A Novel Approach to Parameterized Verification

Yongjian Li Kaiqiang Duan Shaowei Cai Yi Lv

***Abstract*—Parameterized verification of parameterized protocols like cache coherence protocols is important but hard. Our tool paraVerifier handles this hard problem in a unified framework: (1) it automatically discovers auxiliary invariants and the corresponding causal relations from a small reference instance of the verified protocol; (2) the above invariants and causal relation information are automatically generalized into a parameterized form to construct a parameterized formal proof in a theorem prover (e.g., Isabelle). The principle underlying the generalization is the symmetry mapping. Our method is successfully applied to typical benchmarks including snoopy-based and directory-based benchmarks. Another novel feature of our method lies in that the final verification result of a protocol is provided by a formal and readable proof.**

I. Introduction

Verification of parameterized concurrent systems is interesting in the area of formal methods, mainly due to the practical importance of such systems. Parameterized systems exist in many important application areas, including cache coherence, security, and network communication protocols. The hardness of parameterized verification is mainly due to the requirement of correctness that the desired properties should hold in any instance of the parameterized system. The model checkers, although powerful in verification of non-parameterized systems, become impractical to verify parameterized systems, as they can verify only an instance of the parameterized system in each execution. A desirable approach is to provide a proof that the correctness holds for any instance.

Related Work: There have been a lot of studies in the field of parameterized verification [1], [2], [3], [4], [5], [6], [7], [8], [9]. Among them, the 'invisible invariants' method [3] is an automatic technique for parameterized verification. In this method, auxiliary invariants are computed in a finite system instance to aid inductive invariant checking. Combining parameter abstraction and guard strengthening with the idea of computing invariants in a finite instance, Lv et al. [8] 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 works [3], [8] 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 (an variant of BDD_based SMV model checker). They are too raw to have an intuitive meanings. The capacity of the invisible invariant method is seriously limited when computing the reachable set of invisible invariants for the inductive checking is not feasible in the case of a large example like FLASH . Until now, the examples, which can be handled by the "invisible invariant" method, are quite small, we still can't find successful experiments on large examples like FLASH.

The CMP method, which adopts parameter abstraction and guard strengthening, is proposed in [6] for verifying a safety property inv of a parameterized system. An abstract instance of the parameterized protocol, which consists of m + 1 nodes $\{P_1, \ldots, P_m, P^*\}$ with 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 inv. 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 inv', then uses it to strengthen the guards of some transition rules of the abstract node. The 'strengthened' system is then subject to 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, no one has formally proved its correctness in a theorem prover. This situation may be not ideal because its application domain for cache coherence protocols which demands the highest assurance for correctness. Besides, the analysis of counter-example and generation of new auxiliary invariants usually depend on human's deep insightful understanding of the protocol. It is too laborious for people to do these analysis and some effective automatic tool is needed to help people.

Predicate abstraction is also applied to the verification of parameterized systems. Baukus, Lakhnech, and Stahl have used it to verify German (without data paths)[**?**], and Das, Dill, and Park have used it to verify FLASH[10]. The core 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 automatical. Furthermore, the abstracted system is needed to proved to be conservative for certain properties under verification. This proof also needs a set of auxiliary invariants. Therefore searching enough auxiliary invariants cann't be avoided. No further efforts are made to make automatical both the discovery of proper predicates and the searching of auxiliary invariants in the work of applying predicate abstraction to the parameterized verification.

Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaidi have made progress in searching automatically auxiliary invariants[13]. A heuristics-guided algorithm, called Barb, searches auxiliary invariants backward with the help of an oracle (a reference instance of the protocol). Roughly speaking, Barb's work can be seen as a backward reachability analysis. Barb is implemented in an SMT-based model checker Cubicle[**?**]. The correctness of Barb is argued in a generic symbolic framework. The searched auxiliary invariants are claimed to be inductive for deductive proof in a case study of a protocol. However, the formulation of Barb and the proof is not done in a theorem prover. Neither is a formal proof is given adopting the invariants for the protocol. Besides, the configuration of oracle need to be done manually.

The degree of rigorousness and automation are two critical aspects of approaches to parameterized verification. The verification of real-world parameterized systems is, however, rarely both rigorous and automatic. For instance, FLASH protocol is the cache coherence protocol of the Stanford FLASH mutlitprocessor [**?**]. This protocol is so complex that only a few approaches [10], [11], [6], [13] have successfully verified it so far. Furthermore, all existing successful verification approaches have their downsides. [10] is a theorem proving based approach which requires to construct inductive invariants by hand. The cases of [11] and [6] are similar to [10] that hand-crafted invariants are required to provide by human experts. As a contrast, [13] is a model checking based approach which can be carried out automatically. However, the formal proof can not be obtained from the work of [13]. In order to effectively verify complex parameterized protocols like FLASH protocol, there are two issues need to be addressed. The first one is how to find a set of sufficient and necessary invariants without (or with less) human intervention, which is a core research topic in this field. The second one is the rigorousness of the verification. It is preferable to formulate all the verification in a publicly-recognized trust-worthy framework like a theorem prover [6].

It is not difficult to formalize the model and properties of a protocol in a theorem prover like Isabelle, however, it is too hard to construct a proof to prove the properties. Because the most creative choices in a formal proof are done by human and hard to be automated. These choices mainly lie in: (1) the induction scheme; (2) the case analysis: different subproofs are done in different cases. (3) the quantifier instantiations. Up to now, the main efforts are made in searching invariants automatically. Few people have considered how to link the invariant searching with proof checking. The invariants found automatically from a concrete instance of the protocol in [3], [13], [8] without consideration how to use them in a theorem proving, thus, it is still hard to automate the above three kinds of intelligent choices. In detail, the invariants and rules are usually in concrete form in the procedure of invariant searching, but not in a parameterized form which are required in theorem proving. The generalization from a result obtained in the concrete protocol instance to that in the parameterized instance is not fully considered. Therefore, much human intervention is still needed to prove that the properties are invariance. Usually the complexity of the proof is beyond human's power. This is the reason why a theorem prover is still seldom used in parameterized verification.

