A

An Automatic Proving Approach to Parameterized Verification

Yongjian Li, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Kaiqiang Duan, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
David N. Jansen, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Jun Pang, Faculty of Science, Technology and Communication and Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
Lijun Zhang, State Key Laboratory of Computer Science, Institute of Software, University of Chinese Academy of Sciences, Chinese Academy of Sciences
Yi Lv, State Key Laboratory of Computer Science, Institute of Software, University of Chinese Academy of Sciences, Chinese Academy of Sciences
Shaowei Cai, State Key Laboratory of Computer Science, Institute of Software, University of Chinese Academy of Sciences, Chinese Academy of Sciences

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

CCS Concepts: •**General and reference → Verification;** •**Theory of computation → Logic and verification; Automated reasoning; Higher order logic;** *Verification by model checking;* •**Hardware → Theorem proving and SAT solving;** •**Software and its engineering → Software verification; Formal software verification;**

Additional Key Words and Phrases: Automatic verification, theorem proving, inductive methods, invariant and proof generation, cache coherence protocols

ACM Reference Format:
Yongjian Li, Kaiqiang Duan, David N. Jansen, Jun Pang, Lijun Zhang, Yi Lv, and Shaowei Cai *ACM* V, N, Article A (January YYYY), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Formal verification of parameterized protocols is of great interest, mainly due to the practical importance of such systems. Parameterized protocols exist in many important application areas, including cache coherence, security, and communication networks. The most important

---

Yongjian Li is supported by grant 61672503 from National Natural Science Foundation and grant 2017YF-B0801900 from the National Key Research and Development Program in China. Lijun Zhang is supported by grants 61532019,61761136011,61472473 from National Natural Science Foundation.
Author's addresses: Y. Li, K. Duan, D. N. Jansen, L. Zhang, Y. Lv, and S. Cai, The State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, No. 4, South 4th Street, Zhongguancun, Haidian District, Beijing 100190, China. University of Chinese Academy of Sciences, No.19(A) Yuquan Road, Shijingshan District, Beijing, P.R.China 100049; J. Pang, University of Luxembourg, Faculty of Science, Technology and Communication, 6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© YYYY ACM. 0000-0000/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

parameter is typically the number of parties in the protocol or the network. The specification of each node is necessarily of the same form, leading to symmetries; otherwise, the specification of the protocol as a whole would have unbounded size. The significant challenge in parameterized verification is mainly due to the correctness requirement that the desired properties should hold for any instance of the parameterized protocol, be it small or large. *Model checking* [Clarke et al. 1999; Baier and Katoen 2008], as an automatic verification technique based on state exploration, is powerful and efficient in the verification of non-parameterized protocols, but it lacks the power to verify parameterized protocols – mainly because model checking can only check a single instance of the parameterized protocol in each verification. A more desirable approach is *theorem proving,* which provides a proof that the correctness holds for any instance of the verified system.

Related work. There have been a lot of studies in the field of parameterized verification [Park and Dill 1996; Pnueli and Shahar 1996; Bjørner et al. 1997; Arons et al. 2001; Pnueli et al. 2001; Tiwari et al. 2001; Baukus et al. 2002; Chou et al. 2004; Talupur and Tuttle 2008; Pandav et al. 2005; Lv et al. 2007; Ghilardi et al. 2008; Ghilardi and Ranise 2010; Conchon et al. 2012; 2013; Dooley and Somenzi 2016]. We discuss in the following the papers most relevant to our work.

The *invisible invariants* method [Arons et al. 2001; Pnueli et al. 2001] 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. [2007] 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 approaches 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 which can been understood by humans. 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 [Kuskin et al. 1994].

The method of Chou et al. [2004] 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 $\{P_1,\dots,P_m,P^*\}$ (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 use 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 when the refined abstract system eventually satisfies the original invariant as well as all the auxiliary invariants supplied by the user. To the best of our knowledge, its correctness has not been formally proved in a theorem prover, thus the method's soundness is only argued in an informal way. This situation may not be ideal because its application domain, cache coherence protocols, demands the highest assurance for correctness. Besides, the analysis of counterexample and generation of new auxiliary invariants usually depend on human insights of the protocol. It is too laborious for the user to perform this analysis, thus effective and automatic tools are still needed.

