new causal relation rel
1 $g \leftarrow$the guard of r, $S \leftarrow$the statement of r;
2 $inv' \leftarrow \mathsf{preCond}(inv, S)$;
3 **if** $inv = inv'$ **then**
4 $\quad relItem \leftarrow (r, inv, invRule_2, -)$;
5 $\quad$**return** $(\mathsf{NONE}, relItem)$;
6 **else if** $\mathsf{tautChk}(g \rightarrow inv') = true$ **then**
7 $\quad relItem \leftarrow (r, inv, invRule_1, -)$;
8 $\quad$**return** $(NONE, relItem)$;
9 **else**
10 $\quad candidates \leftarrow \mathsf{subsets}(\mathsf{decompose}(\mathsf{dualNeg}(inv') \,\bar{\wedge}\, g))$;
11 $\quad newInv \leftarrow choose(chk, candidates)$;
12 $\quad relItem \leftarrow (r, inv, invRule_3, newInv)$;
13 $\quad$**if** $isNew(newInv, invs)$ **then**
14 $\quad\quad newInv \leftarrow normalize(newInv)$;
15 $\quad\quad$**return** $(\mathsf{SOME}(newInv), relItem)$;
16 $\quad$**else**
17 $\quad\quad$**return** $(\mathsf{NONE}, relItem)$;

Input parameters of Algorithm 2 include a rule instance r, an invariant inv, a sets of invariants $invs$. The sets $invs$ stores the auxiliary invariants constructed up to now. The algorithm searches for new invariants and constructs the causal relation between the rule instance r and the invariant inv. The algorithm returns a formula option and a causal relation item between r and inv. A formula option value NONE indicates that no new invariant is found while $\mathsf{SOME}(f)$ indicates a new auxiliary invariant f is searched.

Algorithm $coreFinder$ works as follows: after computing the pre-condition inv' (line 2), which is the weakest precondition of the input formula inv w.r.t. S, the algorithm takes further operations according to the cases it faces with:

(1) If $inv = inv'$, meaning that statement S does not change inv, then no new invariant is created, and new causal relation item marked with tag $\texttt{invHoldRule}_2$ is recorded between r and inv.
(2) If tautChk verifies that $g \dashrightarrow inv'$ is a tautology, then no new invariant is created, and the new causal relation item marked with tag $\mathsf{invHoldRule}_1$ is recorded between r and inv.
(3) If neither of the above two cases holds, then a new auxiliary invariant $newInv$ will be constructed, which will make the causal relation $\mathsf{invHoldRule}_3$ to hold. The candidate set is $subsets(decompose(dualNeg(inv') \barwedge g))$, where $decompose(f)$ decompose f into a set of sub-formulas f_i such that each f_i is not of a conjunction form and f is semantically equivalent to $f_1 \barwedge f_2 \barwedge ... \barwedge f_N$. $dualNeg(!f)$ returns f. $subsets(S)$ denotes the power set of S. A proper formula is chosen from the candidate set to construct a new invariant $newInv$. This is accomplished by the `choose` function, which calls the oracle `chk` to verify whether a formula is an invariant in the given reference model. After $newInv$ is chosen, the function $isNew$ checks whether this invariant is new w.r.t. $newInvs$ or $invs$. If this is the case, the invariant will be normalized, and then be added into $newInvs$, and the new causal relation item marked with tag $\texttt{invRule}_3$ will be added into the causal relations. The meaning of the word "new" is modulo to the symmetry relation. E.g., $\mathsf{mutualInv}(1, 2)$ is equivalent to $\mathsf{mutualInv}(2, 1)$.

TABLE I
A FRAGMENT OF OUTPUT OF invFinder

rule	ruleParas	inv	causal relation	f'
crit	[1]	mutualInv(1,2)	invHoldRule3	invOnXC(2)
crit	[2]	mutualInv(1,2)	invHoldRule3	invOnXC(1)
crit	[3]	mutualInv(1,2)	invHoldRule2	

Let us continue the example in the end of subsection IV-A. After the three iterations of computations of coreFinder on crit(1), crit(2), crit(3) with $\mathsf{mutualInv}(1, 2)$, the according output of the invFinder, which is stored in file `mutual.tbl`, is shown in Table I. In the table, each line records the index of a normalized invariant, name of a parameterized rule, the rule parameters to instantiate the rule, a causal relation between the ground invariant and a kind of causal relation which involves the kind and proper formulas f' in need (which are used to construct causal relations $\mathsf{invHoldRule}_3$).

