A Novel Approach to Parameterized verification of Cache Coherence Protocols

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 can be also automatically generalized into a parameterized form to construct a formally parameterized 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. Another novel feature of our method lies in that the last verification result of a protocol is provided by a formally 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 protocols, security systems, 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, not just for a single protocol instance. The model checking tools, although powerful in verification of non-parameterized systems, becomes 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 research papers in the field of parameterized verification [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. Among them, the 'invisible invariants' method, proposed in [**?**], 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, which adopts parameter abstraction and guard strengthening, is proposed in [**?**] for verifying 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 is the most important of all to estimate an approach in the field of parameterized verification. For instance, FLASH is a hard real-world benchmark for all existing methods for parameterized verification. Human guidance plays a key role in the existing successful verifications for FLASH [**?**,**?**,**?**], and this is still the case up to now. In order to effectively verify complex parameterized protocols like FLASH, there are two critical problems which need to be handled. The first one is how to find a set of sufficient and necessary invariants, which is a core problem in the field of parameterized verification. It is desirable to get the invariants with less human intervention. Previously, the theory foundation of a technique of parameterized verification and its soundness are only discussed in a paper proof style. It is preferable to formulate all the verification in a publicly-recognized trust-worthy framework like a theorem prover [**?**]. However, theorem proving in a theorem prover like Isabelle is usually interactive, not automatical.

In order to solve the parameterized verification of cache coherence protocols 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 in 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 type and constant definitions to model protocol cases and lemmas to prove invariant 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 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 a parameterized form, 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.

The organization of this work is as follows: Section 2 introduces the theoretical foundation; **??** the invFinder; **??** the generalization strategy; **??** the proofGen and the generated proof. We go through these sections by verifying a small example - mutual exclusion example. Section **??** shows the further experiments on real-world protocols. Section **??** compares ours with the previous work.

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 enumeration, natural number, and Boolean.

Experssions and $formulas$ are defined mutually recursively. *Experssions* 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 boolean 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 from formulas using the logic connectives, including negation (!), conjunction ($\overline{\wedge}$), 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 the state $s :: var \Rightarrow valueType$ as $\mathbb{A}[e, s]$ (or $\mathbb{B}[f, s]$). For a 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 (higer-order logics) as usual, which is shown in the appendix.[1]

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 $\mathsf{guard}(g \rhd \alpha) \equiv g$. If the guard g is satisfied

[1] The logic to specify parameterized system is a special logic, which can be embedded in HOL supported by Isabelle. Therefore, HOL can be seen as the main meta-logic to specify our work.

at 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: $s \xrightarrow{r} s' \equiv s \models \mathsf{guard}(r) \wedge s \overset{\mathsf{act}(r)}{\twoheadrightarrow} s'$.

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 exists 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 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, $pini(N)$ the predicate to specify the inial state, prules(N) the four rules of the protocol, $mutualInv(i, j)$ a property that $n[i]$ and $n[j]$ cannot be C at the same time. We want to verify that $mutualInv(i, j)$ 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)
```

As Hoare logics specifies, after executing statement α, f holds iff $\mathsf{preCond}(f, \alpha)$ holds before the execution.

Lemma 1. *Suppose* $s \overset{\alpha}{\twoheadrightarrow} s'$, $s \models \mathsf{preCond}(f, \alpha)$ *if and only if* $s' \models f$

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. $\mathsf{invHoldRule}_1(s, f, r) \equiv s \models \mathsf{pre}(r) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;[2]
2. $\mathsf{invHoldRule}_2(s, f, r) \equiv s \models f \longleftrightarrow s \models \mathsf{preCond}(f, (\mathsf{act}(r))$;
3. $\mathsf{invHoldRule}_3(s, f, r, fs) \equiv \exists f' \in fs \; s.t. \; s \models (f' \overline{\wedge} (\mathsf{pre}(r)) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;
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 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 exists another invariant $f' \in fs$ such that the conjunction of the guard of r and f' implies the precondition $\mathsf{preCond}(S, f)$.

