An Automatic Proving Approach to Parameterized Verification

Yongjian Li, Kaiqiang Duan, Jun Pang, Yi Lv, and Shaowei Cai

Abstract—Formal verification of parameterized protocols such as cache coherence protocols is an outstanding challenge. In this paper, we propose an automatic proving approach and its prototype paraVerifier to handle this challenge within a unified framework: (1) to prove the correctness of a parameterized protocol, our approach automatically discovers auxiliary invariants and the corresponding causal relations among the discovered invariants and protocol rules from a small instance of the verified protocol; (2) the discovered invariants and causal relation information are then automatically generalized into a parameterized form to construct a parameterized formal proof in a theorem prover (i.e., Isabelle). Our approach has been successfully applied to a number of benchmarks including snoopy-based and directory-based cache coherence protocols. As side effect, the final verification result of a protocol is provided by a formal and human-readable proof.

Index Terms—Automatic verification, theorem proving, inductive methods, invariant and proof generation, cache coherence protocols.

1 Introduction

Formal verification of parameterized protocols is of great interest in the area of formal verification, mainly due to the practical importance of such systems. Parameterized protocols exist in many important application areas, including cache coherence, security, and network communication protocols. The significant challenge in parameterized verification is mainly due to the correctness requirement that the desired properties should hold in any instance of the parameterized protocol. Model checking [1], [2], as an automatic verification technique, is powerful and efficient in verification of non-parameterized protocols, but it becomes impractical when verifying parameterized protocols as it can only check an instance of the parameterized protocol in each verification. A more desirable approach is to provide a proof that the correctness holds for any instance of the verified systems.

Related works. There have been a lot of studies in the field of parameterized verification [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. We discuss in the following the most relevant papers to our work.

Among them, the 'invisible invariants' method [6], [7] is an automatic technique for parameterized verification. In this method, auxiliary invariants are computed in a finite system instance to aid inductive invariant checking. Combining parameter abstraction and guard strengthening with the idea of computing invariants in a finite instance, Lv et al. [12] use a small instance of a parameterized protocol as a 'reference instance' to compute candidate invariants. References to a specific node in these candidate invariants are then abstracted away, and the resulting formulas are used to strengthen guards of the transition rules in the abstract node. Both of the two approaches [6], [12] attempt to automatically find invariants. However, the invisible invariants are 'raw' boolean formulas transferred from the reachable sate set of a small finite instance of a protocol, which are BDDs computed by TLV (a variant of the BDD-based SMV model checker). These formulas do not carry intuitions understandable to human. The capacity of the invisible invariant method is seriously limited when computing the reachable set of invisible invariants, thus it is not feasible in the case of large examples such as FLASH [16]. To the best of our knowledge, the examples that can be handled by the invisible invariant method are quite small.

The method of Chou et al. [10] adopts parameter abstraction and guard strengthening for verifying an invariant of a parameterized protocol. An abstract instance of the parameterized protocol, which consists of $m + 1$ nodes $\{P_1, \ldots, P_m, P^*\}$ (m normal nodes and one abstract node P^*), is constructed iteratively. The abstract system is an abstraction for any protocol instance whose size is greater than m. Normally the initial abstract system does not satisfy the invariant, nevertheless it is still submitted to a model checker for verification. When a counterexample is produced, one needs to carefully analyze it and comes up with an auxiliary invariant, then uses it to strengthen the guards of some transition rules of the abstract node. The strengthened system is then submitted for model checking again. This process stops until the refined abstract system eventually satisfies the original invariant as well as all the auxiliary invariants supplied by the user. However, this method's soundness is only argued in an informal way. To the best of our knowledge, its correctness has not been formally proved in a theorem prover. This situation may not be ideal because its application domain for cache coherence protocols which demands the highest assurance for correctness. Besides, the analysis of counterexample and generation of new auxiliary invariants usually depend on

• *Yongjian Li, Kaiqiang Duan, Yi Lv, and Shaowei Cai and Yi Lv are with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of sciences.*
E-mail: lyj238@ios.ac.cn (for correspondence)
Jun Pang is with Faculty of Science, Technology and Communication and Interdisciplinary Centre for Security, Reliability and Trust at University of Luxembourg.
E-mail: jun.pang@uni.lu

human's insights of the protocol. It is too laborious for the user to perform these analysis, thus effective and automatic tools are needed to help.

Predicate abstraction has also been applied to the verification of parameterized protocols. Baukus et al. used it to verify the German cache coherence protocol (without data paths) [9], and Park and Dill used it to verify the FLASH protocol [3]. The main idea of predicate abstraction is to discover a set of predicates, which are needed to abstract the states of a system, and an abstract state is a valuation of the predicates. Unfortunately, the task of discover proper predicates is neither easy nor automatic. Moreover, the abstracted system is required to be proved to be conservative for certain properties under verification. This proof also needs a set of auxiliary invariants. Therefore, searching for sufficiently many auxiliary invariants cannot be avoided. After the above two works, no further efforts are made applying predicate abstraction to parameterized verification.

Conchon et al. recently have made progress in automatically searching for auxiliary invariants [14]. It first computes a set of reachable states S using a forward-image computation on a finite small instance of the parameterized system. Then it iteratively performs an over-approximate backward reachability analysis of the parameterized system and check whether it presents states that not S. The computed over-approximations will be treated as invariants and be model checked. This algorithm is implemented in an SMT-based model checker [14]. The correctness of the algorithm is argued in a generic symbolic framework. However, the formulation of the algorithm and its proof are not given in a theorem prover.

Most recently, Dooley and Somenzi [15] develop an approach to proving safety properties of parameterized systems. Their main idea is to generalize inductive proofs for small instances to quantified formulas, which are then checked as invariants against the whole family of systems using the small-model theorem [7] or the more general modest-model theorem. They illustrate that their approach is able to produce non-trivial invariants for a suite of benchmarks including hardware circuits and protocols.

Motivations. From the above discussion on related works, we can see that the degree of rigorousness and automation are two critical aspects of existing approaches to parameterized verification. The verification of real-world parameterized protocols is, however, rarely both rigorous and automatic. For instance, FLASH is the cache coherence protocol of the Stanford FLASH mutlitprocessor [16]. This protocol is so complex that only a few approaches [3], [10], [14] have successfully verified it. Furthermore, all existing successful verification approaches have their weakness. The approach of Park and Dill [3] is theorem proving based, and it requires the construction of inductive invariants manually. The work in [10] is similar to Park and Dill's [3] in that handcrafted invariants are required to provide by human users. In contrast, the approach in [14] is a model checking based approach which can be carried out in a largely automatic manner. However, the correctness of this approach has not been fully formalized and proved. Similarly, the approach in [15] is based on proof generalization with a large degree of automation. But it still requires the construction of a model in order to show that the generalized invariants hold on any instance of the parameterized system.

In order to effectively verify complex parameterized protocols like the FLASH protocol, there are two issues need to be addressed. The first one is how to find a set of sufficient and necessary invariants without (or with as little as possible) human intervention, which lies in the core of the field of parameterized verification. The second is the rigorousness of the verification itself. It is preferable to formulate all the verification in a publicly recognized trustworthy framework like a theorem prover.

In general, it is not difficult to formalize a parametrized system and its properties of a protocol in a theorem prover. However, it is very difficult hard to construct a proof to prove the properties. It has been recognized that the most creative choices in a formal proof are done by human and hard to be automated. These creative choices mainly lie in: (1) the induction scheme; (2) performing case analysis: different subproofs are done in different cases; and (3) the quantifier instantiations. Up to now, the main efforts are made in searching invariants automatically. Few existing works have considered how to link the invariant searching with proof checking. The invariants found automatically from a concrete instance of a parameterized protocol (e.g., see [6], [12], [14]) without consideration how to use them later in a theorem proving. Therefore, it is still hard to automate the above three kinds of creative choices. In more details, the invariants and protocol rules of a protocol are usually given in concrete forms in the procedure of invariant searching, but not in a parameterized form which are required later in theorem proving. The generalization from a result obtained in the concrete protocol instance to that in the parameterized instance is often not considered. Therefore, much human intervention is still needed to prove that the properties are invariance. Usually the complexity of the proofs is beyond human's comprehension. This is the reason why theorem provers are seldom used in parameterized verification.

In order to solve the identified difficulties, automation and rigorousness, in parameterized verification we need to consider the invariant searching with proof checking in a unified framework. Our key ideas are to make the aforementioned generalization automatic and to make the aforementioned creative choices in theorem proving to be automated as well.

Contributions. Our contributions in the work can be summarized as follows. JP: The following to be revised after having a complete picture of the research.

1) We propose a specific induction scheme for parameterized verification. Three types of causal relations among a set of formulas and protocol rules are identified, which are essentially special cases of the general induction principle. Notably, with proper case analysis on parameters of a parameterized protocol rule, these three induction proof rules can be applied automatically in a theorem prover.
2) A consistent relation among a protocol instance and a set of formulas is proposed based on the causal relations. If this consistency is obtained, then any

formula in the formula set is an invariant for the protocol instance. This is formulated as a theorem and proved in a theorem prover.

3) From an initially given invariant (i.e., the correctness requirement), our approach will start to search for both auxiliary invariants and causal relations from a small protocol instance, which will then be used to construct a consistent relation between the protocol instance and the set of all found invariants. Notice that both auxiliary invariants and causal relations found in this step are given in a concrete form.
4) The causal relations are identified among the protocol rules (of the small protocol instance) and the auxiliary invariants. By carefully examining the parameters in the causal relations, we generalize each auxiliary invariant and casual relation into a symbolic form. In this way, a symbolic formula is generated based on the analysis of concrete parameters, and it will be used to indicate the case condition when comparing the parameters of the symbolic invariants and the parameters of the symbolic protocol rules rules should be satisfied. Therefore, the information on case splitting on symbolic parameters are identified and kept.
5) Based on the generalization of causal relations and auxiliary invariants into proper symbolic forms, a formal proof script in a theorem prover (i.e., Isabelle [17]) can be automatically generated to prove that the consistency theorem also holds between the parameterized protocol and the set of the parameterized invariants. Since the proof script has sufficient proof commands to perform induction, and case analysis and necessary quantifier instantiation, the proof script can be automatically checked once it is submiited to the theorem prover.