In order to solve the parameterized verification in a both automatical and rigorous way, we must consider the invariant searching with proof checking in a unified framework. The key ideas are to make the aforementioned generalization automatically and to make the aforementioned creative choices in theorem proving to be automated. In detail,

1) We propose a special induction scheme for parameterized verification. Three kinds of causal relations among a formula and a rule and a set of formulas are introduced, which are essentially special cases of the general induction rule. Notably, with proper case analysis on the comparing parameters of a parameterized rule and those of a parameterized form, the three special induction proof rules can be applied automatically in a theorem prover.
2) A so-called consistent relation among a protocol instance and a set of formulas is proposed basing on the above three causal relations, which is the cornerstone in our method. If such a consistent relation holds, then any formula in the formula set is an invariant for the protocol instance. Here the protocol instance can be either concrete or parameterized.
3) From an initially given invariant, our tool search both invariants and causal relations from a small concrete protocol instance which can construct a consistent relation between the protocol instance and the set of all found invariants. Notice that both invariants and causal relations, which are searched in this phase, are concrete and stored in a table.
4) Basing on the analysis and generalization by comparing the parameters of a concrete rule and those of a concrete invariant occurring in a line of the table, we generalize the line into a symbolic form. Namely, the invariants and rules will be generalized into symbolic forms, and a symbolic formula is generated basing on the above analysis of concrete parameters, and used to indicate the case condition in which the comparison should be satisfied between the symbolic parameters of the symbolic invariants and rules. Thus, the information on the splitting cases decided by comparison between the two symbolic parameters of a parameterized rule and a parameterized invariant formula is given in the generalized table. The choice of the three special induction proof rules in each case is also given in a line.
5) From the table in 4, a formal proof script in a theorem prover (e.g., Isabelle) can be generated to prove that a consistent relation also holds between the parameterized protocol instance and the set of the parameterized invariant formulas. Notably, because the proof script has enough proof commands to do induction, and case analysis and necessary quantifier instantiation, the proof script can be automatically checked once it is fed into the theorem prover.

Basing on the above ideals, We design a tool called paraVerifier, which is shown as below:

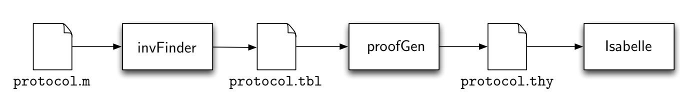

Fig. 1. The workflow of paraVerifier

Our tool paraVerifier is composed of two parts: an invariant finder `invFinder` and a proof generator `proofGen`. Given a protocol $\mathcal{P}$ and a property inv, `invFinder` tries to find useful auxiliary invariants and causal relations which are capable of proving inv. To construct auxiliary invariants and causal relations, we employ heuristics inspired by consistency relation. Also, when several candidate invariants are obtained using the heuristics, we use oracles such as a model checker and an SMT-solver to check each of them under a small reference model of $\mathcal{P}$, and chooses the one that has been verified.

After `invFinder` finds the auxiliary invariants and causal relations, `proofGen` generalizes them into 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. The generated proof is checked automatically.

The organization of this work is as follows: Section II introduces the preliminaries; Section III introduces the theoretical foundation; Section V the invFinder; Section VI the generalization strategy; Section VII the proofGen and the generated proof. We go through these sections by verifying a small example - mutual exclusion example. Section VIII shows the further experiments on real-world protocols. Section IX concludes our work.

II. Preliminaries

In this section, we introduce the theoretical foundation underlining paraVerifier. Consider a set of state variables V, we use e, f and S to denote an expression, a formula, and a statement over the set of state variables V. Variables are divided into two classes: array variables or non-array (global) variables. Basing on the variables, first order expressions and formulas can be defined as usual. We also assume that the variables in V range over a finite set D. 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 write $\mathbb{A}[e, s]$ (and $s \models f$) to denote the evaluation result of the expression and (and formula f is evaluated to be true) at the state s. With a parallel assignment $S = \{x_i := e_i | i > 0\}$, we define the notion of the weakest precondition $\mathsf{preCond}(S, f) \equiv f[x_i := e_i]$, which substitutes each occurrence of x_i by e_i.

Protocols. A cache coherence protocol is formalized as a pair (I, R), where (1) I is an initialization formula; and (2) R is a set of transition rules. Each rule $r \in R$ is defined as $g \triangleright S$, where g is a predicate, and S is a parallel assignment to distinct variables v_i with expressions e_i. We write $\mathsf{pre}(r) = g$, and $\mathsf{act}(r) = S$ if $r = g \triangleright S$. A state transition is caused by trigger and execution of a rule, formally, we define: $s \xrightarrow{r} s' \equiv s \models \mathsf{pre}(r) \wedge (\forall x \in \mathsf{vars}(\mathsf{act}(r)). s'(x) = \mathbb{A}[e, s] \wedge (\forall x \notin \mathsf{vars}(\mathsf{act}(r)). s'(x) = s(x))$.

Reachable state sets. 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 $s_0 \xrightarrow{r} s$.

Parameterized formulas, statemets, rules, and protocols For simplicity, a A parameterized formula(statement, rule, and protocol) is a function $f(x_1, x_2, ..., x_n)$ from a tuple of natural numbers to such an object. Without losing the generality, we require that parameters to instantiate a parameterized object are disjoint. For instance, $\mathsf{mutualInv}(i, j) \equiv \neg(n[i] = C \wedge n[j] = C)$ is a parameterized formula with two disjoint parameters. A list of natural numbers which are different from each other are actual parameters to instantiate a parameterized object. Thus, we need some functions on lists, $x\#xs$ for the list that extends xs by adding x to the front of xs, $[x_1, ..x_n]$ for a list $x_1\#..x_n\#[]$, $xs@ys$ for the result list by concatenating xs with ys, $xs[i]$ for the i^{th} element of the list xs (counting from 1 as the first element), set xs for the set of all the elements in xs, $|xs|$ for the length of the list xs, $\mathsf{hd}(xs)$ for the head element of a non-empty list, $\mathsf{tl}(xs)$ for the tail of xs but the head element.

Now we use a simple example to illustrate the above definitions by a simple mutual exclusion protocol with N nodes. Let I(dle), T(rying), C(ritical), and E(xiting) be enumerating values to indicate the state of a node, x, n are simple and array variables, N a natural number. x is a flag to indicate ? $\mathsf{pini}(N)$ the predicate to specify the inial state, prules(N) the four rules of the protocol, $\mathsf{mutualInv}(i, j)$ a property that $n[i]$ and $n[j]$ cannot be in a critical state at the same time. We want to verify that $\mathsf{mutualInv}(i, j)$ holds at any reachable state of the parameterized protocol $mutualEx(N)$ for any $i \leq N$, $j \leq N$ s.t. $i \neq j$.