Predicate abstraction has also been applied to the verification of parameterized protocols. Baukus et al. [2002] used it to verify the German cache coherence protocol (without data paths), and Park and Dill [1996] used it to verify the FLASH protocol. The main idea of pred-

icate abstraction is to discover a set of predicates to abstract the states of a system, thus, an abstract state is a valuation of the predicates. Unfortunately, the task of discovering proper predicates is neither easy nor automatic. Moreover, the abstract system has to be proven to conserve certain properties under verification. This proof also needs a set of auxiliary invariants. Therefore, searching for sufficiently strong auxiliary invariants cannot be avoided. After for the above two publications, no continued efforts have been made to apply predicate abstraction to parameterized verification.

Conchon et al. [2013] recently have made progress in automatically searching for auxiliary invariants. They first compute a set of reachable states S using a forward-image computation on a finite small instance of the parameterized protocol. Then they iteratively perform an over-approximate backward reachability analysis of the parameterized protocol and check whether it presents states that not contained in S. The computed over-approximations will be treated as invariants and be model checked. This algorithm is implemented in an SMT-based model checker. The correctness of the algorithm is argued in a generic symbolic framework. Similar to previous approaches, the formulation of the algorithm and its proof are not given in a theorem prover.

Most recently, Dooley and Somenzi [2016] 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* [Arons et al. 2001; Pnueli et al. 2001]. Here the small-model theorem indicates that for the assertions we generate, it suffices to validate them on small instantiations (whose size are less than a cut-off value) in order to conclude validity on arbitrary instantiations. The cut-off depends on the system and the candidate assertion.

Motivation. From the above discussion on related work, 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, the cache coherence protocol of the Stanford FLASH multiprocessor [Kuskin et al. 1994] is so complex that only few approaches have successfully verified it [Park and Dill 1996; Chou et al. 2004; Conchon et al. 2013]. Furthermore, all existing successful verification approaches have their own weaknesses. The approach of Park and Dill [1996] is theorem-proving based, but it requires the manual construction of inductive invariants. The work by Chou et al. [2004] is similar to Park and Dill [1996]'s in that hand-crafted invariants are required. In contrast, the approach of Conchon et al. [2013] 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 in a theorem prover. Approaches with a large degree of automation [Arons et al. 2001; Pnueli et al. 2001; Emerson and Kahlon 2003; Dooley and Somenzi 2016] are often based on a small-model theorem. Such an approach works well for simple protocols such as MESI [Papamarcos and Patel 1984]. While being neat theoretically, finding the cut-off value is difficult in practice. Although some work [Abdulla et al. 2013; Kaiser et al. 2010] have pushed the limits of cut-off techniques beyond MESI, these solutions still run into serious problems when the cut-off value itself is large. For instance, the FLASH protocol cannot be verified using these cut-off techniques.

In general, it is not difficult to formalize a parameterized protocol and its properties for a theorem prover. The difficulty lies in proving these properties. The most creative steps in a theorem prover based formal proof are made by humans, and these are hard to automatize. These steps mainly lie in: (1) the invariant finding scheme; (2) performing case analysis – different proof strategies need to be used in different cases; and (3) generalization of the invariants. Up to now, the main effort is concentrated on searching invariants automatically, for instance in the frameworks discussed in Arons et al. [2001], Lv et al. [2007], Ghilardi et al. [2008], Ghilardi and Ranise [2010] and Conchon et al. [2013]. However, few research

has considered to link the invariant search with proof checking. The generalization from a result obtained in the concrete protocol instance to the parameterized version is the key for exploiting theorem provers. This is the reason why few researchers use theorem provers in parameterized verification. Those who did apply theorem provers needed to intervene manually to prove the properties.

Our Contributions. In this paper we propose a unified framework to verify whether a parametric protocol satisfies a property. We combine invariant searching with proof checking techniques. Our framework automates the finding of an invariant, case analysis and generalization. Inspired by existing approaches, we first start with a small concretization of both the property and the protocol. As the protocol under consideration is symmetric, we can enumerate a set of concretizations that together are general enough to catch all relevant cases, when we consider each concretized pair (req, r) of a property and a transition in the protocol separately. We annotate simple cases in which the property *req* can be directly proven under the transition r. If not, we apply an off-the-shelf model checker to find a stronger invariant property *aux* that will be used as candidate auxiliary invariant for proving *req*. Since we carefully register the mapping between concrete indices and the original symbolic parameters, we are able to generalize the stronger requirement *aux* back to a parameterized one easily. As a result, we can automatically generate a proof dependency graph among all the candidate invariants. Finally, we transform the proof dependencies into subproofs, which can then be submitted to a theorem prover. We have implemented our approach in our main tool and have demonstrated that it can solve – automatically – industrial-sized case studies such as the FLASH protocol.