The second way to view $\mathsf{invHoldRule}(s, f, r, fs)$ is to regard it 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 kind of inductive tactics can be done by a theorem prover, which is the cornerstone of our work.

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. *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]≐ C)
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 $i_1 \neq N$, $i_2 \neq N$, and $iR_1 \leq N$.

- $\mathsf{invHoldForRule}_2(s, \mathsf{mutual}(i_1, i_2), \mathsf{crit}(iR_1), (\mathsf{pinvs}(N))$, where $i_1 \neq i_2$, $i_1 \neq iR_1$, and $i_2 \neq iR_1$, since $\mathsf{preCond}(\mathsf{act}(\mathsf{crit}(iR_1)), \mathsf{mutual}(i_1, i_2)) = \mathsf{mutual}(i_1, i_2)$.
- $\mathsf{invHoldForRule}_3(s, \mathsf{mutual}(i_1, i_2), \mathsf{crit}(iR_1), (\mathsf{pinvs}(N))$, where $i_1 \neq i_2$, and $i_1 = iR_1$. Since $\mathsf{invOnXC}(i_2) \in \mathsf{pinvs}(N)$, $\mathsf{preCond}(\mathsf{act}(\mathsf{crit}(iR_1)), \mathsf{mutual}(i_1, i_2)) = !(\mathsf{C} \doteq \mathsf{C} \overline{\wedge} n[i_2] \doteq \mathsf{C})$.
- $\mathsf{invHoldForRule}_3(s, \mathsf{mutual}(i_1, i_2), \mathsf{crit}(iR_1), (\mathsf{pinvs}(N))$, where $i_1 \neq i_2$, and $i_2 = iR_1$. Since $\mathsf{invOnXC}(i_1) \in \mathsf{pinvs}(N)$ implies $\mathsf{preCond}(\mathsf{act}(\mathsf{crit}(iR_1))\mathsf{mutual}(i_1, i_2)) = !(\mathsf{C} \doteq \mathsf{C} \overline{\wedge} n[i_1] \doteq \mathsf{C})$.

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

Suppose that the consistency relation consistent$(invs, inis, rs)$ holds, for any $inv \in invs$, inv holds for any reachable state s such that $s \in$ reachableSet(ini, rs). The following lemma formalizes the essence of the aforementioned causal relation, and is called consistency lemma.

Lemma 2. *If* $P = (ini, rs)$, consistent$(invs, ini, rs)$, *and* $s \in$ reachableSet(P), *then for all* inv *s.t.* $inv \in invs$, $s \models inv$.

Now we apply the consistence lemma to prove that the mutual exclusion property holds for each reachable state of the mutual-exclusion protocol. Let us recall example 1 and 2.

Lemma 3. *If* $P = (pini(N), prules(N))$, $s \in$ reachableSet(P), *and* $0 < N$, *then for all* inv *s.t.* $inv \in pinvs(N)$, $s \models inv$.

Consistency lemma has eliminated the need of directly use of usual induction proof method. We only check the causal relation between f and r by case analysing parameters of a rule and an invariant, and such a causal relation checking is within the ability of an automatical tactic provided by a theorem prover like Isabelle. Due to the uniform style of checking causal relation, it is also feasible to generate a proof of such a style by an external proof generator. However, as shown in Example 2, the three items shows that there are three subcases where we compare rule parameter iR_1 with formula parameters i_1 and i_2, different kinds of causal relations hold between rule crit iR_1 and invariant mutualInv i_1 i_2. Only if a proof script contains enough information on the case splitting and which kind of causal relation to be checked in each case, Isabelle can help us to automatically check whether the kind of causal relation hold in the case. How to provide these key proof information is a central problem in work of both invFinder and proofGen.