Example 1 *Mutual-exclusion example.*

$$\texttt{pini(N)} \equiv \texttt{x=true} \wedge \bigwedge_{i=1}^{N} \texttt{n[i]=I}$$

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

III. Causal Relations and Consistency Lemma

A novel feature of our work lies in that three kinds of causal relations are exploited, which 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:*

1) $\mathsf{invHoldRule}_1(s, f, r) \equiv s \models \mathsf{pre}(r) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;
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.* $s \models (f' \wedge (\mathsf{pre}(r)) \longrightarrow s \models \mathsf{preCond}(f, \mathsf{act}(r))$;

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 $\mathsf{preCond}(S, f)$.

In Hoare logic, a Hoare triple is of the form $\{f\}S\{f'\}$ where f and f' are assertions of formulas and S is a statement. f is named 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:

1) $\mathsf{invHoldRule}_1(s, f, r)$ if and only if $\{\mathsf{pre}(r)\}\mathsf{act}(r)\{f\}$
2) $\mathsf{invHoldRule}_2(s, f, r)$ if and only if $\{\mathsf{pre}(r) \wedge f\}\mathsf{act}(r)\{f\}$
3) $\mathsf{invHoldRule}_3(s, f, r, fs)$ if and only if $\exists f_0 . f' \in fs \wedge (\{\mathsf{pre}(r) \wedge f_0\}\mathsf{act}(r)\{f\})$

$\mathsf{invHoldRule}(s, f, r, fs)$ can be regarded as a kind of general inductive tactics. That is is to say, a property f in fs holds at a state s, and $\mathsf{invHoldRule}(s, f, r, fs)$, then f holds at the post-state s' after a rule r is executed.

Lemma 1 *Let s and s' be two states and r be a rule s.t. $s \xrightarrow{r} s'$, if $s \models f$ and $\mathsf{invHoldRule}(s, f, r, fs)$ for any $f \in fs$, then for any $f \in fs$, $s' \models f$.*

Instead of using the general induction rule (or proving the general causal relation $\mathsf{invHoldRule}(s, f, r, fs)$), we classify it into three special kinds of causal relations $\mathsf{invHoldRule} - 1 - 3(s, f, r, fs)$ because the latter are more-fine grained and easy to be done by a theorem prover. In fact, because each one of fs holds at s, they can be regarded as induction hypothesis, $\mathsf{invHoldRule} - 1 - 3(s, f, r, fs)$ has told the theorem prover how to use the premise in the induction hypothesis to prove the precondition $preCond(f, \mathsf{act}(r))$. We will illustrate this later.

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

Definition 2 *A relation $\mathsf{consistent}(invs, inis, rs)$ holds if the following conditions hold:*

1) *for any formula $inv \in invs$ and $ini \in inis$ and any state s, $s \models ini$ implies $s \models inv$;*
2) *for any formula $inv \in invs$ and rule $r \in rs$ and any state s, $\mathsf{invHoldRule}(s, inv, r, invs)$*

Let us use some examples to illustrate the above definitions. Next example gives a set of auxiliary invariants, in which the initially invariant $mutualInv$ is.

Example 2 *Let us define*

```
invOnXC(i) ≡ ¬(x ≐ true ∧ n[i]≐ C)
invOnXE(i) ≡¬ (x ≐ true∧ n[i] ≐ E)
aux1(i,j) ≡¬ ( n[i]≐ C∧n[j] ≐ E)
aux2 (i,j) ≡¬ ( n[i]≐ E∧n[j]≐ E)
pinvs(N)≡ {f. ∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧
iInv1 ≠ iInv2 ∧
f =mutualInv iInv1 iInv2)
∨(∃ iInv1. iInv1 ≤ N ∧ f =invOnXC iInv1)
∨(∃ iInv1. iInv1 ≤ N ∧ f= invOnXE iInv1)
∨(∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2
∧ f = aux1 iInv1 iInv2)
∨(∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2
∧ f = aux2 iInv1 iInv2) }.
```

In Example 2, $\mathsf{invOnXC}(i)(\mathsf{invOnXE}(i))$ specifies that the variable x will be set to be false once node i is in or exiting the critical section. $\mathsf{aux}_1(i, j)$ says that node i and j can not be in and exiting the critical section at the same time. $\mathsf{aux}_2(i, j)$ that node i and j can not exit critical section at the same time.

Example 3 illustrates the three kinds of causal relations (or inductive tactics).

Example 3 *Suppose that $inv = \mathsf{mutual}(i_1, i_2)$, $r = \mathsf{crit}(iR_1)$, $rs = \mathsf{pinvs}(N)$, and $i_1 \leq N$, $i_2 \leq N$, $i_1 \neq i_2$, and $iR_1 \leq N$.*

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

From the above discussion, we can conclude $\mathsf{invHoldRule}_3(s, inv, r, invs)$.

In example 3, $\mathsf{invHoldRule}_1(s, inv, r)$ and $\mathsf{invHoldRule}_2(s, inv, r)$ can be checked *automatically* by a theorem prover. $\mathsf{invHoldRule}_3(s, inv, r, invs)$ can also be checked *automatically* if the proper formula f' such as $\mathsf{invOnXC}(i_2)$ can be provided for the instantiation for the existence quantifier. We need notice that two things are needed to be done to guide a theorem prover to automatically assist us to check $\mathsf{invHoldRule}(s, inv, r, invs)$: (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 subcase. Therefore

We can check the consistent relation holds between the auxiliary invariant set example 1 and the protocol initial predicate and rules of the mutual exclusion protocol.

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

Proof: By unfolding the definition of consistency, we need to verify that parts (1) and (2) of the consistency relation hold. For (1), the proof is rather straightforward. We only do case analysis on the form of a formula f in $pinvs$, and check $pini(n)$ 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 $s \models n[i_2] = I$ if $s \models pini(N)$, thus $s \models inv$ holds. The other invariants can be proved similarly.