Notice that there is a close correspondence between the three lines in table I and the three case analysis in example 2. Each line in table I is a special one of the cooresponding case in Example 2 if we instantiate iR_1 with LR_1, and i_1 with 1, and i_2 with 2 respectively. Can we generalize the information in the lines on concrete invariants and causal relations into symbolic ones which are key to generate proofs as shown in Example 2?

V. GENERALIZATION

Intuitively, generalization means that a concrete index (formula or rule) is generalized into a set of concrete indices (formulas or rules), which can be formalized by a symbolic index (formula or rules) with side conditions specified by constraint formulas. In order to do this, we adopt a new constructor to model symbolic index or symbolic value $\mathsf{symb}(str)$, where str is a string. We use `N` to denote $symb(”N”)$, which formalizes the size of a parameterized protocol instance. A concrete index i can be transformed into a symbolic one by some special strategy g, namely $\mathsf{symbolize}(g, i) = \mathsf{symb}(g(i))$. In this work, two special transforming function $\mathsf{fInv}(i) = ”iInv”\,\hat{}\,\mathsf{itoa}(i)$ and $\mathsf{fIr}(i) = ”iR”\,\hat{}\,\mathsf{itoa}(i)$, where $\mathsf{itoa}(i)$ is the standard function transforming an integer i into a string. We use special symbols $\mathtt{i_i}$ to denote $\mathsf{symbolize}(fInv, i)$; and $\mathtt{iR_i}$ to denote $\mathsf{symbolize}(fIr, i)$. The former formalizes a symbolic parameter of a parameterized formula, and the latter a symbolic parameter of a parameterized rule. Accordingly, we define $\mathsf{symbolize2f}(g, inv)$ (or $\mathsf{symbolize2r}(g, r)$), which returns the symbolic transformation result to a concrete formula inv (or rule r) by replacing a concrete index i occurring in inv (or r) with a symbolic index $\mathsf{symbolize}(g, i)$.

There are two main kinds of generalization in our work: (1) generalization of a normalized invariant into a symbolic one. The resulting symbolic invariants are used to create definitions of invariant formulas in Isabelle. For instance, !(x $\doteq$ true $\barwedge$ n[1]$\doteq$ C) is generalized into !(x $\doteq$ true $\barwedge$ n[$\mathtt{i_1}$]$\doteq$ C). This kind of generalization is done with model constraints, which specifies that any parameter index should be not greater than the instance size `N`, and parameters to instantiate a parameterized rule (formula) should be different. (2) The generalization of concrete causal relations into parameterized causal relations in Isabelle, which will be used in proofs of the existence of causal relations in Isabelle.

Since the first kind of generalization is simple, we focus on the second kind of generalization, which consists of two phases. Firstly, groups of rule parameters such as [[1],[2],[3]] will be generalized into a list of symbolic formulas such as $[\mathtt{iR_1} \doteq \mathtt{i_1}, \mathtt{iR_1} \doteq \mathtt{i_2}, (\mathtt{iR_1} \neq \mathtt{i_1}) \wedge (\mathtt{iR_1} \neq \mathtt{i_2})]$[3], which stands for case-splittings by comparing a symbolic rule parameter iR_1 and invariant parameters i_1 and i_2. In the second phase, the formula field accompanied with a relation of kind $\mathsf{invHoldRule}_3$ is also generalized by some special strategy.

Now let us look at the first phase, starting with some definitions. Consider a line of concrete causal relation shown in Table I, there is a group of rule parameters LR, and a group of parameters LI occurring in an invariant formula.

Definition 5 *Let LR and LI be two permutations which represent rule parameters and invariant parameters, we define:*

- *symbolic comparison condition generalized from comparing $LR_{[i]}$ and $LI_{[j]}$* : $\mathsf{symbCmp}(LR, LI, i, j) \equiv$

$$\begin{cases} \mathtt{iR_i} \doteq \mathtt{i_j} & \textit{if } LR_{[i]} = LI_{[j]} \quad (1) \\ \mathtt{iR_i} \neq \mathtt{i_j} & \textit{otherwise} \quad (2) \end{cases}$$

- *symbolic comparison condition generalized from comparing $LR_{[i]}$ and with all $LI_{[j]}$* : $\mathsf{symbCaseI}(LR, LI, i) \equiv$

[3] $\mathtt{iR_1} \neq \mathtt{i_1}$ is the abbreviation of $!(\mathtt{iR_1} \doteq \mathtt{i_1})$

$$\begin{cases} symbCmp(LR, LI, i, j) & \text{if } \exists! j.LR_{[i]} = LI_{[j]} \quad (3) \\ forallForm(|LI|, pf) & otherwise \quad (4) \end{cases}$$

where $pf(j) = \mathsf{symbCmp}(LR, LI, i, j)$*, and* $\exists! j.P$ *is a qualifier denoting there exists an unique* j *s.t. property* P*;*

