A Novel Approach to Parameterized verification of Cache Coherence Protocols

Yongjian Li[1], Kaiqiang Duan[1], Jun Pang[2], Shaowei Cai[1], and Yi Lv[1]

[1] State Key Laboratory of Computer Science, Chinese Academy of Sciences
[2] Computer Science and Communications, University of Luxembourg, Luxembourg

Abstract. Parameterized verification of parameterized protocols like cache coherence protocols is important but hard. 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). The principle underlying the generalization is the symmetry mapping. Our method is successfully applied to typical benchmarks including snoopy-based and directory-based benchmarks. Another novel feature of our method lies in that the final verification result of a protocol is provided by a formal and readable proof.

1 Introduction

Verification of parameterized concurrent systems is interesting in the area of formal methods, 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 model checkers, although powerful in verification of non-parameterized systems, become impractical to verify parameterized systems, as they can verify only an instance of the parameterized system in each execution. A desirable approach is to provide a proof that the correctness holds for any instance.

Related Work There have been a lot of studies in the field of parameterized verification [1,2,3,4,5,6,7,8,9]. Among them, the 'invisible invariants' method [3] is an automatic technique for parameterized verification. In this method, auxiliary invariants are computed in a finite system instance to aid inductive invariant checking. The CMP method [6] 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.

The degree of scalability and automatic are two critical merits of approaches to parameterized verification. In this sense, verification of real-world parameterized systems is still a challenging task. For instance, up to now, the verification of a real-world benchmark FLASH requires human guidance in the existing successful verifications [10,11,6]. In order to effectively verify complex parameterized protocols like FLASH, there are two critical problems. 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. The theory foundation of a parameterized verification technique and its soundness are only discussed in a paper proof style in previous work. It is preferable to formulate all the verification in a publicly-recognized trust-worthy framework like a theorem prover [6]. However, theorem proving in a theorem prover like Isabelle is interactive, not automatical.

In order to solve the parameterized verification in a both automatical 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. Then, a so-called consistency lemma is proposed, which is the cornerstone of our method. Especially, 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 automatically.

2 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 ($\bar{\wedge}$), 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-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 [12].

For an expression e and a statement $\alpha = x_1 := e_1; x_2 := e_2; ...; x_k := e_k$, we use $\mathsf{vars}(\alpha)$ to denote the variables to be assigned $\{x_1, x_2, ...x_k\}$; and use e^α to denote 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 $preCond(f, \alpha) \equiv f^\alpha$. 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 $\mathsf{forallForm}((n+1), pf) \equiv \mathsf{forallForm}(n, pf) \bar{\wedge} pf(n+1)$. $\mathsf{existsForm}(1, pf) \equiv \ pf(1)$, and $\mathsf{existsForm}((n+1), pf) \equiv \mathsf{existsForm}(n, pf) \veebar pf(n+1)$.

Now we use a simple example to illustrate the above definitions by a simple mutual exclusion protocol with N nodes. Let I, T, C, and E be three 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) 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. 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)
```

3 Causal Relations and Consistency Lemma

A novel feature of our work lies in that three kinds of causal relations are exploited, which are essentially special cases of the general induction rule. 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:* $\mathsf{invHoldRule}_1 :: state \times formula \times rule \Rightarrow bool$*,* $\mathsf{invHoldRule}_2 :: state \times formula \times rule \Rightarrow bool$*,* $\mathsf{invHoldRule}_3 :: state \times formula \times rule \times ruleset \Rightarrow bool$*, and* $\mathsf{invHoldRule}_3 :: state \times formula \times rule \times ruleset \Rightarrow bool$*.*

1. $\mathsf{invHoldRule}_1(s, f, r) \equiv s \models \mathsf{pre}(r) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;[3]
2. $\mathsf{invHoldRule}_2(s, f, r) \equiv s \models f \longleftrightarrow s \models \mathsf{preCond}(f, (\mathsf{act}(r))$;
3. $\mathsf{invHoldRule}_3(s, f, r, fs) \equiv \exists f' \in fs$ *s.t.* $s \models (f' \barwedge (\mathsf{pre}(r)) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;
4. $\mathsf{invHoldRule}(s, f, r, fs) \equiv s \models \mathsf{invHoldRule}_1(s, f, r) \vee s \models \mathsf{invHoldRule}_2(s, f, r) \vee s \models \mathsf{invHoldRule}_3(s, f, r, fs)$.

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 $\mathsf{invHoldRule}(s, f, r, fs)$ as a special kind 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 done by a theorem prover, which is the cornerstone of our work.

[3] Here $\longrightarrow$ and $\longleftrightarrow$ are HOL connectives.

With the $\mathsf{invHoldRule}$ 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. *We define a relation consistent* $::$ $formula\ set \times formula\ set \times rule\ set \Rightarrow bool$*.* $consistent(invs, inis, rs)$ *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. ∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f =mutualInv iInv1 iInv2)
∨(∃ iInv1. iInv1 ≤ N ∧ f =invOnXC iInv1)
∨(∃ iInv1. iInv1 ≤ N ∧ f= invOnXE iInv1)
∨(∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f = aux1 iInv1 iInv2)
∨(∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f = aux2 iInv1 iInv2) }.
```