Organization of the paper. The rest of the paper is organized as follows. Section 2 introduces the preliminaries and also formalises the praramterized verification problem. An overview of our approach is given in Section 3. Its main steps are then given in Section 4 for concretizing properties and protocols, Section 5 for identifying simple cases and finding auxiliary invariants, Section 6 for generalizing symbolic (auxiliary) invariants and generating the proof dependency graph. An application of our approach to the FLASH protocol is presented in Section 7. Section 8 presents a summary on our experiments on other real-world protocols. Section 9 concludes our article.

For interested readers, we also sketch the automatic proof generation step in Appendix A: it illustrates how the Isabelle proof scripts are generated for our running example.

A preliminary version of the article was presented at the IEEE International Conference on Computer Design [Li et al. 2016]. In this article, we have restructured the material, and simplified most of the technical presentations to make it more accessible.

2. THE PROBLEM: DOES THIS (PARAMETRIC) PROTOCOL SATISFY ITS REQUIREMENTS?

In this section, we give a high-level description of the problem we want to tackle, next to introducing a number of basic notations. We work in the context of a *communicating network* of computers; we call them *nodes.* Every node runs the same protocol; we will first explain how this protocol is described. We explicitly let the number of nodes N vary; this makes the system *parametric.* For simplicity, we assume that nodes are identified by the numbers $1, 2, \ldots, N$. We use arrays indexed by these numbers to describe local variables of nodes. The network as a whole should satisfy some properties, e. g. mutual exclusion or distributed consensus, as a consequence of running the protocol, for every possible value of $N \in \mathbb{N}$. We also explain how this property is described.

Permutations. We frequently use a list of distinct node identities, a *m-permutation* of N with $m \leq N$. A m-permutation of N is an ordered arrangement of a m-element subset of the node identity set $\{1, 2, \ldots, N\}$. For instance, $[1, 2]$ and $[2, 3]$ are two 2-permutations of 3. We

use perms_N^m to stand for the set of all m-permutations of N, e. g., $\mathsf{perms}_3^2 = \{[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]\}$. Recall that there are $N!/(N-m)!$ such m-permutations of N.

States. We first assume given a finite set of state variables $V = \{v_0, v_1, \ldots, v_n\}$, which range over a finite set D. With these variables, first-order expressions e and formulas (also called predicates) f can be defined as usual. We allow arrays of variables indexed with node identities, typically used for local variables of each node, and non-array (global) variables. A state s of the protocol is a mapping from all variables in V to D. We use $s(v_i)$ to get the value for v_i in state s; We extend the notation to $s(e)$ to denote the evaluation result of the expression e in state s. By $s \models f$, we denote that the formula f is evaluated to be true in state s, and $\models f$ means that $s \models f$ for all states s. For parallel assignments, we write $A = \{v_\ell := e_\ell \mid \ell = 1, \ldots, n\}$; every variable v_ℓ can be a non-array variable or an array entry indexed with a parameter.

We define the notion of the weakest precondition $\mathsf{WP}(A, f) \equiv f[v_1 := e_1, \ldots, v_n := e_n]$, which substitutes each occurrence of a variable v_ℓ by the corresponding e_ℓ in formula f.

Protocols. A communication protocol $\mathscr{P}$ is formalized as a pair (I, R), where I is a set of initial predicates over V and R is a set of guarded commands or protocol rules. Each guarded command $r \in R$ has the form $g \triangleright A$, where the guard g is a predicate over V and A is a parallel assignment to variables $v_\ell := e_\ell$. We write $\mathsf{guard}(r) = g$, $\mathsf{action}(r) = A$ for a guarded command r, and vars(A) for the variables $\{v_\ell | v_\ell := e_\ell \in A\}$ to be assigned by A. A state transition is caused by triggering and executing a guarded command r. Formally, we define:[1]

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

Every state that satisfies some predicate $f \in I$ is an initial state.

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

Parameterized formulas, guarded commands, and protocols. For simplicity, a parameterized formula is a function $f(i_1, i_2, \ldots, i_m)$ from a tuple of node identities to a formula. Similarly, a parameterized guarded command is a function $r(j_1, j_2, \ldots, j_k)$ from a tuple of node identities to a rule; we normally use i to denote parameters of formulas and j to denote parameters of guarded commands, a distinction which will become important later. Finally, a parameterized protocol is a protocol $\mathscr{P}(i_1, \ldots, i_m, j_1, \ldots, j_k) = (I, R)$ where I contains parameterized formulas for initial predicates and R contais parameterized guarded commands.