Based on the above ideas, we have implemented a software tool called paraVerifier [18], and its workflow is given in Figure 1. 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 (an invariant) 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 the proposed consistency relation. When several candidate invariants are obtained using the heuristics, we use an oracle 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 can been verified by the oracle. After invFinder finds the auxiliary invariants and causal relations, proofGen generalizes them into a symbolic form, which are then used to construct a completely parameterized formal proof in a theorem prover (i.e., Isabelle) to model $\mathcal{P}$ and to prove the property inv. The generated proof can be checked automatically.

Structure of the paper. The organization of this work is as follows: Section 2 introduces the preliminaries and notations on modeling parameterized protocols; and Section 3 introduces our induction scheme, including the proposed casual relations and the consistency theorem. An overview of our approach is given in Section 4. In the following sections, Section 5 describes the main steps of invFinder to identify auxiliary invariants with causal relations for the aim of proving correctness of a parameterized protocol. Section 6 presents how to generalize the found (auxiliary) invariants and causal relations into a symbolic form; Section 7 focuses on proofGen, an automatic proof generation based on the results of generalization. An application of our approach to the FLASH protocol is presented in Section 8. Section 9 presents a summary on our experiments on other real-world protocols. Section 10 concludes our work with a few future research directions.

2 Preliminaries

In this section, we introduce a number of notations and preliminary concepts. We assume a finite set of state variables $V = \{v_0, v_1, \ldots, v_n\}$. We also assume that the variables in V range over a finite set D. With these variables, first order expressions e and formulas (predicates) f can be defined as usual. Variables are divided into two classes: array variables or non-array (global) variables. A state s of a protocol is an instantaneous snapshot of its behavior given by a mapping from all variables in V to D. We use $s(v_i)$ to get the particular value for the variable v_i at the state s. We write $\mathbb{A}[e, s]$ to denote the evaluation result of the expression e at the state s, and use $s \models f$ to denote that the formula f is evaluated to be true at the state s. For parallel assignments, we write $A = \{v^i := e^i \mid i \geq 0\}$. We use $\mathsf{vars}(A)$ to denote all the variables appearing in A, namely $\mathsf{vars}(A) = \{v^0, v^1, \ldots, v^i\}$. We define the notion of the weakest precondition $\mathsf{WP}(f, A) \equiv f[v^i := e^i]$, which substitutes each occurrence of v^i by e^i in formula f.

Protocols. A cache coherence protocol $\mathcal{P}$ is formalized as a pair (I, R), where (1) I is an initialization (a set of predicates) over V; and (2) R is a set of protocol rules. Each rule $r \in R$ is defined as $g \rhd A$, where g is a predicate over V, and A is a parallel assignment to distinct variables v_i with expressions e_i. We write $\mathsf{guard}(r) = g$, and $\mathsf{action}(r) = A$ for a protocol rule r. A state transition is caused by trigging and executing a protocol rule r. Formally, we define:

$$\begin{aligned} s \xrightarrow{r} s' \quad \equiv \quad & s \models \mathsf{guard}(r) \wedge \\ & \forall v := e \in \mathsf{action}(r).\, s'(v) = \mathbb{A}[e, s] \wedge \\ & \forall v' \notin \mathsf{vars}(\mathsf{action}(r)).\, s'(v') = s(v) \end{aligned}$$

Reachable state sets. As usual, the reachable set of states for a protocol $\mathcal{P} = (I, R)$ denoted as $\mathsf{RS}(\mathcal{P})$, can be defined inductively: (1) a state s is in $\mathsf{RS}(\mathcal{P})$ if there exists a formula $f \in I$ such that $s \models f$; (2) a state s is in $\mathsf{RS}(\mathcal{P})$ if there exists a state s' and a protocol rule $r \in R$ such that $s' \in \mathsf{RS}(\mathcal{P})$ and $s' \xrightarrow{r} s$.

Parameterized formulas, rules, and protocols. For simplicity, a parameterized formula (rule, and protocol) is a function $f(x_1, x_2, ..., x_n)$ from a tuple of natural numbers to such an object. JP: The above formulation is difficult to understand. It is also unclear why talking about lists below. Without losing generality, we require that parameters to instantiate a parameterized object are disjoint. For instance, $\mathsf{mutualInv}(i, j) \equiv \neg(a[i] = \mathsf{C} \wedge a[j] = \mathsf{C})$ is a parameterized

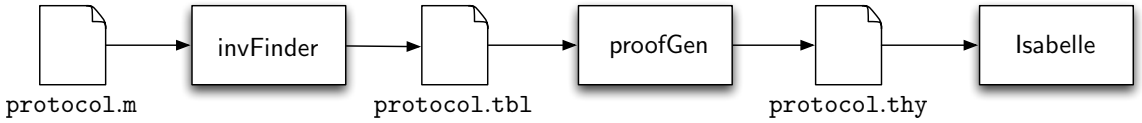

Fig. 1. The workflow of paraVerifier.

formula with two disjoint parameters. JP: The above example cannot be used here, as it is related to the example below. A list of natural numbers which are different from each other are actual parameters to instantiate a parameterized object. Thus, we need a few functions on lists. We write [] for an empty list, $x\#xs$ for the list that extends xs by adding x to the front of the list xs, $[x_1, \ldots, x_n]$ for a list $x_1\#..\#x_n\#[]$, $xs@ys$ to have the resulted list by concatenating xs with ys, $xs[i]$ to retreive the i^{th} element of the list xs (counting from 1 as the first element), set xs for getting the set of all the elements in xs, $\mid xs \mid$ for the length of the list xs, $\mathsf{head}(xs)$ to get the head element of a non-empty list, and $\mathsf{tail}(xs)$ to get the tail of xs.

Running example. Now we illustrate the above definitions by a simple mutual exclusion protocol with N nodes. Let I(dle), T(rying), C(ritical), and E(xiting) be the enumerated values to indicate the state of a node in the protocol. x is a simple non-array variable, used as a flag to indicate the starting state of a protocol node. a is used to an array of nodes. N denotes a natural number. $\mathsf{pini}(N)$ is the predicate to specify the initial state of the protocol, and $\mathsf{prules}(N)$ describes four protocol rules. $\mathsf{mutualInv}(i, j)$ defines a property that $a[i]$ and $a[j]$ cannot be in a critical state at the same time. For this parameterized protocol, we need to verify that $\mathsf{mutualInv}(i, j)$ holds at any reachable state of th for any $i \neq j \leq N$.

Example 1 *A simple mutual exclusion protocol.*

```
pini(N) ≡ x=true ∧ ⋀_{i=1}^{N} a[i]=I
try(i) ≡ a[i]=I ▷ a[i]:=T
crit(i) ≡ a[i]=T ∧ x=true ▷ a[i]:=C; x:=false
exit(i) ≡ a[i]=C ▷ a[i]:=E
idle(i) ≡ a[i]=E ▷ a[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) ≡ ¬(a[i]=C ∧ a[j]=C)
```

3 Casual Relations and The Consistency Theorem

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. Given a protocol $\mathcal{P} = (I, R)$, for a state of the protocol s, a protocol rule $r \in R$, a formula f over V, and a formula set F, three kinds of causal relations are defined as follows:

Definition 1 *We define the following relations:*

1) $\mathsf{CR}_1(s, f, r) \equiv s \models \mathsf{guard}(r) \longrightarrow s \models \mathsf{WP}(f, \mathsf{action}(r))$;
2) $\mathsf{CR}_2(s, f, r) \equiv s \models f \longleftrightarrow s \models \mathsf{WP}(f, \mathsf{action}(r))$;
3) $\mathsf{CR}_3(s, f, r, F) \equiv \exists f' \in F$ *such that* $s \models (f' \wedge (\mathsf{guard}(r)) \longrightarrow s \models \mathsf{WP}(f, \mathsf{action}(r))$.

We write $\mathsf{CRS}(s, f, r, F)$ as $\mathsf{CR}_1(s, f, r) \vee \mathsf{CR}_2(s, f, r) \vee \mathsf{CR}_3(s, f, r, F)$, it defines a casual relation between f, r, and F, which guarantees that if each formula in F holds before the execution of a protocol rule r, then f also holds after the execution of rule r. This captures the intuitive meaning of invariants. There are three cases: 1) $\mathsf{CR}_1(s, f, r)$ means that after rule r is executed, f becomes true immediately; 2) $\mathsf{CR}_2(s, f, r)$ states that $\mathsf{WP}(f, \mathsf{action}(r))$ is equivalent to f, which intuitively means that none of state variables in f is changed, and the execution of the parallel assignments $\mathsf{action}(r)$ does not affect the evaluation of f; 3) $\mathsf{CR}_3(s, f, r, F)$ states that there exists another invariant $f' \in F$ such that the conjunction of the guard of r and f' implies the weakest precondition $\mathsf{WP}(f, \mathsf{action}(r))$.

In Hoare logic, a Hoare triple has the form of $\{f\}S\{f'\}$ where f and f' are assertions of formulas and S is a statement. f is called the precondition and f' the postcondition: when the precondition is met, executing S establishes the postcondition. We can interpret the above three kinds of causality relation in Hoare triples as below:

1) $\mathsf{CR}_1(s, f, r)$ iff $\{\mathsf{guard}(r)\}\mathsf{action}(r)\{f\}$
2) $\mathsf{CR}_2(s, f, r)$ iff $\{\mathsf{guard}(r) \wedge f\}\mathsf{action}(r)\{f\}$
3) $\mathsf{CR}_3(s, f, r, F)$ iff $\exists f' \in F$ such that $\{\mathsf{guard}(r) \wedge f'\}\mathsf{action}(r)\{f\}$

$\mathsf{CRS}(s, f, r, F)$ can be considered as a kind of general inductive tactics. In other words, a property f in F holds at a state s, and $\mathsf{CRS}(s, f, r, F)$, then f also holds at the post-state s' after a rule r is executed. This is captured as a lemma as follows:

Lemma 1 *Let s and s' be two states and r be a protocol rule such that $s \xrightarrow{r} s'$, if $s \models f$ and $\mathsf{CRS}(s, f, r, F)$ for any $f \in F$, then for any $f \in F$, $s' \models f$.*

Instead of using the general induction rule (or proving the general causal relation $\mathsf{CRS}(s, f, r, F)$, we classify it into three special kinds of causal relations $\mathsf{CR}_{1-3}(s, f, r, F)$ because the latter are more-fine grained and easy to be checked by a theorem prover. In fact, because each formula in F holds at s, they can be used as induction hypothesis. Thus, $\mathsf{CR}_{1-3}(s, f, r, F)$ inform the theorem prover how to use the premise in the induction hypothesis to prove the precondition $WP(f, \mathsf{action}(r))$. We will illustrate this through examples later.