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 $i_1 \neq N$, $i_2 \neq N$, $i_1 \neq i_2$, and $iR_1 \leq N$ hold.

- $\mathsf{invHoldRule}_2(s, inv, r)$, where $i_1 \neq iR_1$, and $i_2 \neq iR_1$, since $\mathsf{preCond}(\mathsf{act}(r), inv) = inv$.
- $\mathsf{invHoldRule}_3(s, inv, r, invs)$, where $i_1 = iR_1$. Since $\mathsf{invOnXC}(i_2) \in invs$, $\mathsf{preCond}$ $(\mathsf{act}(r), inv)$ $=!(\mathsf{C} \doteq \mathsf{C} \barwedge n[i_2] \doteq \mathsf{C})$, and $s \models (\mathsf{invOnXC}(i_2) \barwedge \mathsf{pre}(\mathsf{crit}(iR_1)))$ implies $s \models !(\mathsf{C} \doteq \mathsf{C} \barwedge n[i_2] \doteq \mathsf{C})$.
- $\mathsf{invHoldRule}_3(s, inv, r, invs)$, where $i_2 = iR_1$. Since $\mathsf{invOnXC}(i_1) \in invs$, $\mathsf{preCond}$ $(\mathsf{act}(r), inv)$ $=!(n[i_1] \doteq \mathsf{C} \barwedge \mathsf{C} \doteq \mathsf{C})$, and $s \models (\mathsf{invOnXC}(i_2) \barwedge \mathsf{pre}(\mathsf{crit}(iR_1)))$ implies $s \models !(n[i_1] \doteq \mathsf{C} \barwedge \mathsf{C} \doteq \mathsf{C})$.

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.

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

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 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. 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.

4 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:
1  $A \leftarrow F$;
2  $tuples \leftarrow []$;
3  $newInvs \leftarrow F$;
4  while $newInvs$ is not empty do
5      $f \leftarrow newInvs.dequeue$;
6      for $r \in R$ do
7          $paras \leftarrow \mathsf{Policy}(r, f)$;
8          for $para \in paras$ do
9              $cr \leftarrow \mathsf{apply}(r, para)$;
10             $newInvOpt, rel \leftarrow \mathsf{coreFinder}(cr, f, A)$;
11             $tuples \leftarrow tuples@[< r, para, f, rel >]$;
12             if $newInvOpt \neq NONE$ then
13                 $newInv \leftarrow \mathsf{get}(newInvOpt)$;
14                 newInvs.enqueue(newInv);
15                 $A \leftarrow A \cup \{newInv\}$;
16 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.

In Algorithm 1, the parameter generation policy Policy and the core invariant searching function $\mathsf{coreFinder}$ will be illustrated in Section 4.1 and 4.2.

4.1 Parameter Generation Policy

In order to formulate 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 lists which stand for 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 2. *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 2, 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 2 is the theoretical basis of our policy.

Lemma 3. *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 3 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 $m = 2$, $n = 1$, $S = \{[1], [2], [3]\}$ 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 $LR_{[1]} \neq 2$.

Note that the above cases are mutually disjoint and their disjunction is true.

In Algorithm 1, a concrete formula *cinv* is poped 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 exits an identical permutation LI with $|LI| > 0$ such that $cinv = pinv(1, \ldots\ |LI|)$;*

Any normalized *cinv* containing array variables is obtained by instantiating a parameterized invariant *pinv* with a parameter list which is an identical permutation LI (i.e., the j^{th} parameter is j itself $LI_{[j]} = j$). Thus, consider a list of parameter LR which is used to instantiate a parameterized rule *pr*, we have $LR_{[i]} = j$ (or $LR_{[i]} \neq j$) is equivalent to $LR_{[i]} = LI_{[j]}$ (or $LR_{[i]} \neq LI_{[j]}$), which is a factor to specify a case by comparing $LR_{[i]}$ with $LI_{[j]}$.