4 invFinder

invFinder works iteratively in a semi-proving and semi-searching way. A queue $newInvs$ is used to store new invariants found up to now, which initially only contains invariant formulas we want to verify. In each iteration, the head element $cinv$ of $newInvs$ is popped, and used to compute the causal relation between $cinv$ and a parameterized rule pr. Then invFinder instantiates the parameterized rule into a set of concrete rules crs with different groups of concrete parameters according to a parameter instantiation policy. lyj: Each rule in crs represents a pattern w.r.t. the relationship (equivalent or not) between concrete rule parameters and invariant parameters. Such a parameter instantiation to the rule pr can be generalized to a symbolic predicate which is according with a case as shown Example 2, therefore, a good parameter instantiation policy should guarantee the completeness. Namely, each of subcases in the needed case-splitting should be covered by a concrete rule which is instantiated from pr according to the policy. For a rule $cr \in crs$, invFinder tries to prove that some causal relation exists between the rule and an invariant; if there is no such an invariant in the current

invariant set, invFinder automatically generates a new auxiliary invariant and records the corresponding causal relation information between the current rule and invariant. Whenever a new invariant is generated, it is pushed into $newInvs$. invFinder performs the above searching process until $newInvs$ is empty.

4.1 Parameter instantiation policy

In order to formulate our parameter instantiation 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 a n-permutation of m is ordered arrangement of a n–element subset of an m-element set $I = \{i.0 < i \leq m\}$. We use a list with size n to stand for a n-permutation of m, whose elements are mutually different from each other and taken from I. For instance, $[1, 2]$ is a 2-permutation of 3. If $xs_{[i]} = i$ for all $i \leq |xs|$, we call it identical permutation, and sometimes is denoted by 1 upto n if $n = |xs|$.

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 $\mathsf{semiP}(m, n, S)$.*

According to the definition of of the relation $\simeq^n_m$, $L \simeq^n_{m+n} L'$ means that if we compare any element $L_{[i]}$ with any $j \leq m$, the comparing result should be the same as that obtained by comparing $L'_{[i]}$ with j, namely $L_{[i]} = j$ if and only if $L'_{[i]} = j$. For instance, let $L = [2, 3]$ and $L' = [2, 4]$, then $L \simeq^2_4 L'$, for any $0 < j \leq 2$, we have $L_{[i]} = j$ iff $L'_{[i]} = j$. This observation is formally captured by the following lemma.

Lemma 4. *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 $L'_{[i]} = j$.*

Lemma 4 shows that, we can analyze a group of concrete parameters by analyzing only one of them as a presentative. Keeping this in mind, let us look at the following lemma, which together with Lemma 4 is the theoretical basis of our policy.

Lemma 5. *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 indice $i \leq m$ and $j \leq n$ such that $L_{[i]} = j$ and $L'_{[i]} \neq j$.*

Lemma 5 shows that, 1) completeness of S w.r.t. the set perms_m^n under the relation $\simeq$, 2) the distinct 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 $j < n - m$, and the case pattern represented by each element in S is also different from that represented by the other elelments. lyj: my correction (**this sentence should be finished...**). 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 $LR_{[1]} \neq 2$.

lyj: my correction Note that each one in the above cases are disjoint with the other one, and the disjuction of all the cases is true.

Consider natural numbers m and n, computing a quotient of perms_m^n can be implemented by an algorithm cmpSemiperm. Roughly speaking, cmpSemiperm firstly computes the set perms_m^n, and push elements of it into a queue S_0, and select elements of S_0 into a set S. Initially S is set empty. the head element of S_0 is pop into L, then L will be checked whether there is an element L' in S s.t. $L \simeq_m^n L'$. If yes, then L will be discarded, else L is inserted into S. This procedure is repeated until S is empty. Finnaly S will be returned. The detail of cmpSemiperm can be found in [].

Definition 4. *A concrete invariant formula cinv is normalized if one of the following conditions hold:*

- *there exists a parameterized invariant pinv and an identical permutation LI with $|LI| > 0$ such that $cinv = pinv(1, ... |LI|)$;*
- *no array variable occurs in cinv;*

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 $LR_{[i]} \neq LI_{[j]}$) lyj: my correction[**what is LR, you only use it in an example, need to define it clearly in this paragraph**] , which is a factor to specify a case by comparing $LR_{[i]}$ with $LI_{[j]}$. All the auxiliary invariant formulas will be in a normalized form, and checked with any parameterized rule whether some kind of causal relation hold between them.