- *symbolic case generalized from comparing* LR *with* LI *:* $\mathsf{symbCase}(LR, LI) \equiv \mathsf{forallForm}(|LR|, pf)$*, where* $pf(i) = \mathsf{symbCaseI}(LR, LI, i)$*;*
- *symbolic partition generalized from comparing all* $LRS_{[k]}$ *with* LI*, where* LRS *is a list of permutations with the same length:* $\mathsf{partition}(LRS, LI) \equiv \mathsf{existsForm}(|LRS|, pf)$*, where* $pf(i) = \mathsf{symbCase}(LRS_i, LI)$*.*

$\mathsf{symbCmp}(LR, LI, i, j)$ defines a symbolic formula generalized from comparing $LR_{[i]}$ and $LI_{[j]}$; $\mathsf{symbCaseI}(LR, LI, i)$ a symbolic formula summarizing the results of comparison between $LR_{[i]}$ and all $LI_{[j]}$ such that $j \leq |LI|$; $\mathsf{symbCase}(LR, LI)$ a symbolic formula representing a subcase generalized from comparing all $LR_{[i]}$ and all $LI_{[j]}$; $\mathsf{partition}(LRS, LI)$ is a disjunction of subcases $\mathsf{symbCase}(LRS_{[i]}, LI)$. Recall the three lines in Table. I:

- when $LR = [1]$, $\mathsf{symbCmp}(LR, LI, 1, 1) = (\mathtt{iR_1} \doteq \mathtt{i_1})$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 1) = (\mathtt{iR_1} \doteq \mathtt{i_1})$ becuase $LR_{[1]} = LI_{[1]}$.
- when $LR = [2]$, $\mathsf{symbCmp}(LR, LI, 1, 2) = (\mathtt{iR_1} \doteq \mathtt{i_2})$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 2) = (\mathtt{iR_1} \doteq \mathtt{i_2})$ becasue $LR_{[1]} = LI_{[2]}$.
- when $LR = [3]$, $\mathsf{symbCmp}(LR, LI, 1, 1) = (\mathtt{iR_1} \neq \mathtt{i_1})$, $\mathsf{symbCmp}(LR, LI, 1, 2) = (\mathtt{iR_1} \neq \mathtt{i_2})$, $\mathsf{symbCase}(LR, LI) = symbCaseI(LR, LI, 1) = (\mathtt{iR_1} \neq \mathtt{i_1}) \wedge (\mathtt{iR_1} \neq \mathtt{i_2})$ because neither $LR_{[1]} = LI_{[1]}$ nor $LR_{[1]} = LI_{[2]}$.
- let $LRS = [[1], [2], [3]]$, $\mathsf{partition}(LRS, LI) = (\mathtt{iR_1} \doteq \mathtt{i_1}) \vee (\mathtt{iR_1} \doteq \mathtt{i_2}) \vee ((\mathtt{iR_1} \neq \mathtt{i_1}) \wedge (\mathtt{iR_1} \neq \mathtt{i_2}))$

The second phase of generalization of concrete causal relations is to generalize the formula inv' accompanied with a causal relation $\mathsf{invHoldRule_3}$ in a line of table I. An index occurring in f' can either occur in the invariant formula, or in the rule. We need to look it up to determine the transformation.

Definition 6 *Let* LI *and* LR *are two permutations,* $\mathsf{find\_first}(L, i)$ *returns the least index* j *s.t.* $L_{[i]} = j$ *if there exists such an index; otherwise returns an error.*

$$\mathsf{lookup}(LI, LR, i) \equiv \begin{cases} \mathtt{i_{find\_first(LI,i)}} & \text{if } i \in LI \quad (5) \\ \mathtt{iR_{find\_first(LR,i)}} & otherwise \quad (6) \end{cases}$$

After the second phase of the generalization, the symbolic cases and the according causal relations on rule crit and mutualInv, which are transferred from Table I in Table II. Here $r = \mathsf{crit}(iR_1)$, and $f = \mathsf{mutualInv}(i_1, i_2)$.