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 elements of it as a group of parameters to instantiate *pr* into a set *crs* of concrete rules.[4] For instance, for the invariant $\mathsf{mutualInv}(1, 2)$, 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 it and $\mathsf{mutualInv}(1, 2)$. Each of the three probed concrete causal relations will be used to generalized into a symbolic causal relation existing between crit and $\mathsf{mutualInv}$ in a case formulated by a predicate comparing rule parameters and invariant parameters.

[4] the details of computing $\mathsf{cmpSemiperm}(m+n, n)$ can be found in [12].

4.2 Core Searching Algorithm

For a *cinv* and a rule $r \in crs$, the core part of the 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 SMV, and calling SMV 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      $relItem \leftarrow (r, inv, invRule_2, -)$;
5      return $(\mathsf{NONE}, relItem)$;
6  else if $\mathsf{tautChk}(g \rightarrow inv') = true$ then
7      $relItem \leftarrow (r, inv, invRule_1, -)$;
8      return $(NONE, relItem)$;
9  else
10     $candidates \leftarrow \mathsf{subsets}(\mathsf{decompose}(\mathsf{dualNeg}(inv') \barwedge g))$;
11     $newInv \leftarrow choose(chk, candidates)$;
12     $relItem \leftarrow (r, inv, invRule_3, newInv)$;
13     if $isNew(newInv, invs)$ then
14         $newInv \leftarrow normalize(newInv)$;
15         return $(\mathsf{SOME}(newInv), relItem)$;
16     else
17         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 $subsets(decompose(dualNeg(inv')\bar{\wedge}g))$, 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 $f_1 \bar{\wedge} f_2 \bar{\wedge} ... \bar{\wedge} f_N$. $dualNeg(!f)$ 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 `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 $\mathsf{mutualInv}(2,1)$ in a symmetry view.

Table 1. 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	
..	..	..	..	..
crit	[1]	invOnXC(1)	invHoldRule1	_
crit	[2]	invOnXC(1)	invHoldRule1	_

For instance, let $PR = \{try, crit, exit, idle\}$, $invs = \{mutualInv(1,2)\}$, the output of the invFinder, which is stored in file `mutual.tbl`, is shown in Table 1. 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 $\mathsf{invHoldRule}_3$). The auxiliary invariants found by invFinder include: $\mathsf{inv}_2 \equiv !(\mathsf{x} \doteq true \bar{\wedge} n[1] = C)$, $\mathsf{inv}_3 \equiv !(n[1] = C \bar{\wedge} n[2] = E)$, $\mathsf{inv}_4 \equiv !(x \doteq \mathsf{true} \bar{\wedge} n[1] \doteq \mathsf{E})$, $\mathsf{inv}_5 \equiv !(n[1] \doteq \mathsf{C} \bar{\wedge} n[2] \doteq \mathsf{C})$. [5].

5 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

[5] The names mutualEx and invOnX1 in this work are just for easy-reading, their index here is generated in some order by invFinder

or symbolic value $\mathsf{symb}(str)$, where str is a string. We use $\mathtt{N}$ 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 $\mathtt{iInv_i}$ 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[$\mathtt{iInv_1}$]$\doteq$ C). This kind of generalization is done with model constraints, which specify that any parameter index should be not greater than the instance size $\mathtt{N}$, and parameters to instantiate a parameterized rule (formula) should be different. (2) The generalization of concrete causal relations into parameterized causal relations in Isabelle, and 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 $[\mathtt{iR_1} \doteq \mathtt{iInv_1}, \mathtt{iR_1} \doteq \mathtt{iInv_2}, (\mathtt{iR_1} \neq \mathtt{iInv_1}) \wedge (\mathtt{iR_1} \neq \mathtt{iInv_2})]$[6] , which stands for case-splittings by comparing a symbolic rule parameter iR_1 and invariant parameters $iInv_1$ and $iInv_2$. 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 1, there is a group of rule parameters LR, and a group of parameters LI occurring in an invariant formula.