Let *cinv* be a normalized concrete invariant, 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 the set $\mathsf{cmpSemiperm}(m + n, n)$ lyj: see the added para before definition 4[**what is this cmpSemiperm, never defined**], and use elements of it as a group of

parameters to instantiate pr into a set crs of concrete rules. For instance, for the invariant $\mathsf{mutualInv}(1,2)$, three groups of parameters [1], [2], [3] will be used to instantiate crit respectively, each of which will be used to check which kind of causal relation exists between it and $\mathsf{mutualInv}(1,2)$.

4.2 Core Searching Algorithm

Jump from policy to core algorithm suddenly... need to introduce the relation between policy and the below algorithm. For a $cinv$ and a rule $r \in crs$, the core part of the $\mathsf{invFinder}$ tool is shown in Algorithm 1. 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 firstly translating the formula into a formula in SMV, and then 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 (abbreviation for SAT Modulo Theories) format, and then calls an SMT solver such as Z3 to check it.

There are parameters in the algorithm 1, including 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 returned result is a pair of formula option, and causal relation item between r and inv. A formula option value $NONE$ denotes that no new invariant is found; and $SOME(f)$ that a new auxiliary invariant f is searched.

Now let us explain the body of $coreFinder$. After computing the pre-condition inv' in line 2, which is the weakest precondition of the input formula inv w.r.t. S, and takes further operations according to the cases it faces with:

(1) if $inv = inv'$, which means that statement S does not change inv, then no new invariant is created, and new causal relation item marked with tag $\mathtt{invHoldForRule}_2$ is recorded between r and inv, but at this moment there are no new invariants to be added; for instance, let $r = \mathsf{crit}(3)$, $inv = \mathsf{mutualInv}(1,2)$, thus $inv' = \mathsf{preCond}(S, inv) = inv$, then a pair $(\mathsf{NONE}, (crit(3), inv, \mathsf{invHoldForRule}_2, \_))$ will be returned, where $NONE$ means no new invariant formula is returned.

(2) Secondly, if $\mathsf{tautChk}$ verifies that $g \dashrightarrow inv'$ is a tautology, then no new invariant is created, and the new causal relation item marked with tag $\mathsf{invHoldForRule}_1$ is recorded between r and inv. For instance, let $r = \mathsf{crit}(2)$, $inv = \mathsf{invOnXC}(1)$, $inv' = \mathsf{preCond}(S, inv) = \neg(\mathsf{false} \doteq \mathsf{true} \,\overline{\wedge}\, n[1] \doteq \mathsf{C})$, obviously, $g \dashrightarrow inv'$ holds forever because inv' is always evaluated true, thus a pair $(\mathsf{NONE}, (\mathsf{crit}(2), inv, \mathsf{invHoldForRule}_1, \_))$ will be returned.

(3) Thirdly, if neither of the above two cases holds, then a new auxiliary invariant $newInv$ will be constructed, which will make the causal relation $\mathsf{invHoldForRule}_3$ to hold. The candidate set is $subsets(decompose(dualNeg(inv')\overline{\wedge}$

Algorithm 1: Core Searching Algorithm: *coreFinder*

Input: r, inv, $invs$
Output: A formula option f, a new causal relation rel

```
1  g ←the guard of r, S ←the statement of r;
2  inv' ← preCond(inv, S);
3  if inv = inv' then
4      relItem ← (r, inv, invRule_2, −);
5      return (NONE, relItem);
6  else if tautChk(g → inv') = true then
7      relItem ← (r, inv, invRule_1, −);
8      return (NONE, relItem);
9  else
10     candidates ← subsets(decompose(dualNeg(inv') ⩓ g));
11     newInv ← choose(chk, candidates);
12     relItem ← (r, inv, invRule_3, newInv);
13     if isNew(newInv, invs) then
14         newInv ← normalize(newInv);
15         return (SOME(newInv), relItem);
16     else
17         return (NONE, relItem);
```

$g)$), 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. $subsets(S)$ to denote 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. Here, the meaning of the word "new" is modulo to the symmetry relation. For instance, $\mathsf{mutualInv}(1,2)$ is equivalent to $\mathsf{mutualInv}(2,1)$ in a symmetry view. Let $invs = \emptyset$, $r = \mathsf{crit}(1)$, $inv = \mathsf{mutualInv}(1,2)$, $inv' = \mathsf{preCond}(S, inv) = !(true \doteq true \barwedge n[2] \doteq C)$, from all the subsets of $\{n[1] \doteq T, x \doteq true, n[2] \doteq C\}$, the `choose` oracle selects the subset $\{x = true, n[2] \doteq C\}$ combines all the item in this candidate, then constructs a new invariant $inv_0 = !(x \doteq true \barwedge n[2] \doteq C)$. After normalization, the new invariant $!(x = true \barwedge n[1] \doteq C)$ and a relation item $(crit(1), \mathsf{invHoldForRule}_3, inv_0)$ will be returned.