With the defined CRS relation, we formulate a consistency relation $\mathsf{consistent}(INV, I, R)$ between a protocol (I, R) and a set of invariants $INV = \{inv_1, \ldots, inv_n\}$.

Definition 2 *A relation* consistent(INV, I, R) *holds if the following two conditions hold:*

1) *for any formula* $inv \in INV$*, predicate* $f \in I$*, and state* s*,* $s \models f$ *implies* $s \models inv$*;*
2) *for any formula* $inv \in INV$*, rule* $r \in R$ *and state* s*,* $\mathsf{CRS}(s, inv, r, INV)$*.*

Running example. We continue to illustrate the above definitions on the mutual exclusion protocol. We first give a few auxiliary invariants for proving $\mathsf{mutualInv}(i, j)$.

Example 2 *Let us define*

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

JP: The above pinvs(N) is very confusing without any good explanation. The problem is that the formulation is done in Isabelle and it looks odd. In Example 2, $\mathsf{invOnXC}(i)$ (or $\mathsf{invOnXE}(i)$) specifies that the variable x will be set to be false once the node i is staying (or exiting) the critical section. $\mathsf{aux1}(i, j)$ says that nodes i and j cannot be in and exiting the critical section at the same time, while $\mathsf{aux2}(i, j)$ states that nodes i and j cannot exit critical section at the same time.

We continue to illustrate the three kinds of casual relations.

Example 3 *Given* s *a state of the simple mutual exclusion protocol. Suppose that* $inv = \mathsf{mutualInv}(i_1, i_2)$*,* $pinvs(N)$*,* $r = \mathsf{crit}(iR_1)$*,* $rs = \mathsf{prules}(N)$*, and the constraints* $i_1 \leq N$*,* $i_2 \leq N$*,* $i_1 \neq i_2$*, and* $iR_1 \leq N$*. We have the following casual relations.*

- $\mathsf{CR}_2(s, inv, r)$*, with* $i_1 \neq iR_1$*, and* $i_2 \neq iR_1$*. This is because* $\mathsf{WP}(inv, \mathsf{action}(r)) = inv$*.*
- $\mathsf{CR}_3(s, inv, r, pinvs)$*, with* $i_1 = iR_1$*. This is because* $\mathsf{invOnXC}(i_2) \in pinvs$*,* $\mathsf{WP}(inv, \mathsf{action}(r)) = \neg(\mathsf{C} = \mathsf{C} \wedge a[i_2] = \mathsf{C})$*,* $\mathsf{invOnXC}(i_2) \wedge \mathsf{guard}(r) \longrightarrow \neg a[i_2] = \mathsf{C}$*, and* $s \models \neg a[i_2] = \mathsf{C}$ *implies* $s \models \neg(\mathsf{C} = \mathsf{C} \wedge a[i_2] = \mathsf{C})$*.*
- $\mathsf{CR}_3(s, inv, r, pinvs)$*, with* $i_2 = iR_1$*. This is because* $\mathsf{invOnXC}(i_1) \in pinvs$*,* $\mathsf{WP}(inv, \mathsf{action}(r)) = \neg(\mathsf{C} = \mathsf{C} \wedge a[i_1] = \mathsf{C})$*,* $\mathsf{invOnXC}(i_1) \wedge \mathsf{guard}(r) \longrightarrow \neg a[i_1] = \mathsf{C}$*, and* $s \models \neg a[i_1] = \mathsf{C}$ *implies* $s \models \neg(\mathsf{C} = \mathsf{C} \wedge a[i_1] = \mathsf{C})$*.*

JP: In the above example, both item 2 and item 3 require some explanation. They are very similar, why we need them? And how about removing $C = C$? We also miss a case for CR_1.

In Example 3, $\mathsf{CR}_1(s, inv, r)$ and $\mathsf{CR}_2(s, inv, r)$ can be checked automatically by a theorem prover. $\mathsf{CR}_3(s, inv, r, pinvs)$ can also be checked automatically if the proper formula (such as $\mathsf{invOnXC}(i_2)$ in the example) can be provided for the instantiation of the existence quantifier. We notice that two things are needed to be done to guide a theorem prover to automatically check $\mathsf{CRS}(s, inv, r, pinvs)$: (1) the case splitting which is decided by comparison between rule parameter iR_1 and invariant parameters i_1 and i_2; (2) the choice among the three kinds of causal relations to prove in each case.

For the simple mutual exclusion protocol, we can check the consistent relation holds between the auxiliary invariant set $pinvs$ and the protocol initial predicate $\mathsf{pini}(N)$ and rules of the mutual exclusion protocol $\mathsf{prules}(N)$.

Lemma 2 *If* $\mathcal{P} = (\mathsf{pini}(N), \mathsf{prules}(N))$ *is the protocol as listed in Example 1, and* $pinvs$ *is the set of formulas in Example 2, then we have* $\mathsf{consistent}(pinvs, \mathsf{pini}(N), \mathsf{prules}(N))$*.*

Proof. By unfolding the definition of consistent, we need to verify that parts (1) and (2) of the consistent relation hold.

For part (1), the proof is rather straightforward. We only need to perform case analysis on the form of a formula f in $pinvs$, and check whether $\mathsf{pini}(n)$ implies f. For instance, consider the case where $inv = \mathsf{mutualInv}(i_1, i_2)$ for some i_1 and i_2, where $i_1 \leq N$, $i_2 \leq N$, and $i_1 \neq i_2$. We can conclude that $s \models a[i_2] = I$ if $s \models \mathsf{pini}(N)$, thus $s \models inv$ holds. The other invariants in $pinvs$ can be proved similarly.

For part (2), we perform case analysis on the form of a formula f in $pinvs$ first, and then on the form of r in $\mathsf{prules}(N)$. Notice that both f and r are parameterized, thus we need to again perform case analysis by comparing indices in f and r and show CR_{1-3} hold. Example 3 has already shown a few typical cases with $inv = \mathsf{mutualInv}(i_1, i_2)$, and $r = \mathsf{crit}(iR_1)$, where $iR_1 \leq N$, $i_1 \leq N$, and $i_2 \leq N$. ■

We now analyze the complexity of proving part (2) for Lemma 2. For one protocol rule, we need to perform case analysis for each invariant in $pinvs$. In total, we have 52 cases for such a simple protocol with only 4 protocol rules. The number of case analyses will increase dramatically for large protocols. This complexity illustrates the difficulty of parameterized verification of cache coherence protocols. It also explains why there are very few successful case studies in applying a general theorem prover to verifying cache coherence protocols.

In general, for any invariant $inv \in invs$, inv holds at a reachable state s of a protocol $\mathcal{P} = (I, R)$ if the consistency relation $\mathsf{consistent}(invs, I, R)$ holds.

Theorem 3 *If* $\mathcal{P} = (I, R)$*,* $\mathsf{consistent}(invs, I, R)$*, and* $s \in \mathsf{RS}(\mathcal{P})$*, then for all* $inv \in invs$*,* $s \models inv$*.*

Theorem 3 is our main tool to prove that any property inv in a formula set $invs$ is an invariant for a protocol $\mathcal{P} = (I, R)$. It eliminates the need of directly use of induction proof methods. We only check the causal relation between inv and a protocol rule $r \in R$ by case analysing on inv and r. Next we apply Theorem 3 to proving that the mutual exclusion property $\mathsf{mutualInv}(i, j)$ holds for each reachable state of the simple mutual exclusion protocol in Example 1. In order to prove the mutual exclusion property, we prove a more general result:

Lemma 4 *$\mathcal{P} = (\mathsf{pini}(N), \mathsf{prules}(N))$ is the protocol as listed in Example 1, $s \in \mathsf{RS}(\mathcal{P})$, $0 < N$, and $pinvs$ is the set of formulas in Example 2. For any $inv \in pinvs$, we have $s \models inv$.*

Proof. By Theorem 3, we only need to check that $\mathsf{consistent}(pinvs, \mathsf{pini}(N), \mathsf{prules}(N))$ holds. This can be immediately obtained by Lemma 2. ■

4 An Overview of Our Approach

In order to apply Theorem 3 to prove that a given parameterized property inv (e.g., the mutual exclusion property) is an invariant for a parameterized protocol (e.g., the simple mutual exclusion protocol), we need to solve the following two problems.

1) We need to construct a set of auxiliary invariants $invs$ which contains inv and satisfies $\mathsf{consistent}(invs, I, R)$. Constructing a set of auxiliary invariants is the central problem in the filed of parameterized verification.
2) When applying Theorem 3, we actually decompose the original problem of invariant checking into that of checking that some causal relation holds between a formula $f \in invs$ and $r \in R$. Then we need to perform three levels of case analysis: the first is on the form of f, the second is on the protocol rule r, and the last is to compare parameters in r and parameters in f (for instance, see the proof of Lemma 2 and Example 3). Finally, we need to identify a causal relation holds between f and r for each of the cases. How to generate enough information to construct a proof which consists of the above case analysis is the second problem to be solved in our approach.

To tackle these two problems on w.r.t. a parameterized property and a parameterized protocol, we first focus on a concrete property and a concrete protocol instance. Here the concrete property is obtained by instantiating the parameters with concrete indices, and by fixing the parameterized size with concrete value. If we can find answers to the two problems for the concrete ones, then we generalize them into answers to the two problems for the parameterized ones.