Definition 5. *Let LR be a permutation s.t. $|LR| > 0$, which represents a list of actual parameters to instantiate a parameterized rule, let LI be a permutation $|LI| > 0$, which represents a list of actual parameters to instantiate a parameterized invariant, we define:*

1. *symbolic comparison condition generalized from comparing $LR_{[i]}$ and $LI_{[j]}$:*

$$\mathsf{symbCmp}(LR, LI, i, j) \equiv \begin{cases} \mathtt{iR_i} \doteq \mathtt{iInv_j} & \text{if } LR_{[i]} = LI_{[j]} \quad (1)\\ \mathtt{iR_i} \neq \mathtt{iInv_j} & \text{otherwise} \quad (2) \end{cases}$$

2. *symbolic comparison condition generalized from comparing $LR_{[i]}$ and with all $LI_{[j]}$:*

[6] $\mathtt{iR_1} \neq \mathtt{iInv_1}$ is the abbreviation of $!(\mathtt{iR_1} \doteq \mathtt{iInv_1})$

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

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

3. *symbolic case generalized from comparing* LR *with* LI *:* $\mathsf{symbCase}(LR, LI) \equiv \mathsf{forallForm}(|LR|, pf)$*, where* $pf(i) = \mathsf{symbCaseI}(LR, LI, i)$*;*
4. *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 first three lines in Table. 1, and $LI = [1, 2]$ is the list of parameters occurring in $\mathsf{mutualEx}(1, 2)$; and LR is the actual parameter list to instantiate crit.

- when $LR = [1]$, $\mathsf{symbCmp}(LR, LI, 1, 1) = (\mathtt{iR_1} \doteq \mathtt{iInv_1})$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 1) = (\mathtt{iR_1} \doteq \mathtt{iInv_1})$ becuase $LR_{[1]} = LI_{[1]}$.
- when $LR = [2]$, $\mathsf{symbCmp}(LR, LI, 1, 2) = (\mathtt{iR_1} \doteq \mathtt{iInv_2})$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 2) = (\mathtt{iR_1} \doteq \mathtt{iInv_2})$ becasue $LR_{[1]} = LI_{[2]}$.
- when $LR = [3]$, $\mathsf{symbCmp}(LR, LI, 1, 1) = (\mathtt{iR_1} \neq \mathtt{iInv_1})$, $\mathsf{symbCmp}(LR, LI, 1, 2) = (\mathtt{iR_1} \neq \mathtt{iInv_2})$, $\mathsf{symbCase}(LR, LI) = symbCaseI(LR, LI, 1) = (\mathtt{iR_1} \neq \mathtt{iInv_1}) \wedge (\mathtt{iR_1} \neq \mathtt{iInv_2})$ because neither $LR_{[1]} = LI_{[1]}$ nor $LR_{[1]} = LI_{[2]}$.
- let $LRS = [[1], [2], [3]]$, $\mathsf{partition}(LRS, LI) = (\mathtt{iR_1} \doteq \mathtt{iInv_1}) \vee (\mathtt{iR_1} \doteq \mathtt{iInv_2}) \vee ((\mathtt{iR_1} \neq \mathtt{iInv_1}) \wedge (\mathtt{iR_1} \neq \mathtt{iInv_2}))$

If we see a line in table 1 as a concrete test case for some concrete causal relation, then $\mathsf{symbCase}(LR, LI)$ is an abstraction predicate to generalize the concrete case. Namely, if we transform $\mathsf{symbCase}(LR, LI)$ by substituting $\mathtt{iInv_i}$ with $LI_{[i]}$, and $\mathtt{iR_j}$ with $LR_{[j]}$, the result is semantically equivalent to true.

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 1. 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.*

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

$\mathsf{lookup}(LI, LR, i)$ returns the symbolic index transformed from i according to whether i occurs in LI or in LR. The index i will be transformed into $\mathtt{iInv_{find\_first(LI,i)}}$

if i occurs in LI, and $\mathtt{iR}_{\mathtt{find\_first(LR,i)}}$ otherwise. Employing the lookup strategy to transform a concrete index i in inv' to lookup(LI, LR, i), symbolize2f transforms inv' into a symbolic one which will be needed in a proof command for existence of the $\mathsf{invHoldRule}_3$ relation in Isabelle.