For (2), we do case analysis on the form of a formula f in $pinvs$, and then on the form of r in $prules(N)$, notice that both f and r are parameterized, then we do case analysis by comparing indices in f and r, we need show $\mathsf{invHoldRule}_{1-3}$ holds. Example 3 has shown a typical case where $inv = \mathsf{mutualInv}(iInv_1, iInv_2)$, and $r = \mathsf{crit}(iR)$, where $iR \leq N$, $iInv_1 \leq N$, and $iInv_2 \leq N$. ∎

Let us analyze the complexity of part (2) of the proof in Lemma 2. For one rule, we need to analyze three cases for each invariant inv in $\mathsf{mutualInv}$, aux_1, and aux_2, and two cases for the others. There are four rules, thus we need in total $4 \times (3 \times 3 + 2 \times 2) = 52$ cases. Note that the protocol is simple because it has only 4 rules. Let alone a moderate protocol such as German (15-rules) and FLASH with about 50 rules. This complexity illustrates the difficulty of parameterized verification of cache coherence protocols, which also accounts for the reason why there is seldom successful case study in applying a general theorem prover to verify even a moderate protocol such as German protocol.

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

Theorem 3 *If $P = (ini, rs)$, $\mathsf{consistent}(invs, ini, rs)$, and $s \in \mathsf{reachableSet}(P)$, then for all inv s.t. $inv \in invs$, $s \models inv$.*

Theorem 3 is our main tool to prove that any property f in a formula set $invs$ is an invariant for a protocol (ini, rs). It has eliminated the need of directly use of usual induction proof method. We only check the causal relation between f and $r \in rs$ by case analysing on f and r.

Now we apply the consistence lemma to prove that the mutual exclusion property holds for each reachable state of the mutual-exclusion protocol. In order to prove the mutual-exclusion property, we prove a more general result:

Lemma 4 *If $P = (pini(N), prules(N))$ is the protocol listed in example 1, $s \in \mathsf{reachableSet}(P)$, and $0 < N$, and $pinvs$ is the set of formulas in example 2, then for any inv s.t. $inv \in pinvs(N)$, $s \models inv$.*

Proof: By theorem3, we only need to check that $\mathsf{consistent}(pins(N), pini(N), prules(N))$ relation holds. This can be immediately obtained by lemma 2. ∎

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

1) We need to construct a set of auxiliary invariants $invs$ which contains f and satisfies $\mathsf{consistent}(invs, inis, rs)$. Constructing a set of auxiliary invariants is the central problem in the filed of parameter verification.
2) After applying theorem 3, we decompose the original problem of invariant checking into that of checking that some causal relation between some $f \in invs$ and $r \in rs$. Then we need three levels of case analysis: the first is on the form of f, the second is on and r, and the last is case analysis by comparing on the rule parameters in r and invariant parameters in f, which has been illustrated in the proof of Lemma 2 and Example 3, at last the choice among the three kinds of causal relations is needed in each subcase. How to generate enough information to construct a proof which consists of the above case analysis and choices is our second problem.

Notice that the two problems are w.r.t. a parameterized property and a parameterized protocol instance. Now we first turn to a simpler concrete property and a concrete protocol instance, and try to solve the two problems w.r.t. concrete ones. Here the concrete property is obtained by instantiating the parameters with concrete indices, and by fixing the parameterized size with concrete value. For instance, Namely, $\mathsf{mutualInv}(i, j)$ (¬ (n[i]= C ∧ n[j]= C) and $(pinit(3), prules(3))$ are concrete. 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.

IV. An Overview of Our Verification Strategy

Due to theorem 3, the original problem of invariant checking boils down to checking causal relation some causal relation between some $f \in invs$ and $r \in rs$, and different kinds of causal relation among hold in different cases which are split by parameter comparison scheme between rule parameters and formula parameters. Notice that the comparison scheme is only determined by the number of rule parameters and formula parameters. For instance, if the number of rule parameters and formula parameters are 1 and 2 respectively, the splitted cases are $iR_1 = i_1$, or $iR_1 = i_2$, and $iR_1 \neq i_1 \wedge iR_1 \neq i_2$.

In order to know which kind of causal relation hold, we can select a special instance of the generalized symbolic case, and compute the precondition of f w.r.t. $act(r)$, then test in a concrete protocol instance which kind of causal relation hold among f, r, and the current set of invariant formulas $invs$. For instance, we can instantiate both iR_1 and i_1 with 1, i_2 with 2 test the case $iR_1 = i_1$. If the one of causal relation $invHoldRule_{1-2}$ holds, then only the causal relation is recorded. Otherwise, we need consider two cases: (1) there is a formula in $invs$ to make $invHoldRule_{1-2}$ to hold; (2) there is not a formula in $invs$ $invHoldRule_{1-2}$ to hold. In the second case, we need construct a new invariant formula f' which makes the relation $\mathsf{invHoldRule}_3(f, r, invs \cup \{f'\})$ to hold. The coupling process of checking causal relation and generating new invariants is not finished until no new invariant formula can be found. The returned result is a table which records all the causal relation among the newly found invariant formulas and concrete rules and the set of auxiliary invariant formulas.

Basing on the table recording the information on causal relation, we generalize it into a parameterized version. Each line of table is associated with a symbolic formula to indicate a case by comparison between 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.

In short,

1) Due to symmetry, we can check in a concrete protocol instance the causal relation between a concrete rule and a concrete invariant formula whose concrete parameters.
2) For each cases, we select a special instance of the case to check.
3) Checking the causal relation is naturally coupled with finding new auxiliary invariants.
4) Generalization is needed to extend the searching results.

V. Searching Concrete Auxiliary Invariants with Causal Relations

Given a parameterized protocol P and a property set F containing concrete invariant formulas each which is an instantiation of a parameterized invariant formula we want to initially verify, invFinder in Algorithm 1 aims to find useful auxiliary invariants and causal relations which are capable of proving any element in F. For instance, let $P = (pini(N), prules(N))$ to be the mutual exclusion protocol listed in example 1, $f = \mathsf{mutualInv}(1, 2)$, and $F = \{f\}$.