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 17. In the table, each line records the index of a normalized invariant, name of a

Table 1. A fragment of output of invFinder

rule	ruleParas	inv	causal relation	f'
..	..	..	..	..
crit	[1]	mutualInv(1,2)	invHoldForRule3	invOnXC(2)
crit	[2]	mutualInv(1,2)	invHoldForRule3	invOnXC(1)
crit	[3]	mutualInv(1,2)	invHoldForRule2	
..	..	..	..	..
crit	[1]	invOnXC(1)	invHoldForRule1	_
crit	[2]	invOnXC(1)	invHoldForRule1	_

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 $\mathsf{invHoldForRule}_3$). The auxiliary invariants found by invFinder includes: $\mathsf{inv}_2 \equiv !(\mathsf{x} \doteq true \barwedge n[1] = C)$, $\mathsf{inv}_3 \equiv !(n[1] = C \barwedge n[2] = E)$, $\mathsf{inv}_4 \equiv !(x \doteq \mathsf{true} \barwedge n[1] \doteq \mathsf{E})$, $\mathsf{inv}_5 \equiv !(n[1] \doteq \mathsf{C} \barwedge n[2] \doteq \mathsf{C})$. [3].

5 Generalization

Intuitively, generalization means that a concrete index (formula or rule) is generalized into a set of concrete indice (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 an parameterized protocol instance. A concrete index i can be transformed into a symbolic one by some special strategy g. Namely $symbolize(g, i) = symb(g(i))$. In this work, two special transforming function $fInv(i) = "iInv"\hat{}itoa(i)$ and $fIr(i) = "iR"\hat{}itoa(i)$, where $itoa(i)$ is the standard function transforming an integer i into a string. We use special symbols $\mathtt{iInv}_\mathtt{i}$ to denote $symbolize(fInv, i)$; and $\mathtt{iR}_\mathtt{i}$ to denote $symbolize(fIr, i)$. The former formalizes a symbolic parameter of a parameterized formula, and the latter a symbolic parameter of a parameterized rule. Accordingly, we define $symbolize2f(g, inv)$ (or $symbolize2r(f, r)$), 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 $symbolize(g, i)$.

There are two main kinds generalization in our work: (1) generalization on model constraints, which is specifying that any parameter index should be not greater than the instance size `N`, and different parameters to instantiate a parameterized rule (formula) should be different. They are rather straightforward, and are put in the appendix; (2) generalization for generation of case-splittings in the proofs, e.g., $iR_1 = iInv_1$ or $iR_1 = iInv_2$ stand for case splitting by comparing a symbolic rule parameter iR_1 and invariant parameters $iInv_1$ and $iInv_2$.