In the second problem, to know which causal relation holds, we can select a special instance of the generalized symbolic case, and compute the precondition of f w.r.t. $\mathsf{action}(r)$, then test in a concrete protocol instance which causal relation holds among f, r, and the current set of invariant formulas $invs$. If either causal relation CR_1 or CR_2 holds, then this causal relation is recorded. Otherwise, we need to consider two cases: (1) there is a formula in $invs$ to make CR_3 hold; (2) there is no formula in $invs$ to make CR_3 hold. In the second case, we have to try to construct a new invariant formula f' which makes the relation $\mathsf{CR}_3(s, f, r, invs \cup \{f'\})$ hold. The combined process of checking causal relation and generating new invariants is not finished until no new invariant formula can be found. The returned result also contains a table which records all the causal relation among the newly found invariant formulas and the concrete rules and the set of auxiliary invariant formulas.

Based on this table recording the information on causal relation, we generalize it into a parameterized form. Each line of table is associated with a symbolic formula to indicate a case by comparing the symbolic parameters of a parameterized rule and those of a parameterized invariant formula. Furthermore, a formal proof script is generated according the table on the generalized information.

5 invFinder: Finding Auxiliary Invariants

Given a parameterized protocol $\mathcal{P}$ and a property set F containing concrete invariant formulas, each of which is an instantiation of a parameterized invariant formula we want to initially verify, invFinder explores Algorithm 1 to find (1) useful auxiliary invariants and (2) causal relations that are capable of proving any property in F.

A set A is used to store all the invariants found so far now, and it is initialized as F. A relation table tbl is used to record the causal relation between a parameterized rule in some parameter setting and a concrete invariant. Initially tbl is set as empty. A queue $newInvs$ is used to store invariants which have not been checked, and it is initialized as F. The tool invFinder works iteratively in a semi-proving and semi-searching way. In each iteration, the head element cf of $newInvs$ is popped, then $\mathsf{Policy}(r, cf)$ generates groups of parameters $paras$ according to a protocol r and the formula cf. For each parameter $para$ in $paras$, $para$ is applied to instantiate the protocol rule r into a concrete rule cr. This policy should guarantee that each case split by the aforementioned parameter comparison scheme should be sampled. Here $\mathsf{apply}(r, para) = r$ if r contains no parameters and $para = []$; otherwise $\mathsf{apply}(r, para) = r(para_{[1]}, \ldots, para_{[|para|]})$. Then $\mathsf{coreFinder}(cr, cf, A)$ is called to check which kinds of causal relation exists between cr and f. If there is 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. A tuple $(r, para, cf, rel)$ will be inserted into tbl to indicate some causal relation rel exits among cr and $cf$and $f \cup A$. If the formula option $newInvOpt$ is empty, then no new invariant formula is generated; otherwise $newInvOpt = \mathsf{Some}(cf')$ for some formula cf', then $\mathsf{get}(newInvOpt)$ returns cf', and the new invariant formula cf' 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 tbl and the augment invariant set A are returned.

Note that, in Algorithm 1, $\mathsf{Policy}(r, cf)$ analyzes the number of the formal parameters of r and that of actual parameters cf respectively, and generates groups of concrete parameters, each of which will be used to instantiate r into a concrete rule cr. The core invariant searching function $\mathsf{coreFinder}(cr, cf, A)$ JP: a better name later! returns the causal relation among cr and $cf$and A. The two function Policy and coreFinder will be explained in Section 5.1 and Section 5.2, respectively.

5.1 Parameter Generation Policy

Let cf to be a special instance of some parameterized formula f, in a way that $cf = \mathsf{apply}(f, idp(|\ \mathsf{apOfCForm}(cf)\ |))$. JP: What is apOfCForm? It is strange that cf is obtained

Algorithm 1: The main algorithm of invFinder

Input: Initially given a set of invariants F, a protocol $\mathcal{P} = (I, R)$

Output: tbl : a table which represent causal relations between concrete protocol rules and invariants; and A: a set of concrete invariant formulas

```
A ← F;
tbl ← [];
newInvs ← F;
while newInvs ≠ [] do
    cf ← head(newInvs);
    newInvs ← tail(newInvs);
    for r ∈ R do
        paras ← Policy(r, cf);
        for para ∈ paras do
            cr ← apply(r, para);
            newInvOpt, rel ← coreFinder(cr, cf, A);
            tbl ← tbl@[(r, para, cf, rel)];
            if newInvOpt ≠ NONE then
                newInv ← get(newInvOpt);
                newInvs ← newInvs@[newInv];
                A ← A ∪ {newInv};
return A, tbl.
```

by applying a function to itself. cf is just a formula. Recall that the aim of our policy is to compute groups of protocol rule parameters to instantiate a parameterized rule into a set of concrete rules. Combinations of any group of the above generated rule parameters with the actual parameters occurring in the concrete invariant formula cf will be a special instantiation of some case where we compare symbolic parameters in r and those in f. JP: We need to give the term 'symbolic parameters' earlier. Furthermore, each of the subcases should be sampled, JP: What do we mean by 'sampling'? which are partitioned by the aforementioned parameter comparison scheme.

Suppose that the number of actual parameters occurring in cf is n, and that of formal parameters occurring in r is n', we choose n'–element subset among $\{1, \ldots, (n+n')\}$ to instantiate r. Namely, a n'-permutation of $n+n'$, $para$ will be treated as a group of rule parameters to instantiate r. Furthermore, if $n' > 0$ and $i < n'$, and $para[i] > n$, JP: We never have $para[i]$ before. So what is the precise definition of $para$? then $para[i]$ will not equal to any actual parameters in cf. Thus, we don't need to care which value $para[i]$ is if JP: we can guarantee $para[i]$ is???. In the sense of sampling cases, $para$ is the equivalent to $para'$ if $para'$ is the updated result by only replacing $para[i]$ with another index j with $j > n$. For instance, let $n = n' = 2$, then $[1,2]$, $[2,3]$, $[2,4]$ are groups of rule parameters to instantiate r. $[2,3]$ and $[2,4]$ are equivalent to each other in the sense of sampling cases. Thus, if we have used $[2,3]$ to instantiate r, we don't need to use $[2,4]$ to instantiate r again.

In order to formulate our parameter generation policy, we define:

Definition 3 *Let m and n be two natural numbers with $n \leq m$, $para$ and $para'$ are two lists which stand for two n-permutations of m, defined as follows*

1) $\mathsf{normPara}(para, n)[i] =$

$$\begin{cases} para[i] & \text{if } para[i] \leq n \quad (1) \\ n+1 & \text{otherwise} \quad (2) \end{cases}$$

2) $para \sim_m^n para' \equiv \mathsf{normPara}(para, n) = \mathsf{normPara}(para', n)$
3) $\mathsf{semiP}(m, n, S) \equiv (\forall para \in \mathsf{perms}_m^n \; \exists para' \in S.\; para \simeq_m^n para') \wedge (\forall para \in S.\; \forall para' \in S.\; para \neq para' \longrightarrow (para \not\simeq_m^n para')$. *A set S is called a quotient of the set perms_m^n under the relation $\simeq_m^n$ if $\mathsf{semiP}(m, n, S)$.*

JP: In the above definition, what is S?

In Definition 3, the function $\mathsf{normPara}(para, n)$ is used to normalize a group of parameters by uniformly replacing the i-th element with $n+1$ if it is greater than n. Naturally the aforementioned equivalence between two groups of parameters $para$ and $para'$, w.r.t., sampling is defined by the normalized form of them are the same. Based on the relation $\sim_m^n$, we can define a quotient of the set perms_m^n under the relation $\simeq_m^n$. Here a quotient of $\mathsf{perms}_{n+n'}^{n'}$ is the set of all groups of parameters to instantiate r to check the causal relation between r and cf in the above paragraph. JP: I need to fully understand the meaning to revise.

Example 4 *Recall that the number of actual parameters occurring in cf is n, and that of formal parameters occurring in r is n'.*

1) *If $n = 2$, $n' = 1$, then $paras = \{[1],[2],[3]\}$. $paras$ is the quotient set of parameter groups to instantiate r;*
2) *If $n = 2$, $n' = 2$, then $paras = \{[1,2],[1,3],[2,1],[2,3],[3,4]\}$. $paras$ is the quotient set of parameter groups to instantiate r;*

JP: In the above example, what is $paras$?

Now we formally define the symbolic case which is generalized from a group of rule parameters if we compare it with an identical permutation with length n which represents a group of invariant formula parameters.

JP: For formal definitions, it is better to give intuitions first, why it is needed and then formalisation.

Definition 4 *Let $para$ be a permutation and $n > 0$, JP: Why $para$ is a permutation? we define:*

$\mathsf{caseGen}([], n, pos) = []$

$\mathsf{caseGen}(i\#paran, pos) =$

$$\begin{cases} (iR_{pos} = iInv_i) \wedge \mathsf{caseGen}(paran, pos+1) & \text{if } i \leq n \\ (\bigwedge_{j=1}^{n} iR_{pos} \neq iInv_j) \wedge \mathsf{caseGen}(paran, pos+1) & \text{otherwise} \end{cases}$$

JP: I am completely lost in the above definition.

We illustrate the meaning of caseGen by the following example.

Example 5 *Recall the example 4,*

1) $\mathsf{caseGen}([1], 2, 1) = iR_1 = iInv_1$
2) $\mathsf{caseGen}([3], 2, 1) = \bigwedge_{j=1}^{2} iR_i \neq iInv_j)$

Algorithm 2: Computing a quotient of perms_m^n

Input: m, n
Output: A list of permutations L

```
1 L_0 ← perms_m^n;
2 L ← [];
3 while L_0 ≠ [] do
4     para ← hedd(L_0);
5     L_0 ← tail(L_0);
6     if ∀para' ∈ set(L). para' ≉_m^n para then
7         L ← L@[para];
8 return L.
```

3) $\mathsf{caseGen}([1,3],2,1) = iR_1 = iInv_1 \bigwedge_{j=1}^{2} iR_2 \neq iInv_j)$

In the end, we give an algorithm to compute a quotient set of perms_m^n. JP: Is this algorithm actually used again in the rest of the paper? JP: perms_m^n is not defined.