TABLE II
THE RESULT OF GENERALIZING LINES OF TABLE I

rule	inv	case	causal relation	f'
r	f	$\mathtt{iR_1} \doteq \mathtt{i_1}$	invHoldRule3	invOnXC(i_2)
r	f	$\mathtt{iR_1} \doteq \mathtt{i_2}$	invHoldRule3	invOnXC(i_1)
r	f	$(\mathtt{iR_1} \neq \mathtt{i_1}) \wedge$ $(\mathtt{iR_1} \neq \mathtt{i_2})$	invHoldRule2	

VI. Automatic Generation of Isabelle Proof

A formal model for a protocol in a theorem prover like Isabelle includes the definitions of constants and rules and invariants, lemmas, and proofs. Readers can refer to [14] for detailed illustration of the formal proof script. In this section, we focus on the generation of a lemma $\mathtt{critVsinv_1}$[4] on the existence of causal relation between rule $\mathsf{crit}(iR_1)$ and invariant $\mathsf{mutualInv}(i_1, i_2)$ basing on the aforementioned information listed in the Table. II.

In order to generate a lemma such as critVsinv1, proofGen needs the following two kinds of key information: name and parameters of crit and mutualInv, the case analysis and the according sunproofs for the sub-cases. The former can be provided by the result information derived from the first kind of generation. The latter can be provided by the symbolic causal table listed in Table II. Due to length limitation, we illustrate the algorithm for generating a key part of the proof of the lemma $\mathtt{critVsinv1}$: the generation of a subproof (e.g., lines 7-8) according to a symbolic relation tag of $\mathsf{invHoldRule_{1-3}}$, which is shown in Algorithm 3. Input $relTag$ is a symbolic causal relation listed in a line of table after generalization, e.g., the first line listed in Table II.

```
1lemma critVsinv1:
2 assumes a1: ∃ iR1. iR1 ≤ N ∧ r=crit iR1 and
a2: ∃i1 i2. i1≤N ∧ i2 ≤ N ∧ i1≠i2 ∧ f=inv1 i1 i2
3 shows invHoldRule s f r (invariants N)
4 proof -
from a1 obtain iR1 where a1:iR1 ≤ N ∧ r=crit iR1
  by blast
from a2 obtain i1 i2 where a2: i1 ≤ N
∧ i2 ≤ N ∧ i1 ≠ i2 ∧ f=inv1 i1 i2
  by blast
5 have iR1=i1 ∨ iR1=i2 ∨ (iR1 ≠ i1 ∧ iR1 ≠ i2)
by auto
6 moreover{assume b1:iR1=i1
7   have invHoldRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp,
rule_tac x=! (x=true ⊼ n[i2]=C) in exI,auto)qed
8   then have invHoldRule s f r (invariants N) by auto}
9 moreover{assume b1:iR1=i2
10   have invHoldRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp,
rule_tac x=! (x=true ⊼ n[i1]=C in exI,auto)qed
11   then have invHoldRule s f r (invariants N) by auto}
12 moreover{assume b1:(iR1 ≠ i1 ∧ iR1 ≠ i2)
13   have invHoldRule2 s f r
    proof(cut_tac a1 a2 b1, auto) qed
14   then have invHoldRule s f r (invariants N) by auto}
15ultimately show invHoldRule s f r (invariants N) by blast
16qed
```

In the body of function rel2proof, sprintf writes a formatted data to string and returns it. In line 10, $\mathsf{getFormField}(relTag)$ returns the field of formula f' if $relTag = \mathsf{invHoldRule_3}(f')$. rel2proof transforms a symbolic relation tag into a paragraph of proof. For instance, the subproofs shown in lines 7-8, 10-11, and 13-14 in the Lemma critVsinv1 is generated according to the 1st, 2snd, and 3rd lines in Table II. If the tag is among $\mathsf{invHoldRule_{1-2}}$, the transformation is rather

[4]Each computed invariant will be referenced by an internal index i in proofGen, and mutualInv's index is 1. Thus $\mathtt{inv_1}$ is the name for mutualInv to print.

Algorithm 3: Generating a kind of proof which is according with a relation tag of $invHoldRule_{1-3}$: rel2proof

Input: A symbolic causal relation item $relTag$
Output: An Isablle proof: $proof$
if $relTag = invHoldRule_1$ **then**
  $proof \leftarrow$ sprintf
  "have invHoldRule1 f r (invariants N)
  by(cut_tac a1 a2 b1, simp, auto)
  then have invHoldRule f r (invariants N) by blast" ;
else if $relTag = invHoldRule_2$ **then**
  $proof \leftarrow$ sprintf
  "have invHoldRule2 f r (invariants N) by(cut_tac a1 a2 b1, simp, auto)
  then have invHoldRule f r (invariants N) by blast" ;
else
  $f' \leftarrow getFormField(relTag)$;
  $proof \leftarrow$ sprintf
  "have invHoldRule3 f r (invariants N)
  proof(cut_tac a1 a2 b1, simp, rule_tac x=%s in exI,auto)qed
  then have invHoldRule f r (invariants N) by blast" (symbf2Isabelle f')";
return *proof*

straight-forward, otherwise the form f' is assigned by the formula getFormField(relTag), and provided to tell Isabelle the formula which is used to construct the $\mathsf{invHoldRule}_3$ relation.