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

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

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

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

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

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

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

$symbCmp(LR, LI, i, j)$ defines a symbolic formula generalized from comparing $LR_{[i]}$ and $LI_{[j]}$; $symbCaseI(LR, LI, i)$ a symbolic formula summarizing the results of comparison between $LR_{[i]}$ and all $LI_{[j]}$ such that $j \leq |LI|$; $symbCase(LR, LI)$ a symbolic formula representing a subcase generalized from comparing all $LR_{[i]}$ and all $LI_{[j]}$; $partition(LRS, LI)$ is a disjunction of subcases $symbCase(LRS_{[i]}, LI)$. Recall the first three lines in Table. 17, and $LI = [1, 2]$ is the list of parameters occurring in $mutualEx(1, 2)$

- $LR = [1]$ is the actual parameter list to instantiate *crit*, $symbCmp(LR, LI, 1, 1) = (\mathtt{iR_1} = \mathtt{iInv_1})$, $symbCase(LR, LI, 1) = (\mathtt{iR_1} = \mathtt{iInv_1})$.
- $LR = [2]$ is the actual parameter list to instantiate *crit*, $symbCmp(LR, LI, 1, 1) = (\mathtt{iR_1} \neq \mathtt{iInv_1})$, $symbCase(LR, LI) = (\mathtt{iR_1} = \mathtt{iInv_2})$ becasue $LR_{[1]} = LI_{[2]}$.
- $LR = [3]$ is the actual parameter list to instantiate *crit*, $symbCmp(LR, LI, 1, 1) = \neg(\mathtt{iR_1} = \mathtt{iInv_1})$, $symbCase(LR, LI) = symbCaseI(LR, LI, 1) = (\mathtt{iR_1} \neq \mathtt{iInv_1}) \wedge (\mathtt{iR_1} \neq \mathtt{iInv_2})$ because neither $LR_{[1]} = LI_{[1]}$ nor $LR_{[1]} = LI_{[2]}$.
- let $LRS = [[1], [2], [3]]$, $partition(LRS, LI) = (\mathtt{iR_1} = \mathtt{iInv_1}) \vee (\mathtt{iR_1} = \mathtt{iInv_2}) \vee ((\mathtt{iR_1} \neq \mathtt{iInv_1}) \wedge (\mathtt{iR_1} \neq \mathtt{iInv_2}))$

Note that $partition(LRS, LI)$ is a complete partition if it is a tautology. In the above example, the two partitions are both complete. If we see a line as a concrete test case for some concrete causal relation in table 17, then $symbCase(LR, LI)$ is an abstraction predicate to generalize the concrete case. Namely, if we transform $symbCase(LR, LI)$ by substitute $\mathtt{iInv_i}$ with $LI_{[i]}$, and $\mathtt{iR_j}$ with $LR_{[j]}$, the result is semantically equivalent to true.

The last thing is how to generalize the formula f' in a line of table 17 where the causal relation $\mathsf{invHoldForRule}_3$ holds. An index occurring in f' can occur in the invariant formula, or in the rule. We need look up the parameters occurring in the formula or in the rule to determine the transformation.

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

$lookup(LI, LR, i)$ returns the symbolic index transformed from i according to whether i occurs in LI or in LR, which defines the generalization strategy into the concrete index i. For instance, let $LI = [1, 2]$, $LR = [2]$, then $lookup(LI, LR, 2) = \mathtt{iInv}_2$, and $invOnXC(2)$ is transformed into $\neg(x \doteq true \wedge n[\mathtt{iInv}_2] \doteq C)$ by the strategy *lookup*.

For convenience in generating Isabelle proofs, a table $symbCausalTab$ is generated, which stores causal relation between a parameterized rule and an invariant w.r.t. a size N. An entry of the table is referenced by concatation of the name of a rule and an invariant formula. Such an entry is a record containing fields `symbCases` and `relationItems`, and created by collecting all the lines on the concrete relation items between the rule and the invariant. `symbCases` stores the list of symbolic cases each of which is derived by $\mathtt{symbCase}(LR, 1\ \mathtt{upto}\ n)$, where LR is the rule parameters, and n the number of parameters occuring in the invariant. relations store a list of generalized causal relation items. For instance, for the rule *crit* and *inv*5, a record $(|\mathtt{symbCases} = [\mathtt{iR}_1 = \mathtt{iInv}_1, \mathtt{iR}_1 = \mathtt{iInv}_2, (\mathtt{iR}_1 \neq \mathtt{iInv}_1) \wedge (\mathtt{iR}_1 \neq \mathtt{iInv}_2)]; \mathtt{relationItems} = [invHoldForRule_3(f_1'), invHoldForRule_3(f_2'), invHoldForRule_2]|)$, where $f_1' = symbolize2f(lookup([1, 2], [1]), invONXC(2))$, and $f_2' = symbolize2f(lookup([1, 2], [2]), invONXC(1))$.