Algorithm 2 computes a quotient of perms_m^n. First, it set $L_0 = \mathsf{perms}_m^n$, then it fetches the head element of L_0 into L, and find whether there is one element $para'$ in L such that $para \simeq_m^n para'$. If it is the case, then $para$ will be discarded. Otherwise, $para$ is inserted into L. This procedure is repeated until L_o is empty. JP: $\mathsf{Policy}(cf, r)$ is simply $\mathsf{perms}_{n+n'}^{n'}$.??? If it is the case, we shall be consistent, use Policy instead. But how about cf and r, which are never used in Algorithm 2.

Take the example of the simple mutual exclusion protocol. For the invariant $\mathsf{mutualInv}(1,2)$, according to Policy, three groups of parameters [1], [2], [3] are used to instantiate the rule crit respectively. Each of the instantiation results will be used to check which causal relation holds between the derived concrete rule and $\mathsf{mutualInv}(1,2)$.

5.2 Core Searching Algorithm

For a concrete cf, a concrete protocol rule cr and a set of formulas of found invariants $invs$, Algorithm 3 checks the causal relation among cf, cr and $invs$, as well as finds a new invariant if it is needed. The algorithm returns a formula as an option and a causal relation item between r and inv. JP: r or cr?inv or cf?

In order to explain coreFinder, we first need to introduce some functions on symmetry transformation of a formula. We define $\mathsf{indices}(f)$ to denote the list of concrete parameters occurring in f, which is arranged by the pre-order traversal of the syntax of f. A bijection π induced from a list L of mutually-different natural numbers is the mapping $\pi(i) = L[i]$ for any i such that $i \leq |L|$. As usual, we use π^{-1} to denote the inverse function of π. We use $\mathsf{induced}(L)$ to denote the bijection induced from a list L, $\mathsf{symApp}(\pi, f)$ to denote the formula obtained by simultaneously replacing all occurrences of each i with $\pi(i)$. A formula f is symmetric to another f' if there is a bijection $\pi(i)$ such that $\mathsf{symApp}(\pi, f) = f'$. A normalized form of a concrete formula f is defined as $\mathsf{normalize}(f) \equiv \mathsf{symApp}((\mathsf{induced}(\mathsf{indices}(L)))^{-1}, f)$.

coreFinder needs to call two oracles. The first one, denoted by `chk`, checks whether a concrete formula is an invariant. Such an oracle can be implemented by translating the formula into a formula in the model checker SMV, and then asking SMV to check whether it is an invariant in a given small reference instance of the protocol. If the reference instance is too small to check the invariant, then the formula will be checked by Murphi in a big reference model. JP: It is unclear why SMV is used to check small instances while Murphi is used for larger ones. 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 3: Core searching algorithm: coreFinder

Input: $cr, cf, invs$
Output: A formula option $fOpt$, and a new causal relation rel

```
1  g ← guard(cr);
2  cf' ← WP(cf, action(cr));
3  if cf = cf' then
4      rel ← (cr, cf, CR_2, −);
5      return (NONE, rel);
6  else if tautChk(g → cf') = true then
7      rel ← (cr, cf, CR_1, −);
8      return (NONE, rel).
9  else
10     candidates ←
       subsets(decompose(dualNeg(cf') ∧ g));
11     newInv ← choose(chk, candidates);
12     rel ← (cr, cf, CR_3, newInv);
13     if isNew(newInv, invs) then
14         newInv ← normalize(newInv);
15         return (SOME(newInv), rel).
16     else
17         return (NONE, rel).
```

Algorithm coreFinder works as follows: after computing cf' (line 2), which is the weakest precondition of the input formula cf w.r.t. $\mathsf{action}(cr)$, the algorithm takes further operations according to the following cases:

1) If $cf = cf'$, meaning that statement $\mathsf{action}(cr)$ does not change cf, then no new invariant is created, and the new causal relation item marked with tag CR_2 is recorded between cr and cf. At this moment there are no new invariants to be added.
   For example, let $cr = crit(3)$, $cf = \mathsf{mutualInv}(1,2)$, thus $cf' = cf$, then a tuple $(\mathsf{NONE}, (crit(3), inv, \mathsf{CR}_2, \_))$ will be returned.
2) If `tautChk` verifies that $g \rightarrow cf'$ is a tautology, then no new invariant is created, and the new causal relation item marked with tag CR_1 is recorded between cr and cf.
   For example, let $cr = crit(2)$, $cf = \mathsf{invOnXC}(1)$, $cf' = \neg(\mathsf{false} = \mathsf{true} \wedge n[1] = \mathsf{C})$. Obviously, $g \rightarrow f'$ holds, thus a tuple $(\mathsf{NONE}, (crit(2), inv, \mathsf{CR}_1, \_))$ will be returned.
3) If neither of the above two cases holds, then a

new auxiliary invariant $newInv$ will be constructed, which will make the causal relation CR_3 to hold. The candidate set is $\mathsf{subsets}(\mathsf{decompose}(\mathsf{dualNeg}(cf') \wedge g))$, where $\mathsf{decompose}(f)$ decomposes 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 $\bigwedge_{i=1}^{N} f_i$ for some N. $\mathsf{dualNeg}(\neg f)$ returns f. $\mathsf{subsets}(S)$ denotes the power set of a set 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 $\mathsf{isNew}(newInv, invs)$ checks whether $newInv$ is new w.r.t. $invs$. If this is the case, the invariant will be normalized and then added into $invs$, and the new causal relation item marked with tag CR_3 will be added into the causal relations. Here the meaning of $\mathsf{isNew}(newInv, invs)$ is that $newInv$ is not equivalent to any formula in $invs$.
For example, let $invs = \emptyset$, $r = crit(1)$, $inv = \mathsf{mutualInv}(1,2)$, $cf' = \neg(\mathsf{true} = \mathsf{true} \wedge n[2] = C)$, from all the subsets of $\{a[1] = T, x = \mathsf{true}, a[2] = C\}$, the coreFinder calls choose to select the subset $\{x = \mathsf{true}, a[2] = C\}$, combines all the item in this set, then constructs a new formula $f_0 = \neg(x = \mathsf{true} \wedge n[2] = C)$. After normalizing f_0, the resulting new invariant $newInv = \neg(x = \mathsf{true} \wedge a[1] = C)$ and a relation item $(crit(1), \mathsf{CR}_3, f_0)$ are returned. JP: Verify the details of this example later.

Let us continue the example in the end of Section 5.1. After the three iterations of computations of coreFinder on $crit(1)$, $crit(2)$, $crit(3)$ with $\mathsf{mutualInv}(1,2)$, the corresponding outputs 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 CR_3). JP: 'ground' is not used before. JP: Please check the table, it does not contain the information as described above.

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	

6 Casual Relation Generalization

Intuitively, generalization means that a concrete index (formula or rule) is generalized into a set of concrete indices (formulas or rules), JP: The above sentence is quite ambiguous. which can be formalized by a symbolic index (formula or rules) with side conditions specified as 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 $\mathsf{symb}(\text{“}N\text{”})$, 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 $\mathsf{symbolize}(g, i) = \mathsf{symb}(g(i))$. In this work, two special transforming functions $\mathsf{fInv}(i) = \text{“}iInv\text{”}\hat{}\,\mathsf{itoa}(i)$ and $\mathsf{fIr}(i) = \text{”}iR\text{”}\hat{}\,\mathsf{itoa}(i)$, 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 protocol 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 mainly two types of generalization in our work. The first is the 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, $\neg(x = \mathsf{true} \wedge a[1] = \mathsf{C})$ is generalized into $\neg(x = \mathsf{true} \wedge a[\mathtt{iInv_1}] = \mathsf{C})$. This kind of generalization is done with model constraints, which specify that any parameter index should be not greater than the instance size `N`, and parameters to instantiate a parameterized rule (formula) should be different. The second is the generalization of concrete causal relations into parameterized causal relations in Isabelle, which will be used later in proofs of the existence of causal relations in Isabelle.

Since the first type of generalization is usually simple, we focus on the second type of generalization, which consists of two phases. Firstly, groups of rule parameters in the form of $[i]$ (e.g., see $try[i]$ and $crit(i)$ in the simple mutual exclusion protocol) will be generalized into a list of symbolic formulas (e.g., $[\mathtt{iR_1} = iInv_1, \mathtt{iR_1} = iInv_2, (\mathtt{iR_1} \neq iInv_1) \wedge (\mathtt{iR_1} \neq iInv_2)]$, 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 CR_3 relation is generalized by some special strategy. JP: The above last sentence is unclear. Please be more specific. 'formula field' is never mentioned before.

For the first phase, let us consider a line of concrete causal relation shown in Table 1 as an example. There is a group of rule parameters (denoted as LR), and a group of parameters occurring in an invariant formula (denoted as LI). JP: To have better names for LI and LR.

Definition 5 *Let LR be a permutation and $|LR| > 0$, JP: What is LR actually? 'permutation' on what? It is a group of parameters. which represents a list of actual parameters to instantiate a parameterized rule. Let LI be a permutation and $|LI| > 0$, which represents a list of actual parameters to instantiate a parameterized invariant. We define:*

1) *symbolic comparison condition generalized by comparing $LR[i]$ and $LI[j]$:*
$\mathsf{symbCmp}(LR, LI, i, j) \equiv$
$$\begin{array}{ll} \mathtt{iR_i} = iInv_j & \text{if } LR[i] = LI[j] \\ \mathtt{iR_i} \neq iInv_j & \text{otherwise} \end{array}$$
2) *symbolic comparison condition generalized by comparing $LR[i]$ and with all $LI[j]$:*

$\mathsf{symbCaseI}(LR, LI, i) \equiv$

$$\begin{array}{ll} \mathsf{symbCmp}(LR, LI, i, j) & \textit{if } \exists!j.LR[i] = LI[j] \\ \mathsf{forallForm}(|LI|, pf) & \textit{otherwise} \end{array}$$

where $pf(j) = \mathsf{symbCmp}(LR, LI, i, j)$*, and* $\exists!j.P$ *is an qualifier meaning that there exists a unique* j *such that property* P *holds; JP: I don't see* $pf[j]$ *in the definition. And this requires some explanation.*

3) *symbolic case generalized by 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 by 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* $pf(i) = \mathsf{symbCase}(LRS_i, LI)$*.*

JP: In general, it is always good to give auxiliary definitions and intuitions first, then the formal definitions. Otherwise, the definitions are given first and very hard to understand.