A set A is used to store all the invariants found up to now, and 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 NULL. A queue $newInvs$ is used to store new invariants which have not been checked, and is initialized as F. 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 r and cf by some policy. For each parameter $para$ in $paras$, it is applied to instantiate 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 $\mathsf{apply}(r, para) = r(para_{[1]}, ..., para_{[|para|]})$. Then $\mathsf{coreFinder}(cr, cf, A)$ is called to check which kinds of causal relation exists between cr and f; if there is such one relation item, the relation item rel and a formula option $newInvOpt$ is returned; otherwise a run-time error occurs in $coreFinder$, which indicates no proof can be found. 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 NONE, 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.

Algorithm 1: Algorithm: $invFinder$

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

$A \leftarrow F$;
$tbl \leftarrow []$;
$newInvs \leftarrow F$;
while $newInvs \neq []$ **do**
    $cf \leftarrow \mathsf{hd}(newInvs)$; $newInvs \leftarrow \mathsf{tl}(newInvs)$;
    for $r \in R$ **do**
        $paras \leftarrow \mathsf{Policy}(r, cf)$;
        for $para \in paras$ **do**
            $cr \leftarrow \mathsf{apply}(r, para)$;
            $newInvOpt, rel \leftarrow \mathsf{coreFinder}(cr, cf, A)$;
            $tbl \leftarrow tbl@[< r, para, cf, rel >]$;
            if $newInvOpt \neq NONE$ **then**
                $newInv \leftarrow \mathsf{get}(newInvOpt)$;
                $newInvs \leftarrow newInvs@[newInv]$;
                $A \leftarrow A \cup \{newInv\}$;
return A, tbl;

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 $\mathsf{coreFinder}(cr, cf, A)$ returns the causal relation among cr and $cf$and A. They will be illustrated in Section V-A and V-B.

A. Parameter Generation Policy

Let cf to be a special instance of some parameterized formula f s.t. $cf = apply(f, idp(|apOfCForm(cf)|))$. Recall that the aim of our policy is to compute groups of rule parameters to instantiate a parameterized rule into a set of concrete rules. Combination 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. Furthermore, each of the subcases should be sampled, 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 $\{1, ..., (n+n')\}$ to instantiate r. Namely a n'-permutation of $n + n'$, $para$ will be regarded as a group of rule parameters to instantiate r. Furthermore, if $n' > 0$ and $i < n'$, and $para[i] > n$, then $para[i]$ will not equal to any actual parameters in cf. Thus we don't care which value $para[i]$ is if we can gurantee $para[i]$ 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 s.t. $j > n$. 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 need not use $[2, 4]$ to do so.

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

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

1) $normPara(para, n)[i] =$
$$\begin{cases} para[i] & \text{if } para[i] \leq n \quad (1)\\ n+1 & \text{otherwise} \quad (2)\end{cases}$$
2) $para \sim^n_m para' \equiv normPara(para, n) = normPara(para', n)$
3) $\mathsf{semiP}(m, n, S) \equiv (\forall para \in \mathsf{perms}^n_m \exists para' \in S. para \simeq^n_m para') \wedge (\forall para \in S. \forall para' \in S. para \neq para' \longrightarrow (para \not\simeq^n_m para'))$. *A set S is called a quotient of the set* perms^n_m *under the relation* $\simeq^n_m$ *if* $\mathsf{semiP}(m, n, S)$.

In definition 3, 1 defines a function $normPara(para, n)$ 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. Basing on the relation $\sim^n_m$, we can define a quotient of the set perms^n_m under the relation $\simeq^n_m$. 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.

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) *Let $n = 2$, $n' = 1$, $paras = \{[1], [2], [3]\}$, and $paras$ is the quotinent set of parameter groups to instantiate r;*
2) *Let $n = 2$, $n' = 2$, $paras = \{[1, 2], [1, 3], [2, 1], [2, 3], [3, 4]\}$, and $paras$ is the quotinent set of parameter groups to instantiate r;*

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.

Definition 4 *Let $para$ be a permutation, $0 < n$, we define:*

$\mathsf{caseGen}([], n, pos) = []$
$\mathsf{caseGen}(i\#paran, pos)$=
$$\begin{cases} (iR_{pos} = iInv_i) \wedge \mathsf{caseGen}(paran, pos+1) & \text{if } i \leq n \quad (3)\\ (\bigwedge_{j=1}^{n} iR_{pos} \neq iInv_j) \wedge \mathsf{caseGen}(paran, pos+1) & \text{otherwise} \quad (4)\end{cases}$$

Next example shows the meaning of caseGen.

Example 5 *Recall the example 4,*

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

In the end of this section, we give an algorithm to compute a quotinent set of perms^n_m.

Algorithm 2: Computing a quotient of perms^n_m: $cmpSemiperm$

```
Input: m, n
Output: A list of permutations L
L0 ← perms^n_m; L ← [];
while L0 ≠ [] do
    para ← hd(L0); L0 ← tl(L0);
    if ∀para' ∈ set(L). para' ≄^n_m para then
        L ← L@[para];
return L;
```

Algorithm 2 computes a quotient of perms^n_m. Firstly it set $L_0 = \mathsf{perms}^n_m$, then we fetch the head element of L_0 into L, and find whether there is an element $para'$ in L s.t. $para \simeq^n_m para'$. If yes, then $para$ will be discarded, else $para$ is inserted into L. This procedure is repeated until L is empty. $\mathsf{Policy}(cf, r)$ is simply $\mathsf{perms}^{n'}_{n+n'}$.

Let us return to the example of mutual-exclusion protocol, for the invariant $\mathsf{mutualInv}(1, 2)$, according to Policy, three groups of parameters $[1]$, $[2]$, $[3]$ are used to instantiate crit respectively, each of the instantiation results will be used to check which kind of causal relation exists between the derived concrete rule and $\mathsf{mutualInv}(1, 2)$.

B. Core Searching Algorithm

For a concrete cf and a concrete rule cr and a set of formulas of found invariants $invs$, Algorithm 3 checks the causal relation among cf and cr and $invs$, as well as find a new invariant if it is needed. The algorithm returns a formula option and a causal relation item between r and inv. A formula option value NONE indicates that no new invariant is found, while $\mathsf{SOME}(f)$ indicates a new auxiliary invariant f is searched.