6 Automatical generation of Isabelle proof by `proofGen`

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. An overview of the hierarchy of the formal proof scripts of is shown in Fig **??**.

In detail, the proof script is divided into parts as follows: (1) Definitions of protocol under case study including enumerating datatypes; definitions of formally parameterized invariant formulas, and the set of all actual invariants w.r.t. a protocol instance; definitions of formally parameterized rules, and the set of all actual rules w.r.t. a protocol instance; definitions of specification of the initial state; (2) A lemma such as $\mathtt{rule}_i\mathtt{\_Vs\_inv}_j$ on a causal relation of a rule and an invariant; (3) A Lemma such as $\mathtt{rules\_inv}_i$ on causal relations for all rules in the rule set and an invariant. (4) A lemma `rules_invs` on a causal relation for all rules in the rule set and all invariants in the invariant set. (4) A Lemma such as $\mathtt{iniImply\_inv}_i$ on a fact that an invariant $\mathtt{inv}_i$ hold at the initial state defined by the specification of the initial state. (5) A lemma `on_inits` proves that for all invariants they hold at the initial state of the protocol. (6) Main

theorem proving that any invariant formula holds at any reachable state of the N-parameterized FLASH protocol instance.

A lemma at the bottom level, specifies that causal relation hold between a rule like `crit` and a parameterized rule like inv_1, An example lemma $\texttt{critVsinv}_1$ 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 invHoldForRule 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 invHoldForRule3 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 invHoldForRule s f r (invariants N) by auto}
9 moreover{assume b1:iR1=iInv2
10   have invHoldForRule3 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 invHoldForRule s f r (invariants N) by auto}
12 moreover{assume b1:(iR1 ≠ iInv1 ∧ iR1 ≠ iInv2)
13   have invHoldForRule2 s f r
    proof(cut_tac a1 a2 b1, auto) qed
14   then have invHoldForRule s f r (invariants N) by auto}
15ultimately show invHoldForRule 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 parts of the proof is a typically readable one in Isar style [**?**], which uses 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 $invHoldForRule_3$ holds by supplying a formula, which is $symbolize'(invOnXC(2),[1,2],[1])$. From the conclusion at line 7, line 8 furthermore proves the causal relation $invHoldForRule$ hold; 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.

Algorithms of Proof Generator proofGen A lemma such as `critVsinv1` is generated by collecting all the records on the invariant `inv1` and rule `crit` in the aforementioned tables. Due to length limitation, we illustrate the algorithm to

generate a key part of the proof of the lemma: the generation of a subproof according to a symbolic relation tag of $invHoldForRule_{1-3}$, which is shown in Algorithm 2.