Without loss of generality, we require that node identities assigned to distinct parameters to instantiate a parameterized object (be it a formula, rule or protocol) are distinct. To instantiate a parameterized object with m parameters, we use a m-permutation of N as actual parameter list. A parameterized formula is satisfied if every assignment of distinct node identities to the formal parameters leads to a satisfying instantiation. For instance, $\mathsf{mutualInv}(i_1, i_2) \equiv \neg(a[i_1] = \mathsf{C} \wedge a[i_2] = \mathsf{C})$ is a parameterized formula with two distinct parameters i_1 and i_2. It requires that the local variables a of any two *distinct* nodes never are C (for “critical section”) at the same time, ensuring mutual exclusion.

The Problem. We assume given a network of N nodes; its (global) state space is described by the variables in V, and every node runs a communication protocol described by $\mathscr{P} = (I, R)$.

[1]We use $\equiv$ to describe syntactic equality between formulas. We use $\Longleftrightarrow$ for (logical) equivalence between two formulas.

One typically wants to prove some requirements of the protocol; we concentrate here on invariant requirements, which can be specified using a (parametric) formula without temporal modalities. The question we want to answer by our approach is:

> *Do the nodes, which run $\mathscr{P}$, satisfy some invariant requirement req?*
> Or, equivalently:
> *Does req hold in every reachable state of the nodes running $\mathscr{P}$?*

In general, a requirement of a protocol can be any formula in a suitable temporal logic like LTL or CTL. In this work, we focus on invariant checking, and we write the requirement as an invariant of the form $\neg \bigwedge_{n=1}^{k} f_n$, where each f_n is a literal formula of the form $e_1 = e_2$ or $e_1 \neq e_2$.

Example 2.1. 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 global boolean variable, used to indicate that the critical resource is available. a is an array containing local variables: each $a[i]$ describes the state of node i.

Our simple mutual exclusion protocol has the following guarded commands:

$$\begin{array}{ll}
\mathsf{try}(j) \equiv & a[j] = \mathsf{I} \ \triangleright\ a[j] := \mathsf{T} \\
\mathsf{crit}(j) \equiv & x = \mathsf{true} \wedge a[j] = \mathsf{T} \triangleright \{a[j] := \mathsf{C}, x := \mathsf{false}\} \\
\mathsf{exit}(j) \equiv & a[j] = \mathsf{C} \triangleright\ a[j] := \mathsf{E} \\
\mathsf{idle}(j) \equiv & a[j] = \mathsf{E} \triangleright \{a[j] := \mathsf{I}, x := \mathsf{true}\}.
\end{array}$$

The protocol itself is defined by $\mathsf{mutexini}_N$, the predicate to specify the initial state of the protocol, and $\mathit{mutexrules}_N$, the set of protocol rules or guarded commands:

$$\begin{array}{ll}
\mathsf{mutexini}_N & \equiv x = \mathsf{true} \wedge \bigwedge_{i=1}^{N} a[i] = \mathsf{I} \\
\mathit{mutexrules}_N & = \{\mathsf{try}(j), \mathsf{crit}(j), \mathsf{exit}(j), \mathsf{idle}(j) \mid j = 1, \ldots, N\} \\
\mathscr{M}\mathit{utual}\mathscr{E}x_N & = (\mathsf{mutexini}_N, \mathit{mutexrules}_N)
\end{array}$$

and it should satisfy the invariant requirement:

$$\mathsf{mutualInv}(i_1, i_2) \equiv \neg(a[i_1] = \mathsf{C} \wedge a[i_2] = \mathsf{C}).$$

The property $\mathsf{mutualInv}(i_1, i_2)$, as defined above, describes that $a[i_1]$ and $a[i_2]$ cannot be in a critical state at the same time. It is silently understood that $i_1 \neq i_2$.

With the above example one can see why our problem is non-trivial. Both the guarded commands and the formula may contain parameters. In particular, the guarded commands of node i typically contain parameter i to access local variables; the property formula like $\mathsf{mutexini}_N$ typically contains N, the number of protocol nodes. Model checking would allow to check a protocol instance for a fixed N (as long as there is no state space explosion); theorem proving, while in principle powerful enough to solve it, faces the difficulty that a general-purpose theorem prover does not know what approach might lead to a successful proof.