$\mathsf{symbCmp}(LR, LI, i, j)$ defines a symbolic formula generalized by comparing $LR[i]$ and $LI[j]$; $\mathsf{symbCaseI}(LR, LI, i)$ a symbolic formula summarizing the results of comparison between $LR[i]$ and all $LI[j]$; $\mathsf{symbCase}(LR, LI)$ a symbolic formula representing a subcase generalized by comparing all $LR[i]$ and all $LI[j]$; $\mathsf{partition}(LRS, LI)$ is a disjunction of subcases $\mathsf{symbCase}(LRS_{[i]}, LI)$. JP: The above sentences are not helping. Probably, we shall move them before the formal definition, precisely explain why we have such cases/definitions. Are these definitions actually used in later parts of the paper?

Recall the first three lines in Table 1, and $LI = [1, 2]$ is the list of parameters occurring in $\mathsf{mutualInv}(1, 2)$; and LR is the actual parameter list to instantiate crit. JP: What is the precise values of LR?

- when $LR = [1]$, $\mathsf{symbCmp}(LR, LI, 1, 1) = (\mathtt{iR}_1 = iInv_1)$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 1) = (\mathtt{iR}_1 = iInv_1)$ as $LR[1] = LI[1]$.
- when $LR = [2]$, $\mathsf{symbCmp}(LR, LI, 1, 2) = (\mathtt{iR}_1 = iInv_2)$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 2) = (iR_1 = iInv_2)$ as $LR[1] = LI[2]$.
- when $LR = [3]$, $\mathsf{symbCmp}(LR, LI, 1, 1) = (\mathtt{iR}_1 \neq iInv_1)$, $\mathsf{symbCmp}(LR, LI, 1, 2) = (\mathtt{iR}_1 \neq iInv_2)$, $\mathsf{symbCase}(LR, LI) = \mathsf{symbCaseI}(LR, LI, 1) = (\mathtt{iR}_1 \neq iInv_1) \wedge (\mathtt{iR}_1 \neq iInv_2)$, because neither $LR[1] = LI[1]$ nor $LR[1] = LI[2]$.
- let $LRS = [[1], [2], [3]]$, $\mathsf{partition}(LRS, LI) = (\mathtt{iR}_1 = iInv_1) \vee (\mathtt{iR}_1 = iInv_2) \vee ((\mathtt{iR}_1 \neq iInv_1) \wedge (\mathtt{iR}_1 \neq iInv_2))$

JP: The example is good, but it does not generalise. The main idea of this generalisation should be given first and more precisely.

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 $iInv_i$ with $LI[i]$, and $iR[j]$ with $LR[j]$, the result is semantically equivalent to true. JP: I am confused by the above sentence. Don't know how to revise.

The second phase of generalization of concrete causal relations is to generalize the formula inv' accompanied with a causal relation CR_3 in a line of Table 1. JP: We shall give the table a specific name, and not mention Table 1 all the time. Table 1 is one concrete example. An index occurring in f' can either occur in the invariant formula, or in the rule. We need to check the situations to determine the transformation.

JP: I have no idea about the definition below. It requires a clear revision.

Definition 6 *Let* LI *and* LR *are two permutations,* $\mathsf{find-first}(L, i)$ *returns the least index* j *such that* $L[i] = j$ *if there exists such an index; otherwise it returns an error.*
$\mathsf{lookup}(LI, LR, i) \equiv$

$$\begin{array}{ll} \mathtt{iInv}_{\mathsf{find-first}(\mathtt{LI},\mathtt{i})} & \textit{if } i \in LI \\ \mathtt{iR}_{\mathsf{find-first}(\mathtt{LR},\mathtt{i})} & \textit{otherwise} \end{array}$$

$\mathsf{lookup}(LI, LR, i)$ returns the symbolic index transformed from i according to whether i occurs in LI or in LR. The index i will be transformed into $\mathtt{iInv}_{\mathsf{find-first}(\mathtt{LI},\mathtt{i})}$ if i occurs in LI, and $\mathtt{iR}_{\mathsf{find-first}(\mathtt{LR},\mathtt{i})}$ otherwise. Employing the lookup strategy to transform a concrete index i in inv' to $\mathsf{lookup}(LI, LR, i)$, symbolize2f transforms inv' into a symbolic one which will be needed in a proof command for existence of the CR_3 relation in Isabelle. JP: symbolize2f is not mentioned in the definition.

7 PROOFGEN: AUTOMATIC PROOF GENERATION

JP: Norte: in genera for this section, I try to clean up the latex codes. As discussed before, this section in general is very hard to understand. It does not convey an idea on how automatically generate proofs. Although some examples are given, they don't give a general idea which can commonly be applied.

A formal model of a protocol in a theorem prover like Isabelle includes the definitions of constants and rules and invariants, lemmas, and proofs (readers can refer to [19] 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 parameterized rule and invariant formula based on the aforementioned generalization of lines of concrete causal relations. JP: Since the table storing the causal relations is so important, let us give it a proper name.

JP: Structure of this section: give a concrete proof in Isabelle with a clear structure, then for each part of the structure, explain how to automatically generate.

An example lemma $\mathtt{critVsinv}_1$ JP: Use a lemma which appeared in the previous section. Explain what exactly is the lemma. for the simple mutual exclusion protocol and its proof in Isabelle are presented as follows:

```
1 lemma 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 CRS 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 CR3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp,
    rule_tac x=¬(x=true ∧ a[iInv2]=C) in exI, auto)
    qed
8   then have CR s f r
    (invariants N) by auto}
9 moreover {assume b1:iR1=iInv2
10  have CR3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp,
    rule_tac x=¬ (x=true ∧ a[iInv1]=C in exI,auto)
    qed
11  then have CR s f r
    (invariants N) by auto}
12 moreover {assume b1:(iR1 ≠ iInv1 ∧ iR1 ≠ iInv2)
13  have CR2 s f r
    proof(cut_tac a1 a2 b1, auto)
    qed
14 then have CR2 s f r
    (invariants N) by auto}
15 ultimately show CRS s f r
    (invariants N) by blast
16  qed
```

JP: In the above proof, check carefully which causal relations are used.

In the above proof, Line 2 contains assumptions on the parameters of the invariant and protocol 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 exist 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 $\texttt{a1} : \texttt{iR1} \leq \texttt{N} \wedge \texttt{r} = \texttt{critiR1}$. From line 5, the remaining part is a typically readable Isar proof using calculation reasoning such as `moreover` and `ultimately` to perform 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 $\texttt{iR1} = \texttt{iInv1}$, Line 7 first proves that the causal relation CR_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 CRS holds; Lines 9-11 proves the case where $\texttt{iR1} = \texttt{iInv2}$, proof of which is similar to that of case 1; Lines 12-14 are for the case where neither $\texttt{iR1} = \texttt{iInv1}$ nor $\texttt{iR1} = \texttt{iInv2}$. Each proof of a sub case is done in a block `moreoverb1 : asm1proof1`, the `ultimately` proof command in Line 15 concludes by summing up all the sub cases.

With the help of all the lemmas such as ruleVsinv1, we can prove the following lemma lemma_inv_1_on_rules which specifies that for all $r \in rules\ N$, and f is a formula f which is generated by instantiating $inv1$ with some parameters $iInv_1$ and $iInv_2$, CRS $s\ f\ r\ (invariants\ N)$. JP: Make sure these are consistent with previous parts. I will check later.

```
lemma lemma_inv1_on_rules:
assumes a1: r ∈ rules N and
  a2: (∃ _iInv1 _iInv2. _iInv1≤   N∧_iInv2≤N∧
  iInv1≠iInv2∧f=inv1 iInv1 iInv2)
shows CRS s f r (invariants N)
proof -
  have (∃ i. i≤N∧r=try i)∨
  (∃ i. i≤N∧r=crit i)∨
  (∃ i. i≤N∧r=exit i)∨
  (∃ i. i≤N∧r=idle i)
apply (cut_tac a1, auto) done
moreover {assume b1: (∃ i. i≤N∧r=try i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis tryVsinv1) done}
moreover {assume a1: (∃ i. i≤ N∧r=crit i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis critVsinv1) done}
moreover {assume a1: (∃ i. i≤ N∧r=exit i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis exitVsinv1) done}
moreover {assume a1:(∃ i. i≤N∧r=idle i)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1, metis idleVsinv1) done}
ultimately show invHoldForRule' s f r
   (invariants N) by auto
qed
```

JP: While cleaning up the latex source codes, I feel that in the above proof there are many typos which I don't know how to fix.