Algorithm 2: Generating a kind of proof which is according with a relation tag of $invHoldForRule_{1-3}$: rel2proof

Input: A causal relation item $relTag$
Output: An Isablle proof: $proof$

```
1  if relTag = invHoldForRule1 then
2      proof ← sprintf
3        "have invHoldForRule1 f r (invariants N)
4        by(cut_tac a1 a2 b1, simp, auto)
5        then have invHoldForRule f r (invariants N) by blast" ;
6  else if  relTag = invHoldForRule2 then
7      proof ← sprintf
8        "have invHoldForRule2 f r (invariants N)   by(cut_tac a1 a2 b1, simp, auto)
9        then have invHoldForRule f r (invariants N) by blast" ;
10 else
11     f' ← getFormField(relTag);
12     proof ← sprintf
13       "have invHoldForRule3 f r (invariants N)
14       proof(cut_tac a1 a2 b1, simp, rule_tac x=%s in exI,auto)qed
15       then have invHoldForRule f r (invariants N) by blast" (symbf2Isabelle f')";
16 return proof
```

In the body of function rel2proof, $sprintf$ writes a formatted data to string and returns it. In line 10, $getFormField(relTag)$ returns the field of formula f' if $relTag = invHoldForRule_3(f')$. rel2proof transforms a symbolic relation tag into a paragraph of proof, as shown in lines 7-8, 10-11, or 13-14. Such a relation tag is stored as an element in `relationTags` field of an entry of $symbCausalTable$ on $crit$ and $inv1$. If the tag is among $invHoldForRule_{1-2}$, 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 should be used to construct the $invHoldForRule_3$ relation.

Roughly speaking, the generation of a lemma such as `critVsinv1` needs the information stored in the aforementioned three tables in the end of section **??**. Lines 1-4 can be generated by looking up information on `inv1` in the table `symbInvs` and that on `crit` in `symbRules`. Line 5 can be generated by combining the disjunction of elements of the `symbCases` field of the entry of`symbCausalTable` on `crit` and `inv1`, which is a command of case splitting. Lines 6, 9, 12 are generated by the elements of the `symbCases` field of the entry respectively, which are assumptions of subcases. Subproofs of subcases can be

generated by Algorithm 2 according to three elements in `relationTags` field of the entry. Lines 15-16 are string with fixed pattern, which can be output directly.

7 Experiments and related work

We implement our tool in Ocaml [**?**]. More experiments are done including typical bus-snoopy ones such as MESI and MOESI, directory-based ones such as Germanish, and German protocols. The detail experiment codes and data can be found in JP: [**?**]. Each experiment data includes the paraVerifier instance, invariant sets found, Isabelle proof scripts, and the simulation flow graph of one single node. A table summarizes our experiments below. Among the benchmarks, a case study is done We have successfully verified the important properties of the FLASH protocol, resulting into an independent and different proof from the literature. As Chou, Mannava, Park pointed out in [**?**], FLASH is a good benchmark for any proposed method for parameterized verification: if the method works on FLASH, then there is a good chance that it will also work on many real-world cache coherence protocols. Therefore, our approach has reached this most important landmark.

There have been a lot of research papers in the field of parameterized verification [**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**,**?**]. Among them, only four of them have verified FLASH. The first full verification of safety properties of FLASH is done by work in [**?**]. Park and Dill proved the safety properties of FLASH using PVS [**?**]. The CMP method, which adopts parameter abstraction and guard strengthening, is proposed in [**?**] for verifying a safety property of a parameterized system. An abstract instance of the parameterized protocol is constructed by a counter-example-guided refinement process. McMillan applied compositional model checking [**?**] and used Candence SMV [**?**] to the verification of both safety and liveness properties of FLASH. However, auxiliary invariants are needed in all the above three approaches, and provided manually depending on verifier's deep insight in the FLASH protocol itself. In contrast to these work, auxiliary invariants are found automatically by invFinder. Auxiliary invariants are searched automatically 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 not generalized to the paramterized form. Thus there is no a parameterized proof derived for parameterized verification of FLASH. The heuristics-guided algorithm used in our work is different from that in [**?**] because the heuristics is basing on the construction of causal relation which is uniquely proposed in our work. Thus the auxiliary invariants are different from those found in [**?**]. We also generalize these concrete invariants into a parameterized proof according to a strategy, and generate a parameterized proof in Isabelle, which is readable for human.

8 Conclusion

Within paraVerifier, our automatic framework for parameterized verification of cache coherence protocol, (1) instead of directly proving the invariants of a pro-

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 [**?**]	6	3	2.9	7.8
German [**?**]	13	52	38.67	14
FLASH_nodata	60	152	280	26
FLASH_data	62	162	510	26

tocol 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 computed in a small protocol instance.[4]

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.

[4] Technical details of paraVerifier will be made available in a technical report.