In order to illustrate coreFinder, we also need introduce some functions on symmetry transformation to 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 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 $\pi(i)$ s.t. $\mathsf{symApp}(\pi, f) = f'$. 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 SMV, and calling 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. 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 $fOpt$, a new causal relation rel

```
1  g ← pre(cr);
2  cf' ← preCond(cf, act(cr));
3  if cf = cf' then
4      relItem ← (cr, cf, invRule_2, −);
5      return (NONE, relItem);
6  else if tautChk(g → cf') = true then
7      relItem ← (cr, cf, invRule_1, −);
8      return (NONE, relItem);
9  else
10     candidates ← subsets(decompose(dualNeg(cf') ∧ g));
11     newInv ← choose(chk, candidates);
12     relItem ← (cr, cf, invRule_3, newInv);
13     if isNew(newInv, invs) then
14         newInv ← normalize(newInv);
15         return (SOME(newInv), relItem);
16     else
17         return (NONE, relItem);
```

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

(1) If $cf = cf'$, meaning that statement $\mathsf{act}(cr)$ does not change cf, then no new invariant is created, and new causal relation item marked with tag $\mathsf{invHoldRule}_2$ is recorded between cr and cf, but at this moment there are no new invariants to be added; for instance, let $cr = \mathsf{crit}(3)$, $cf = \mathsf{mutualInv}(1, 2)$, thus $cf' = cf$, then a pair $(\mathsf{NONE}, (crit(3), inv, \mathsf{invHoldRule}_2, \_))$ will be returned.

(2) If tautChk verifies that $g \dashrightarrow cf'$ is a tautology, then no new invariant is created, and the new causal relation item marked with tag $\mathsf{invHoldRule}_1$ is recorded between cr and cf. For instance, let $cr = \mathsf{crit}(2)$, $cf = \mathsf{invOnXC}(1)$, $cf' = \neg(\mathsf{false} = \mathsf{true} \wedge n[1] = \mathsf{C})$, obviously, $g \dashrightarrow f'$ holds forever, thus a pair $(\mathsf{NONE}, (\mathsf{crit}(2), inv, \mathsf{invHoldRule}_1, \_))$ 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 $\mathsf{invHoldRule}_3$ to hold. The candidate set is $\mathsf{subsets}(\mathsf{decompose}(\mathsf{dualNeg}(cf') \wedge g))$, where $\mathsf{decompose}(f)$ 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 $\bigwedge_{i=1}^{N} f_i$ 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 be added into $newInvs$, and the new causal relation item marked with tag $\mathsf{invHoldRule}_3$ will be added into the causal relations. Here the meaning of $\mathsf{isNew}(newInv, invs)$ is that $newInv$ is not symmetric to any formula in $invs$. Let $invs = \emptyset$, $r = \mathsf{crit}(1)$, $inv = \mathsf{mutualInv}(1, 2)$, $cf' = \neg(true = true \wedge n[2] = C)$, from all the subsets of $\{n[1] = T, x = true, n[2] = C\}$, the coreFinder calls choose oracle to select the subset $\{x = true, n[2] = C\}$, combines all the item in this set, then constructs a new formula $f_0 = \neg(x = true \wedge n[2] = C)$. After normalization tomf_0, the resulting new invariant $newInv = \neg(x = true \wedge n[1] = C)$ and a relation item $(crit(1), \mathsf{invHoldRule}_3, f_0)$ are returned.

TABLE I
A FRAGMENT OF OUTPUT OF invFinder

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

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

VI. Generalization

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

There are two main kinds of generalization in our work: (1) generalization of a normalized invariant into a symbolic one. The resulting symbolic invariants are used to create definitions of invariant formulas in Isabelle. For instance, $\neg(\mathrm{x} \doteq \mathrm{true} \wedge \mathtt{n}[1] \doteq \mathrm{C})$ is generalized into $\neg(\mathrm{x} \doteq \mathrm{true} \wedge \mathtt{n}[\mathtt{iInv_1}] \doteq \mathrm{C})$. This kind of generalization is done with model constraints, which specify that any parameter index should be not greater than the instance size $\mathtt{N}$, and parameters to instantiate a parameterized rule (formula) should be different. (2) The generalization of concrete causal relations into parameterized causal relations in Isabelle, and will be used in proofs of the existence of causal relations in Isabelle.

Since the first kind of generalization is simple, we focus on the second kind of generalization, which consists of two phases. Firstly, groups of rule parameters such as [[1],[2],[3]] will be generalized into a list of symbolic formulas such as $[\mathtt{iR_1} = \mathtt{iInv_1}, \mathtt{iR_1} = \mathtt{iInv_2}, (\mathtt{iR_1} \neq \mathtt{iInv_1}) \wedge (\mathtt{iR_1} \neq \mathtt{iInv_2})]$[1] , 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 invHoldRule3 relation is also generalized by some special strategy.

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

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

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

1) *symbolic comparison condition generalized from comparing $LR_{[i]}$ and $LI_{[j]}$:*
$\mathsf{symbCmp}(LR, LI, i, j) \equiv$
$$\begin{cases} \mathtt{iR_i} = \mathtt{iInv_j} & \text{if } LR_{[i]} = LI_{[j]} \quad (5)\\ \mathtt{iR_i} \neq \mathtt{iInv_j} & \text{otherwise} \quad (6)\end{cases}$$
2) *symbolic comparison condition generalized from comparing $LR_{[i]}$ and with all $LI_{[j]}$:*
$\mathsf{symbCaseI}(LR, LI, i) \equiv$
$$\begin{cases} symbCmp(LR, LI, i, j) & \text{if } \exists! j. LR_{[i]} = LI_{[j]} \quad (7)\\ forallForm(|LI|, pf) & \text{otherwise} \quad (8)\end{cases}$$
where $pf(j) = \mathsf{symbCmp}(LR, LI, i, j)$, and $\exists! j.P$ is an qualifier meaning that there exists a unique j s.t. property P;
3) *symbolic case generalized from comparing LR with LI : $\mathsf{symbCase}(LR, LI) \equiv \mathsf{forallForm}(|LR|, pf)$, where $pf(i) = \mathsf{symbCaseI}(LR, LI, i)$;*
4) *symbolic partition generalized from comparing all $LRS_{[k]}$ with LI, where LRS is a list of permutations with the same length: $\mathsf{partition}(LRS, LI) \equiv \mathsf{existsForm}(|LRS|, pf)$, where $pf(i) = \mathsf{symbCase}(LRS_i, LI)$.*

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

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

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

The second phase of generalization of concrete causal relations is to generalize the formula inv' accompanied with a causal relation $\mathsf{invHoldRule}_3$ in a line of table I. An index occurring in f' can either occur in the invariant formula, or in the rule. We need to look it up to determine the transformation.