With the help of all the lemmas such as lemma_inv$_i$_on_rules, we can prove the following lemma invs_on_rules which specifies that for all $f \in invariants\ N$ and $r \in rules\ N$, $invHoldForRule\ s\ f\ r\ (invariants\ N)$. JP: In this section, I feel that we are mixing Isabelle codes and our formalisations as described in previous sections. We need to discuss what to do. In the above paragraph, latex source codes should be cleaned up.

```
lemma invs_on_rules:
assumes a1: f ∈ invariants N and
  a2: r ∈ rules N
shows invHoldForRule' s f r (invariants N)
proof -
have b1: (∃ iInv1 iInv2. iInv1≤N∧
     iInv2≤N∧iInv1≠iInv2∧
  f=inv1 iInv1 iInv2)∨
  (∃ iInv2. iInv2≤N∧f=inv2 iInv2)∨
  (∃ iInv1 iInv2. iInv1≤N∧iInv2≤N∧iInv1≠iInv2∧
  f=inv3 iInv1 iInv2)∨
  (∃ iInv2. iInv2≤N∧f=inv4 iInv2)∨
  (∃ iInv1 iInv2. iInv1≤N∧iInv2≤N∧iInv1≠iInv2∧
  f=inv5 iInv1 iInv2)
apply (cut_tac a1, auto) done
moreover {assume b1:(∃ iInv1 iInv2. iInv1≤N∧
    iInv2≤ N∧iInv1≠iInv2∧f=inv1 iInv1 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1,
    metis lemma_inv1_on_rules) done}
moreover {assume b1:(∃ iInv2. iInv2≤ N
    ∧f=inv2 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1,
    metis lemma_inv2_on_rules) done}
moreover {assume b1:(∃ iInv1 iInv2. iInv1≤N∧
  iInv2≤ N∧iInv1≠iInv2∧f=inv3 iInv1 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1,
    metis lemma_inv3_on_rules) done}
moreover {assume b1:(∃ iInv2. iInv2≤N∧
  f=inv4 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1,
    metis lemma_inv4_on_rules) done}
moreover {assume b1:(∃ iInv1 iInv2. iInv1≤N∧
  iInv2≤ N∧iInv1≠iInv2∧f=inv5 iInv1 iInv2)
  have invHoldForRule' s f r (invariants N)
  apply (cut_tac a2 b1,
    metis lemma_inv5_on_rules) done}
ultimately show invHoldForRule' s f r
    (invariants N)
  apply fastforce done
qed
```

Lemmas on initial states. We first discuss the definition on the initial state of the protocol, and the lemmas specifying that each invariant formula holds at the initial state. A typical Isabelle definition on the initial state of the protocol is shown as follows:

```
definition initSpec0::nat ⇒ formula where [simp]:
initSpec0 N ≡ (forallForm (down N)
(% i. (eqn (IVar (Para (Ident "n") i)) (Const I))))
definition initSpec1::formula where [simp]:
initSpec1 ≡ (eqn (IVar (Ident ''x'')) (Const true))
definition allInitSpecs::nat ⇒ formula list
allInitSpecs N ≡ [(initSpec0 N),(initSpec1 )]
lemma iniImply_inv4:
assumes a1: (∃iInv1. iInv1≤N∧f=inv4 iInv1)
and a2: formEval (andList (allInitSpecs N)) s
shows formEval f s
using a1 a2 by auto
```

JP: The above Isabelle code fragment is very difficult to read. `initSpec0` and `initSpec1` specifies the assignments on each variable `a[i]` where `i` ≤ `N` and `x`. JP: What is x? The specifications of the initial state is the list of all the specification definition on related state variables. Lemma `iniImply_inv4` simply specifies that the invariant formula`inv4` holds at a state `s` which satisfies the conjunction of the specification of the initial state. Isabelle's `auto` method can solve this goal automatically. Other lemmas specifying that other invariant formulas hold at the initial state are similar.

With the lemmas such as `iniImply_inv4`, for any invariant $inv \in (\text{invariants } N)$, any state s, if ini is evaluated true at state s, then inv is evaluated true at state s.

```
lemma on_inis: assumes a1: f ∈ (invariants N) and
  a2: ini ∈ {andList (allInitSpecs N)} and
  a3: formEval ini s
shows formEval f s
proof -
have c1: (∃ iInv1 iInv2. iInv1≤N∧
  iInv2≤N∧iInv1≠iInv2∧
  f=inv__1 iInv1 iInv2)∨
  (∃ iInv2. iInv2≤N∧f=inv__2 iInv2)∨
  (∃ iInv1 iInv2. iInv1≤N∧iInv2≤N∧iInv1≠iInv2∧
  f=inv__3 iInv1 iInv2)∨
  (∃ iInv2. iInv2≤N∧f=inv__4 iInv2)∨
  (∃ iInv1 iInv2. iInv1≤N∧iInv2≤N∧
  iInv1≠iInv2∧f=inv__5 iInv1 iInv2)
apply (cut_tac a1, simp) done
moreover {assume b1:(∃ iInv1 iInv2. iInv1≤N∧
    iInv2≤N∧iInv1≠iInv2∧f=inv__1 iInv1 iInv2)
  have formEval f s
  apply (rule iniImply_inv__1)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done}
moreover {assume b1:(∃ iInv2. iInv2≤N∧
    f=inv__2 iInv2)
  have formEval f s
  apply (rule iniImply_inv__2)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done}
moreover {assume b1:(∃ iInv1 iInv2. iInv1≤N∧
    iInv2≤N∧iInv1≠iInv2∧
    f=inv__3 iInv1 iInv2)
  have formEval f s
  apply (rule iniImply_inv__3)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done}
moreover {assume b1:(∃ iInv2. iInv2≤N∧
    f=inv__4 iInv2)
  have formEval f s
  apply (rule iniImply_inv__4)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done}
moreover {assume b1:(∃ iInv1 iInv2. iInv1≤N∧
    iInv2≤N∧iInv1≠iInv2∧
    f=inv__5 iInv1 iInv2)
  haveformEval f s
  apply (rule iniImply_inv__5)
  apply (cut_tac b1, assumption)
  apply (cut_tac a2 a3, blast) done}
ultimately show formEval f s by auto
qed
```

The proof structure of lemma_inv1_on_rules and invs_on_rules and on_inis are also typical case analysis ones using moreover blocks and ultimately commands. Therefore, a generic program of generating a typical case analysis proof will be adopted in our framework. JP: How?

The main theorem. With the preparation of lemma on_inis and invs_on_rules, JP: I don't know what exactly the first sentence means. the generation of the main lemma is quite easy. Recall that the consistency lemma is our main weapon to prove the main lemma, which requires proving two parts of obligations. JP: 'weapon'? 'obligations'?

1) For any invariant $inv \in$ (invariants N), any state s, if ini is evaluated true at state s, then inv is evaluated true at state s. This can be solved done by applying lemma on_inis.
2) For any invariant $inv \in$ (invariants N), any r in rule set rules N , one of the causal relations CR_{1-3} holds. This can be solved done by applying lemma invs_on_rules.

```
lemma main:
assumes a1: 0<N and
  a2: s∈ reachableSet {andList (allInitSpecs N)}
    (rules N)
shows ∀ inv. inv ∈ (invariants N) ⟶
  formEval inv s
proof (rule consistentLemma)
show consistent (invariants N)
    {andList (allInitSpecs N)}
    (rules N)
proof(cut_tac a1, unfold consistent_def,rule conjI)
show ∀inv ini s. inv ∈ (invariants N) ⟶
    ini ∈{andList (allInitSpecs N)} ⟶
    formEval ini s ⟶
    formEval inv s
proof((rule allI)+,(rule impI)+)
  fix inv ini s
  assume b1:inv ∈ (invariants N) and
    b2:formEval ini s and
    b3:ini ∈ {andList (allInitSpecs N)}
  show "formEval f s"
  apply (rule on_inis, cut_tac b1, assumption,
    cut_tac b2, assumption, cut_tac b3,
    assumption) done
qed

next show ∀inv r. inv ∈ invariants N⟶
    r ∈rules N ⟶
    invHoldForRule inv r (invariants N)
proof((rule allI)+,(rule impI)+)
  fix f r
  assume b1: f ∈ invariants N and
    b2:r ∈ rules N
  show invHoldForRule' s f r (invariants N)
  apply (rule invs_on_rules, cut_tac b1,
    assumption, cut_tac b2, assumption) done
qed

next show s ∈ reachableSet
    andList (allInitSpecs N) (rules N)
  apply (metis a1) done
qed
```

The generation of the main lemma is quite easy because it is in a standard form. JP: Really? And for the rest of this section, I just clean as much as I can without understanding the algorytms.

Algorithms of The Proof Generator proofGen. In the following, we illustrate the key techniques and algorithms of generation of the lemmas and their proofs. According to the order in which we present the above lemmas, we introduce their proof generation in a bottom-up manner. First let us introduce the generation of a subproof according to a relation tag of CR_{1-3}, which is shown in Algorithm 11.

In the body of function rel2proof, $sprintf$ writes a formatted data to string and returns it. In line 10, $getFormField(relTag)$ returns f' if $relTag = invHoldForRule_3(f')$. rel2proof transforms a a relation tag into a paragraph of proof. If the tag is among CR_{1-2}, the transformation is rather straight-forward, otherwise the

Algorithm 4: Generating a kind of proof which is according with a relation tag of CR_{1-3}: rel2proof

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

1 **if** $relTag = \mathsf{CR}_1$ **then**
2 $proof \leftarrow$ 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** $relTag = \mathsf{CR}_2$ **then**
7 $proof \leftarrow$ 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 $f' \leftarrow$ getFormField$(relTag)$;
12 $proof \leftarrow$ 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*

Algorithm 5: Generating one sub-proof for a subcase: oneMoreOverGen

Input: A formula $caseFsm$ standing for the assumption of the subcase, a relation item $relItem$ containing the information of causal relation
Output: An Isablle proof: $subProof$

1 $proof \leftarrow$ rel2proof$(relItem)$;
2 $subProof \leftarrow$ sprintf
3 "moreover{assume b1:%s
4 %s } "
5 (asm, proof);
6 **return** *subproof*

form f' is assigned by the formula $getFormField(relTag)$, and provided to tell Isabelle the formula which should be used to construct the CR_3 relation.

In Algorithm 5, oneMoreOverGen generates a subproof for a subcase in a proof of case analysis. It returns a subproof which is composed by filling an assumption of the subcase such as "iR1=iInv1" and a paragraph of proof generated by rel2proof$(relItem)$ into a format of block `morover` { ... }.

Due to the common use of case analysis proof of using moreover and ultimately commands, we design a generic program for performing case analysis doCaseAnalz. In algorithm 6, formulas standing for case-splitting $partition$, subproofs $subproofs$, and the conclusion $concluding$ are needed in case analysis to fill the format.