6 Automatical Generation of Isabelle Proof

A formal model for a protocol case in a theorem prover like Isabelle includes the definitions of constants and rules and invariants, lemmas, and proofs. Readers can refer to [12] for detailed illustration of the formal proof script. In this section, we focus on the generation of a lemma on the existence of causal relation between a parameterize rule and invariant formula based on the aforementioned generalization of lines of concrete causal relations.

An example lemma $\mathtt{critVsinv}_1$ and its proof in Isabelle in the `mutualEx` protocol, is illustrated as follows:

```
1lemma critVsinv1:
2 assumes a1: ∃ iR1. iR1 ≤ N ∧ r=crit iR1 and
a2: ∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f=inv1 iInv1 iInv2
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 iInv1 iInv2 where
a2: iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f=inv1 iInv1 iInv2
  by blast
5 have iR1=iInv1 ∨ iR1=iInv2 ∨ (iR1 ≠ iInv1 ∧ iR1 ≠ iInv2) by auto
6 moreover{assume b1:iR1=iInv1
7   have invHoldRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x=! (x=true ⊼ n[iInv2]=C) in exI,auto)qed
8   then have invHoldRule s f r (invariants N) by auto}
9 moreover{assume b1:iR1=iInv2
10   have invHoldRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x=! (x=true ⊼ n[iInv1]=C in exI,auto)qed
11   then have invHoldRule s f r (invariants N) by auto}
12 moreover{assume b1:(iR1 ≠ iInv1 ∧ iR1 ≠ iInv2)
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 above proof, line 2 are assumptions on the parameters of the invariant and rule, which are composed of two parts: (1) assumption `a1` specifies that there exists an actual parameter `iR1` with which `r` is a rule obtained by instantiating `crit`; (2) assumption `a2` specifies that there exists actual parameters `iInv1` and `iInv2` with which `f` is a formula obtained by instantiating `inv1`. Line 4 are two typical proof patterns forward-style which fixes local variables such as `iR1` and new facts such as `a1: iR1 ≤ N ∧ r=crit iR1`. From line 5, the remaining part is a typically readable Isar proof using calculation reasoning such as `moreover` and `ultimately` to do case analysis. Line 5 splits cases of `iR1` into all possible cases by comparing `iR1` with `iInv1` and `iInv2`, which is in fact characterized by partition$([1], [2], [3]], [1, 2])$. Lines 6-14 proves these cases one by one: Lines 6-8 proves the case where `iR1=iInv1`, line 7 first proves that the causal relation $\mathsf{invHoldRule}_3$ holds by supplying a symbolic formula, which is transformed from

$invOnXC(2)$ by calling symbolize2f with lookUp strategy. From the conclusion at line 7, line 8 furthermore proves the causal relation invHoldRule holds; Lines 9-11 proves the case where `iR1=iInv2`, proof of which is similar to that of case 1; Lines 12-14 the case where neither `iR1=iInv1` nor `iR1=iInv2`. Each proof of a subcase is done in a block `moreover b1:asm1 proof1`, the `ultimately` proof command in line 15 concludes by summing up all the subcases.

Due to length limitation, we illustrate the algorithm for generating a key part of the proof of the lemma `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 the result of the generalization step, which is discussed in Section 5. In the body of function rel2proof, sprintf writes a formatted

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: $proof$
1  if $relTag = invHoldRule_1$ then
2      $proof \leftarrow$ sprintf
3        "have invHoldRule1 f r (invariants N)
4        by(cut_tac a1 a2 b1, simp, auto)
5        then have invHoldRule f r (invariants N) by blast" ;
6  else if $relTag = invHoldRule_2$ then
7      $proof \leftarrow$ sprintf
8        "have invHoldRule2 f r (invariants N)   by(cut_tac a1 a2 b1, simp, auto)
9        then have invHoldRule f r (invariants N) by blast" ;
10 else
11     $f' \leftarrow getFormField(relTag)$;
12     $proof \leftarrow$ sprintf
13       "have invHoldRule3 f r (invariants N)
14       proof(cut_tac a1 a2 b1, simp, rule_tac x=%s in exI,auto)qed
15       then have invHoldRule f r (invariants N) by blast" (symbf2Isabelle f')";
16 return $proof$
```

data to string and returns it. In line 10, 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, as shown in lines 7-8, 10-11, or 13-14. If the tag is among $\mathsf{invHoldRule}_{1-2}$, the transformation is rather straight-forward, else 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.