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

$$\mathsf{lookup}(LI, LR, i) \equiv \begin{cases} \mathtt{iInv}_{\mathtt{find\_first(LI,i)}} & \text{if } i \in LI \quad (9) \\ \mathtt{iR}_{\mathtt{find\_first(LR,i)}} & \text{otherwise} \quad (10) \end{cases}$$

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

VII. Automatical Generation of Isabelle Proof

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

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

```
1lemma critVsinv1:
2 assumes a1: ∃ iR1. iR1 ≤ N ∧ r=crit iR1 and
a2: ∃ iInv1 iInv2. iInv1 ≤ N ∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2
∧ f=inv1 iInv1 iInv2
3 shows invHoldRule s f r (invariants N)
4 proof -
from a1 obtain iR1 where a1:iR1 ≤ N ∧ r=crit iR1
  by blast
from a2 obtain iInv1 iInv2 where a2: iInv1 ≤ N
∧ iInv2 ≤ N ∧ iInv1 ≠ iInv2 ∧ f=inv1 iInv1 iInv2
  by blast
5 have iR1=iInv1 ∨ iR1=iInv2 ∨ (iR1 ≠ iInv1 ∧ iR1 ≠ iInv2)
by auto
6 moreover{assume b1:iR1=iInv1
7   have invHoldRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp,
rule_tac x=¬ (x=true ∧ n[iInv2]=C) in exI,auto)qed
8   then have invHoldRule s f r (invariants N) by auto}
9 moreover{assume b1:iR1=iInv2
10   have invHoldRule3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp,
rule_tac x=¬ (x=true ∧ n[iInv1]=C in exI,auto)qed
11   then have invHoldRule s f r (invariants N) by auto}
12 moreover{assume b1:(iR1 ≠ iInv1 ∧ iR1 ≠ iInv2)
13   have invHoldRule2 s f r
    proof(cut_tac a1 a2 b1, auto) qed
14   then have invHoldRule s f r (invariants N) by auto}
15ultimately show invHoldRule s f r (invariants N) by blast
16qed
```

In the above proof, line 2 are assumptions on the parameters of the invariant and rule, which are composed of two parts: (1) assumption `a1` specifies that there exists an actual parameter `iR1` with which `r` is a rule obtained by instantiating `crit`; (2) assumption `a2` specifies that there exists actual parameters `iInv1` and `iInv2` with which `f` is a formula obtained by instantiating `inv1`. Line 4 are two typical proof patterns forward-style which fixes local variables such as `iR1` and new facts such as `a1: iR1 ≤ N ∧ r=crit iR1`. From line 5, the remaining part is a typically readable Isar proof using calculation reasoning such as `moreover` and `ultimately` to do case analysis. Line 5 splits cases of `iR1` into all possible cases by comparing `iR1` with `iInv1` and `iInv2`, which is in fact characterized by $\mathsf{partition}([1],[2],[3]],[1,2])$. Lines 6-14 proves these cases one by one: Lines 6-8 proves the case where `iR1=iInv1`, line 7 first proves that the causal relation $\mathsf{invHoldRule}_3$ holds by supplying a symbolic formula, which is transformed from $invOnXC(2)$ by calling symbolize2f with lookUp strategy. From the conclusion at line 7, line 8 furthermore proves the causal relation invHoldRule holds; Lines 9-11 proves the case where `iR1=iInv2`, proof of which is similar to that of case 1; Lines 12-14 the case where neither `iR1=iInv1` nor `iR1=iInv2`. Each proof of a subcase is done in a block `moreover b1:asm1 proof1`, the `ultimately` proof command in line 15 concludes by summing up all the subcases.

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 $iInv_2$, $invHoldForRule\ s\ f\ r\ (invariants\ N)$.

```
lemma lemma_inv1_on_rules:
asumes a1: r ∈ rules N and
a2: (∃ _iInv1 _iInv2. _iInv1≤ N∧_iInv2≤
N∧iInv1≠iInv2∧f=inv1 iInv1 iInv2)
shows invHoldForRule 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
```

With the help of all the lemmas such as $\mathsf{lemma\_inv}_i\mathsf{\_on\_rules}$, we can prove the following lemma invs_on_rules which specifies that for all $f \in invariants\ N$ and $r \in rules\ N$, $invHoldForRule\ s\ f\ r\ (invariants\ N)$.

```
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 end
```