3. OUR SOLUTION: GET INSPIRED BY CONCRETE INSTANCES

In this section, we want to give a first overview of the method we propose to tackle parametric protocol verification. Assume given a communication protocol and an invariant requirement formula as described in Section 2. We propose to guide a theorem prover by providing *inductive invariant* candidates, which are not only satisfied throughout the running time of the protocol, but also can be proven to be invariants by induction over the guarded commands of the protocol. An inductive invariant is defined as follows:

Definition 3.1. A formula *inv* is an *inductive invariant* of protocol $\mathscr{P} = (I, R)$ if

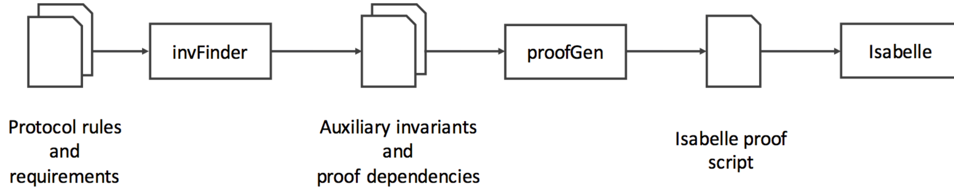

Fig. 1: The workflow of paraVerifier.

— the initial state satisfies the formula: $\models I \to inv$, and
— transitions preserve the formula: for each $r \in R$, we have $\models inv \wedge \mathsf{guard}(r) \to \mathsf{WP}(\mathsf{action}(r), inv)$.

A formula describing the set of reachable states is an example of an inductive invariant.

Others define invariants by comparing the pre- and post-state of a guarded command: If an inductive invariant inv holds at a state s, then inv also holds at the post-state s' after a rule r is executed from s. The lemma below captures that the two definitions are equivalent.

LEMMA 3.2. *Let $\mathscr{P} = (I,R)$ be a protocol. A formula inv is an inductive invariant of the protocol if and only if for all states s and s' and guarded commands $r \in R$ with $s \xrightarrow{r} s'$, $s \models inv$ implies $s' \models inv$.*

PROPOSITION 3.3. *Assume given a protocol $\mathscr{P} = (I,R)$ and an (invariant) requirement req. $\mathscr{P}$ satisfies req if there exists an inductive invariant inv of $\mathscr{P}$ such that $\models inv \to req$.*

We will use this proposition to prove the correctness of the protocol. In simple cases, the requirement *req* itself can be used as inductive invariant; however, sometimes we need to add auxiliary invariants to make the (conjunction of all) invariants inductive. As usual, finding suitable auxiliary invariants is the challenging task. In principle, we need take advantage of the symmetry of the protocol. Look at a small concretization of the protocol: as every node runs the same protocol rules, which means a node has similar behaviors as any others. This allow us to find a set of concretizations that is general enough to catch all relevant cases.

To find candidates for auxiliary invariants, we use the following steps:

(1) We first concretize the requirement *req* (See Section 4). Assume req contains n parameters $i_1, \dots, i_n$, pairwise different. We can choose a single injective mapping of these parameters to natural numbers (the easiest one is $i_1 \mapsto 1, \dots, i_n \mapsto n$); as the node identities are symmetric, the natural numbers chosen do not matter, so a single concretization suffices.
(2) We then concretize the guarded commands that describe how a single node runs the protocol. Because we already have concretized some node identities for *req*, we have to be careful to select concretizations that still cover all possible combinations of instantiations of parameters in *req* and the guarded command r together. Note that we concretize each pair (req, r) and record all the necessary actual parameter indices for each r separately. (See Section 4 for details of the concretization strategy.)
(3) We check whether the concretized requirement is an inductive invariant and, if necessary, find a strengthening *aux* that will be used as candidate auxiliary invariant (See Section 5). Here the concretizations allow us to exploit a model checker (we use NuSMV and Murphi) to ensure that *aux* holds in a small instance of the protocol.

(4) We generalize the strengthened requirement *aux* back to a parameterized one. This is possible because we carefully registered the indices of instances in the first two steps. (See Section 6 for details of the generalization strategy.)

We implemented the above four steps in our main tool invFinder, whose usage is shown in Figure 1. Our tool chain also contains proofGen (See Appendix A for this step) and Isabelle [Nipkow et al. 2002]; altogether, the tool chain is called paraVerifier. We emphasize that the tool-chain works *fully automatic:* in contrast to existing approaches, our approach automatically finds auxiliary candidate invariants, and automatically tags their proof dependencies. proofGen automatically transforms them into Isabelle proof scripts. In turn, the output proof script will be fed into Isabelle to guide it in proving the requirement. The scripts provide enough detail for Isabelle to create the proof without human intervention even for the highly non-trivial FLASH protocol (See Section 7). Isabelle will also detect whether some step introduced an unsound invariant candidate *aux*.