7 Experiments

We implement our tool in Ocaml. Experiments are done with typical bus-snoopy benchmarks such as MESI and MOESI, as well as directory-based benchmarks such as German and FLASH. The detailed codes and experiment data can be found in [12]. Each experiment data includes the paraVerifier instance, invariant sets, Isabelle proof scripts. Experiment results are summarized in Table 2.

Among all the work in the field of parameterized verification, only four of them have verified FLASH. The first full verification of safety properties of FLASH is

Table 2. Verification results on benchmarks.

Protocols	#rules	#invariants	time (seconds)	Memory (MB)
mutualEx	4	5	3.25	7.3
MESI	4	3	2.47	11.5
MOESI	5	3	2.49	23.2
Germanish [9]	6	3	2.9	7.8
German [6]	13	52	38.67	14
FLASH_nodata	60	152	280	26
FLASH_data	62	162	510	26

done in [10]. Park and Dill proved the safety properties of FLASH using PVS. The CMP method, which adopts parameter abstraction and guard strengthening, is applied in [6] for verifying safety properties of FLASH. McMillan applied compositional model checking [11] and used Candence SMV to the verification of both safety and liveness properties of FLASH. Sylvain et.al have applied Cubeic to the verification FLASH [9,13], which is theoretically based on an SMT model checking to the verification of array-based system. In the former three methods [10,6,11], auxiliary invariants are provided manually depending on verifier's deep insight in the FLASH protocol itself, while in Cubeic, auxiliary invariants are found automatically. In Cubeic, auxiliary invariants are searched backward by a heuristics-guided algorithm with the help of an oracle (a reference instance of the protocol), but these auxiliary invariants are in concrete form, and are not generalized to the parameterized form. Thus there is no parameterized proof derived for parameterized verification of FLASH.

The invariants-searching algorithm used in our work differs from that in Cubeic [9,13] in that the heuristics in our work are based on the construction of causal relation which is uniquely proposed in our work. Thus the auxiliary invariants in our work are different from those found in [9,13]. Moreover, we generalize these concrete invariants and causal relations into a parameterized proof, and generate a parameterized proof in Isabelle. The found invariants have abundant semantics reflecting the deep insight of the FLASH protocol design, and the readable Isabelle proof script formally proves these invariants. In this way, we prove the protocol with the highest assurance. 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.

8 Conclusion

The originality of paraVerifier lies in the following aspects: (1) instead of directly proving the invariants of a protocol by induction, we propose a general proof method based on the consistency lemma to decompose the proof goal into a number of small ones; (2) instead of proving the decomposed subgoals by hand, we automatically generate proofs for them based on the information of causal relation computed in a small protocol instance.

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.

References

1. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In Alur, R., Henzinger, T., eds.: Computer Aided Verification. Volume 1102 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1996) 184–195
2. Björner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Computer Science **173**(1) (1997) 49 – 87
3. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions? In: Proc. 13th International Conference on Computer Aided Verification. Volume 2102 of LNCS. Springer (2001) 221–234
4. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2031 of LNCS. Springer (2001) 82–97
5. Tiwari, A., Rueß, H., Saïdi, H., Shankar, N.: A technique for invariant generation. In: Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2031 of LNCS. Springer (2001) 113–127
6. Chou, C.T., Mannava, P., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Proc. 5th International Conference on Formal Methods in Computer-Aided Design. Volume 3312 of LNCS. Springer (2004) 382–398
7. Pandav, S., Slind, K., Gopalakrishnan, G.: Counterexample guided invariant discovery for parameterized cache coherence verification. In: Proc. 13th IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Volume 3725 of LNCS. Springer (2005) 317–331
8. Lv, Y., Lin, H., Pan, H.: Computing invariants for parameter abstraction. In: Proc. the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, IEEE CS (2007) 29–38
9. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel smt-based model checker for parameterized systems. In Madhusudan, P., Seshia, S., eds.: Proc. 24th International Conference on Computer Aided Verification. Volume 7358 of LNCS. Springer (2012) 718–724
10. Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, Padua, Italy, ACM (1996) 288–296
11. Mcmillan, K.L., Labs, C.B.: Parameterized verification of the flash cache coherence protocol by compositional model checking. In: In CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144, Springer (2001) 179–195
12. Y. Li, K.d.: paraverifier (2016) `https://github.com/paraVerifier/paraVerifier`.
13. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: FMCAD, Portland, Oregon, USA (October 2013)