VII. Experiments

We implement our tool in Ocaml. Experiments are done with typical snooping cache coherence protocol benchmarks such as MESI and MOESI protocol, as well as directory cache coherence protocol benchmarks such as German and FLASH protocol. The detailed codes and experiment data can be found in [14]. Each experiment data includes the paraVerifier instance, invariant sets, and Isabelle proof scripts. Experiment results are summarized in Table III. Among them, German protocol was posted as a challenge to the formal verification community by Steven German, and FLASH protocol is a real-world protocol at an industrial scale.

It is the construction of causal relation with readable invariants that differs our work from any previous work. In detail, the invariants have a clean and neat semantics, which reflect the deep insight of the protocol design. Moreover, we generalize these concrete invariants and causal relations into a parameterized proof, and generate a parameterized proof in Isabelle. The readable Isabelle proof script formally proves these invariants. In this way, these proof scripts with easily readable invariants in our work establish "a chain of evidence" for the correctness of the protocol. Thus, we gain with the highest assurance for the design of the protocol. To the best of knowledge, this work for the first time automatically generates a proof of safety properties of full version of FLASH in a theorem prover without auxiliary invariants manually provided by people.

TABLE III
Verification results on benchmarks.

Protocols	#rules	#invariants	time (sec.)	Memory (MB)
mutualEx	4	5	3.25	7.3
MESI	4	3	2.47	11.5
MOESI	5	3	2.49	23.2
German [4]	13	52	38.67	14
FLASH_nodata	60	152	280	26
FLASH_data	62	162	510	26

VIII. Conclusion

Within paraVerifier, we provide an automatic framework for parameterized verification of cache coherence protocol, The originality of using paraVerifier to verify a protocol lies in the following aspects: (1) instead of creating the needed auxiliary invariants manually, we use invFinder to generate automatically these invariants, which is guided by the heuristics to construct a consistent relation to apply the consistency lemma. (2) instead of formally proving verification goals by hand, we use proofGen to generate automatically proofs to prove the correctness of the protocol. The ultimate correctness of the protocol design is guaranteed by the formally readable proof. Therefore, we can verify the protocol in both an automatic and rigorous way.

As we demonstrate in this work, combining theorem proving with automatic proof generation is promising in the field of formal verification of industrial protocols. Theorem proving can guarantee the rigorousness of the verification results, while automatic proof generation can release the burden of human interaction.

Acknowledgments This work was supported by grants (61170073, 61672503, 61272135, 61572478) from the National Natural Science Foundation of China.

References

[1] J. Kuskin *et al*, "The Stanford FLASH multiprocessor," in *ISCA*, 1994, pp.302–313.
[2] S. Park and D. L. Dill, "Verification of flash cache coherence protocol by aggregation of distributed transactions," in *SPAA*, 1996, pp. 288–296.
[3] K. L. McMillan, "Parameterized verification of the flash cache coherence protocol by compositional model checking," in *CHARME*, 2001, pp. 179–195.
[4] C.-T. Chou *et al*, "A simple method for parameterized verification of cache coherence protocols," in *FMCAD*, 2004, pp. 382–398.
[5] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaidi, "Invariants for finite instances and beyond," in *FMCAD*, 2013, pp. 61–68.
[6] A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in *CAV*, 1996, pp. 184–195.
[7] N. Björner *et al*, "Automatic generation of invariants and intermediate assertions," *Theoretical Computer Science*, **173**(1) 49 – 87, 1997.
[8] T. Arons *et al*, "Parameterized verification with automatically computed inductive assertions." in *CAV*, 2001, pp. 221–234.
[9] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *TACAS*, 2001, pp. 82–97.
[10] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *TACAS*, 2001, pp. 113–127.
[11] S. Pandav *et al*, "Counterexample guided invariant discovery for parameterized cache coherence verification," in *CHARME*, 2005, pp. 317–331.
[12] Y. Lv, H. Lin, and H. Pan, "Computing invariants for parameter abstraction," in *MEMOCODE*, 2007, pp. 29–38.
[13] S. Conchon *et al*, "Cubicle: A parallel SMT-based model checker for parameterized systems," in *CAV*, 2012, pp. 718–724.
[14] "paraVerifier," 2016, https://github.com/paraVerifier/paraVerifier.