1) Lemmas on initial states: In this section, we 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 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 Rightarrow> 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
```

`initSpec0` and `initSpec1` specifies the assignments on each variable `n[i]` where `i ≤ N` and `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)
have formEval 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.

2) The main theorem: With the preparation of lemma on_inis and invs_on_rules, 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.

(1) 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. This can be solved done by applying lemma on_inis.

(2) For any invariant $inv \in (\text{invariants } N)$, any r in rule set rules N , one of the causal relations $\text{invHoldForRule}_{1-3}$ 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.

A. *Algorithms of Proof Generator* proofGen

In this subsection, we illustrate the key techniques and algorithms of generation of the lemmas and their proofs in subsection **??**. Being according with the order in which we introduce the above lemmas, we also introduce their generation in a bottom-up order. First let us introduce the generation of a subproof according to a relation tag of $invHoldForRule_{1-3}$, which is shown in Algorithm 11.

Algorithm 4: 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** $relTag = invHoldForRule_1$ **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** $relTag = invHoldForRule_2$ **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** $proof$

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 $invHoldForRule_{1-2}$, 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.

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: $subProof$

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

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 of generating doing 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.

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 Isablle proof: $proof$

$proof$ ←sprintf
" have %s by auto
%s
ultimately show %s by auto"
(partition, subproofs, concluding) ;
return *proof*

In algorithm 7, caseAnalzI generates a typical proof of doing case analysis to prove some causal 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 $doCaseAnalz(partition, subproofs, concluding)$.

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: $proof$

$cases \leftarrow caseField(rec)$;
$rels \leftarrow relItems(rec)$; $partition \leftarrow \bigvee cases$;
$subproofs \leftarrow$ "";
while $(cases \neq [])$ **do**
  $case \leftarrow hd(cases)$;
  $cases \leftarrow tl(cases)$;
  $rel \leftarrow hd(rels)$;
  $rels \leftarrow tl(rels)$;
  $subproofs \leftarrow subproofs\hat{}oneMoreOverGenI(case, rel)$;
$concluding \leftarrow$"invHoldForRule s f r (invariants N) ";
$proof \leftarrow doCaseAnalz(partition, subproofs, concluding)$;
return $proof$

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 $iInv_1...iInv_{lenPInv}$ 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.

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

$invItem \leftarrow tbl\_element(symbInvs, invName)$;
$lenPInv \leftarrow invParaNum(invItem)$;
$invParas \leftarrow invParasGen(lenPInv)$;
$constrOnInv \leftarrow symbForm2Isabelle(constrOfInv(invItem))$;
if *lenPInv = 0* **then**
  $asm \leftarrow$ "$a1 : f =$ "$\hat{}invName$;
else
  asm ← sprintf "a1: ∃ %s. %s ∧ f=%s %s" (invParas, constrOnInv, invName, invParas);
return *asm*

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 preparing functions, now the generation of a lemma on the causal relation such as $critVsInv1$ 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

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

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

$invItem \leftarrow tbl\_element(symbInvs, invName)$;
$lenPInv \leftarrow invParaNum(invItem)$;
$invParas \leftarrow invParasGen(lenPInv)$;
if $lenPInv = 0$ **then**
  obtain ← "";
else
  obtain ← sprintf "from a1 obtain %s where a1:%s ∧ f=%s %s by auto"
  $(invParas, constrOnInv, invName, invParas)$;
return *obtain*

the rule, $symRelItem$ is retrieved from $symCausalTab$ by $ruleName\hat{}invName$, and a proof $proof$ is generated by calling $caseAnalzI(symRelItem)$. At last these parts are filled into proper places in the lemma format.

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: $lemmaWithProof$

$asm1 \leftarrow asmGenOnInv(symbInvs, invName)$;
$asm2 \leftarrow asmGenOnRule(symbRules, ruleName)$;
$obtain1 \leftarrow obtainGenOnInv(symbInvs, invName)$;
$obtain2 \leftarrow obtainGenOnRule(symbRules, ruleName)$;
$symRelItem \leftarrow tbl\_element(symCausalTab, (ruleName\hat{}invName))$;
$proof \leftarrow caseAnalzI(symRelItem)$;
$lemmaWithProof \leftarrow$ 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 $lemmaWithProof$

Due to length limitation, we illustrate the algorithm for generating a key part of the proof of the lemma `critVsinv1`: the generation of a subproof (e.g., lines 7-8) according to a symbolic relation tag of $\mathsf{invHoldRule}_{1-3}$, which is shown in Algorithm 11. Input $relTag$ is the result of the generalization step, which is discussed in Section VI. In the body of function rel2proof, sprintf writes a formatted data to string and returns it. In line 10, $\mathsf{getFormField}(relTag)$ returns the field of formula f' if $relTag = \mathsf{invHoldRule}_3(f')$. rel2proof transforms a symbolic relation tag into a paragraph of proof, as shown in lines 7-8, 10-11, or 13-14. If the tag is among $\mathsf{invHoldRule}_{1-2}$, the transformation is rather straight-forward, else the form f' is assigned by the formula getFormField(relTag), and provided to tell Isabelle the formula which is used to construct the $\mathsf{invHoldRule}_3$ relation.

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

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

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

TABLE II
VERIFICATION RESULTS ON BENCHMARKS.

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

VIII. EXPERIMENTS

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

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

The invariants-searching algorithm used in our work differs from that in Cubeic [9], [13] in that the heuristics in our work are based on the construction of causal relation which is uniquely proposed in our work. Thus the auxiliary invariants in our work are different from those found in [9], [13]. Moreover, we generalize these concrete invariants and causal relations into a parameterized proof, and generate a parameterized proof in Isabelle. The found invariants have abundant semantics reflecting the deep insight of the FLASH protocol design, and the readable Isabelle proof script formally proves these invariants. In this way, we prove the protocol with the highest assurance. To the best of knowledge, this work for the first time automatically generates a proof of safety properties of full version of FLASH in a theorem prover without auxiliary invariants manually provided by people.

IX. CONCLUSION

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

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

REFERENCES

[1] Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In Alur, R., Henzinger, T., eds.: Computer Aided Verification. Volume 1102 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1996) 184–195

[2] Björner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Computer Science **173**(1) (1997) 49 – 87

[3] Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions? In: Proc. 13th International Conference on Computer Aided Verification. Volume 2102 of LNCS. Springer (2001) 221–234

[4] Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2031 of LNCS. Springer (2001) 82–97

[5] Tiwari, A., Rueß, H., Saïdi, H., Shankar, N.: A technique for invariant generation. In: Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2031 of LNCS. Springer (2001) 113–127

[6] Chou, C.T., Mannava, P., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Proc. 5th International Conference on Formal Methods in Computer-Aided Design. Volume 3312 of LNCS. Springer (2004) 382–398

[7] Pandav, S., Slind, K., Gopalakrishnan, G.: Counterexample guided invariant discovery for parameterized cache coherence verification. In: Proc. 13th IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods. Volume 3725 of LNCS. Springer (2005) 317–331
[8] Lv, Y., Lin, H., Pan, H.: Computing invariants for parameter abstraction. In: Proc. the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, IEEE CS (2007) 29–38
[9] Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel smt-based model checker for parameterized systems. In Madhusudan, P., Seshia, S., eds.: Proc. 24th International Conference on Computer Aided Verification. Volume 7358 of LNCS. Springer (2012) 718–724
[10] Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, Padua, Italy, ACM (1996) 288–296
[11] Mcmillan, K.L., Labs, C.B.: Parameterized verification of the flash cache coherence protocol by compositional model checking. In: In CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144, Springer (2001) 179–195
[12] Y. Li, K.d.: paraverifier (2016) https://github.com/paraVerifier/paraVerifier.
[13] Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: FMCAD, Portland, Oregon, USA (October 2013)