In Algorithm 7, caseAnalzI generates a typical proof of doing case analysis to prove some causal

Algorithm 6: Generating a whole proof of doing case analysis: doCaseAnalz

Input: A formula $partition$ standing for case-splittings, a proof list $subproofs$ standing all the subproofs of each subcases, concluding parts $concluding$
Output: An Isabelle proof: $proof$

```
1 proof ←sprintf
2    " have %s by auto
3    %s
4    ultimately show %s by auto"
5  (partition, subproofs, concluding) ;
6 return proof
```

Algorithm 7: Generating a whole proof of doing case analysis on parameters of rule and invariant: caseAnalzI

Input: A record rec fetched from $symbCausal$
Output: An Isablle proof: $proof$

```
1  cases ← caseField(rec);
2  rels ← relItems(rec); partition ← ⋁ cases;
3  subproofs ← "";
4  while (cases ≠ []) do
5      case ← head(cases) ;
6      cases ← tail(cases) ;
7      rel ← head(rels) ;
8      rels ← tail(rels) ;
9      subproofs ←
       subproofs^oneMoreOverGenI(case, rel);
10 concluding ←"invHoldForRule s f r (invariants N) ";
11 proof ←
   doCaseAnalz(partition, subproofs, concluding);
12 return proof
```

Algorithm 8: Generating an assumption on an invariant formula: asmGenOnInv

Input: An invariant name $invName$, a table $symInvs$ storing invariant formulas
Output: An assumption on an invariant formula: asm

```
1 invItem ← tbl_element(symbInvs, invName);
2 lenPInv ← invParaNum(invItem);
3 invParas ← invParasGen(lenPInv);
4 constrOnInv ←
  symbForm2Isabelle(constrOfInv(invItem));
5 if lenPInv = 0 then
6     asm ← "a1 : f = "^invName;
7 else
8     asm ← sprintf "a1: ∃ %s. %s ∧ f=%s %s"
      (invParas, constrOnInv, invName, invParas);
9 return asm
```

Algorithm 9: Generating an obtain proof command on an invariant formula: obtainGenOnInv

Input: An invariant name $invName$, a table $symInvs$ storing rules

```
1 invItem ← tbl_element(symbInvs, invName);
2 lenPInv ← invParaNum(invItem);
3 invParas ← invParasGen(lenPInv);
4 if lenPInv = 0 then
5     obtain ← "";
6 else
7     obtain ← sprintf "from a1 obtain %s where a1:%s
      ∧ f=%s %s by auto"
8       (invParas, constrOnInv, invName, invParas);
9 return obtain
```

relation hold between some rule and invariant. oneMoreOverGenI$(case, rel)$ formula comes from the disjunction of formulas in the `symbCases` field of rec, which is returned by $caseField(rec)$, subproofs $subproofs$ are generated by concatenation of all the subproofs, each of which is generated by oneMoreOverGenI$(case, rel)$. The proof is simply composed by calling doCaseAnalz$(partition, subproofs, concluding)$.

Next we discuss how to generate assumptions on an invariant formula of an lemma such as $critVsInv1$. In the body of Algorithm 8, tbl_element$(symbInvs, invName)$ retrieves the record on a invariant formula from $symbInvs$ to $invItem$ by its name $invName$, invParaNum$(invItem)$ and constrOfInv$(invItem)$) return the field `invNumFld` and `constr` of $invItem$, respectively. invParasGen$(lenPInv)$ generates a string of a list of actual parameters such as $iInv_1 \ldots iInv_{lenPInv}$ if $lenPInv > 0$, else an empty string "". At last, the assumption on the invariant is created by filling $invParas$, $constrOnInv$, and $invName$ into a proper place in the format if needed.

Similar to asmGenOnInv, obtainGenOnInv, which is shown in Algorithm 9, generates a proof command of obtain by retrieving and generating the related information and filling them in a format on obtain. Similar to asmGenOnInv and obtainGenOnInv, asmGenOnRule and obtainGenOnRule generate an assumption and obtain proof command on a rule.

After the above functions, now the generation of a lemma on the causal relation is rather easy, which is shown in Algorithm 10. After generating an assumption on invariant formula $asm1$, $asm2$ on a rule, an obtain command $obtain1$ on the invariant, and $obtain2$ on the rule, $symRelItem$ is retrieved from $symCausalTab$ by $ruleName^{\wedge}invName$, and a proof $proof$ is generated by calling caseAnalzI$(symRelItem)$. At last these parts are filled into proper places in the lemma format.

Due to page 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 CR_{1-3}, which is shown in Algorithm 11. Input $relTag$ is the result of the generalization step, which is discussed in Section 6.

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 $relTag =$ 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 invHoldRule$_{1-2}$, the transformation is rather straight-forward, else the form f' is

Algorithm 10: Generating a lemma on a causal relation: lemmaOnCausalRuleInv

Input: A parameterized rule name $ruleName$, a formula name $invName$, a table $symRules$ storing rules, a table $symInvs$ storing invariant formulas, a table $symCausalTab$ storing causal relation
Output: An Isablle proof script for a lemma: $lemmaWithProof$

```
asm1 ← asmGenOnInv(symbInvs, invName);
asm2 ← asmGenOnRule(symbRules, ruleName);
obtain1 ← obtainGenOnInv(symbInvs, invName);
obtain2 ← obtainGenOnRule(symbRules, ruleName);
symRelItem ←
tbl_element(symCausalTab, (ruleName^ invName));
proof ← caseAnalzI(symRelItem);
lemmaWithProof ← sprintf
 "lemma %sVs%s:
 assumes %s and %s
 shows invHoldForRule s f r (invariants N)
 proof - %s %s %s qed"
 (ruleName, invName, asm1, asm2,
   obtain1, obtain2, proof) ;
return lemmaWithProof
```

Algorithm 11: Generating a kind of proof which is according with a relation tag of CR_{1-3}: rel2proof

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

```
if relTag = CR1 then
    proof ← sprintf
      "have invHoldRule1 f r (invariants N)
      by(cut_tac a1 a2 b1, simp, auto)
      then have invHoldRule f r (invariants N) by
    blast" ;
else if relTag = CR2 then
    proof ← sprintf
      "have invHoldRule2 f r (invariants N)
    by(cut_tac a1 a2 b1, simp, auto)
      then have invHoldRule f r (invariants N) by
    blast" ;
else
    f' ← getFormField(relTag);
    proof ← sprintf
      "have invHoldRule3 f r (invariants N)
      proof(cut_tac a1 a2 b1, simp, rule_tac x=%s in
    exI,auto)qed
      then have invHoldRule f r (invariants N) by
    blast" (symbf2Isabelle f')";
return proof
```

assigned by the formula getFormField(relTag), and provided to tell Isabelle the formula which is used to construct the CR_3 relation.

JP: In the end, I feel that it might be better to explain everything 'top-down'. In many of the presented algorithms, there are many functions are not explained well. Maybe the whole idea of proof generation is simple (as syntax transformation), but this whole section is quite messy, in a way that it mixes information texts and formalisations that we try to achieve in previous sections.

TABLE 2
Verification results on benchmarks.

Protocols	#rules	#invs	time (s)	Mem. (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

8 Application to FLASH

9 Other Examples

We have implemented our approach in a prototype ParaVerifier [18]. We have also checked a few other examples, including the 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 at the website of our tool [19]. Each experiment data includes the paraVerifier instance, invariant sets, Isabelle proof scripts. Experiment results are summarized in Table 2.

10 Conclusion and Future Work

JP: In general, we need to argue why our approach can work. Especially, why it can deal with large examples and the other existing methods cannot.

The originality of our approach to automatically verifying parameterized cache coherence protocols 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 based on the information of causal relation computed in a small protocol instance. JP: The above description is not consistent with the whole approach. It will be revised later. Contributions should also contain the implementation and the case studies of large protocols.

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 and generalization can release the burden of human interaction. JP: In the future, we will extend our work to deal with general forms of safety properties (not only focusing on invariant properties) and liveness properties which also play an important role in realistic protocols to guarantee that good things will eventually happen.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT PressPress, 1999.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT PressPress, 2008.

[3] S. Park and D. L. Dill, "Verification of FLASH cache coherence protocol by aggregation of distributed transactions," in *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1996, pp. 288–296.

[4] A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in *Proc. 8th International Conference on Computer Aided Verification*, ser. LNCS, vol. 1102. Springer, 1996, pp. 184–195.

[5] N. Björner, A. Browne, and Z. Manna, "Automatic generation of invariants and intermediate assertions," *Theoretical Computer Science*, vol. 173, no. 1, pp. 49 – 87, 1997.

[6] T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions," in *Proc. 13th International Conference on Computer Aided Verification*, ser. LNCS, vol. 2102. Springer, 2001, pp. 221–234.

[7] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2031. Springer, 2001, pp. 82–97.

[8] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2031. Springer, 2001, pp. 113–127.

[9] K. Baukus, Y. Lakhnech, and K. Stahl, "Parameterized verification of a cache coherence protocol: Safety and liveness," in *Proc. 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation*, ser. LNCS, vol. 2294. Springer, 2002, pp. 317–330.

[10] C.-T. Chou, P. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proc. 5th International Conference on Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 3312. Springer, 2004, pp. 382–398.

[11] S. Pandav, K. Slind, and G. Gopalakrishnan, "Counterexample guided invariant discovery for parameterized cache coherence verification," in *Proc. 13th IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, ser. LNCS, vol. 3725. Springer, 2005, pp. 317–331.

[12] Y. Lv, H. Lin, and H. Pan, "Computing invariants for parameter abstraction," in *Proc. the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE CS, 2007, pp. 29–38.

[13] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi, "Cubicle: A parallel smt-based model checker for parameterized systems," in *Proc. 24th International Conference on Computer Aided Verification*, ser. LNCS, vol. 7358. Springer, 2012, pp. 718–724.

[14] ——, "Invariants for finite instances and beyond," in *Proc. 13th International Conference on Formal Methods in Computer-Aided Design*. IEEE CS, 2013, pp. 61–68.

[15] M. Dooley and F. Somenzi, "Proving parameterized systems safe by generalizing clausal proofs of small instances," in *Proc. 28th International Conference on Computer Aided Verification*, ser. LNCS, vol. 9779. Springer, 2016, pp. 292–309.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford FLASH multiprocessor," in *Proc. 21st Annual International Symposium on Computer Architecture*. ACM, 1994, pp. 302–313.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. LNCS 2283. Springer, 2002.

[18] Y. Li, J. Pang, Y. Lv, D. Fan, S. Cao, , and K. Duan, "paraVerifier: An automatic framework for proving parameterized cache coherence protocols," in *Proc. 13th International Symposium on Automated Technology for Verification and Analysis*, ser. LNCS, vol. 9364. Springer, 2015, pp. 207–213.

[19] ——, "The software tool paraverifier," 2016, available at https://github.com/paraVerifier/.