4. HOW TO CONCRETIZE THE REQUIREMENT AND THE PARAMETRIC PROTOCOL

This section describes the first two steps of the overall procedure implemented in invFinder: we show how a requirement and a guarded command can be concretized. We only need to concretize one requirement and one guarded command at the same time, as the second condition of Def. 3.1 can be proven separately per pair (requirement, command).

Because every node in the network runs the same protocol, the execution is highly symmetric, and we can restrict our attention to a small set of exemplary instantiations that cover all possible combinations of nodes that might be involved in a requirement or in a guarded command.

In the following sections, we use $\mathsf{nth}(xs, i)$ to denote the the i-th element of the list xs (counting from 1 as the first element).

4.1. Concretizing parameter values for a requirement

As the numeric value of a node identity does not matter, the protocol is highly symmetric. If some parametric requirement $req(i_1, \ldots, i_m)$ holds for one m-permutation of nodes, it also holds for other m-permutations. If one wants to prove this in detail, one can define symmetry transformations between the different possible concretizations of a requirement, mapping $req(xs)$ (with parameter list xs) to $req(ys)$. As xs and ys need to be lists of m *distinct* elements, it is easy to replace every occurrence of $\mathsf{nth}(xs, i)$ by $\mathsf{nth}(ys, i)$. Similarly, one could adapt a proof of $req(xs)$ to a proof of $req(ys)$ by the same replacement.

Therefore, we simply insert $1, 2, \ldots, m$ as the node identities. We denote the concretized requirement by req^{c}.

4.2. Concretizing parameter values for a guarded command

We want to prove, for every guarded command separately, that the requirement is an inductive invariant. Again we will try to use symmetry as much as possible, but we have to take care that we cover every combination of nodes involved in the several roles of the requirement. We first demonstrate the idea by our running example.

Example 4.1. Assume that the number of actual parameters occurring in *req* is m, and that of formal parameters occurring in $g \triangleright A$ is k. We want to find a set of k-element lists that covers the needed concretizations of $g \triangleright A$.

(1) If $m = 2$ and $k = 1$, then we have obviously three possible concretizations, which we denote by $paras = \{[1], [2], [3]\}$. Note here we assume [3] is symmetry-equivalent to [4], [5] etc. This explains why the latter concretizations are not needed.
For example, we get this case when we concretize the invariant $\mathsf{mutualInv}(i_1, i_2)$ with $i_1 = 1$, $i_2 = 2$ and the guarded command $\mathsf{crit}(j)$ with $j = 1$, 2 or 3.

(2) If $m = 2$ and $k = 2$, then we have $paras = \{[1,2],[2,1],[1,3],[2,3],[3,1],[3,2],[3,4]\}$. Note that $[1,3]$ is symmetry-equivalent to $[1,4]$, thus only one of the two is selected. However, we need to choose an additional $[3,4]$ to cover the case that both parameters in $g \triangleright A$ are disjoint with the parameters of *req*.

Generally, for a k-element list of node identities, we have to choose which ones of them are in $\{1,\ldots,m\}$, and set the remaining parameters to distinct values $> m$. This leads to $\sum_{j=0}^{\min\{k,m\}} \binom{k}{j} \frac{m!}{(m-j)!}$ possibilities.

We denote a concretized guarded command by $r^{c} = g^{c} \triangleright A^{c}$. Here, c is a placeholder for an indication which element of paras is chosen.

Finally, we remark that the above two steps can be swapped: we could have chosen to concretize the guarded command first, and then correspondingly the requirement.

5. CONSTRUCT CONCRETE AUXILIARY INVARIANTS

After having concretized both the requirements and the protocol rules, invFinder proceeds to find out whether the requirements can serve as inductive invariants or need to be strenghtened by adding auxiliary invariants.

5.1. Identifying Simple Cases

A novel feature of our approach lies in that three kinds of proof dependencies are exploited, which are essentially special cases of the general rule for inductive invariants.

Definition 5.1. Assume given a protocol $\mathscr{P} = (I,R)$, a state of the protocol s, a guarded command $r \in R$, a requirement f over V, and a formula set F. We consider the following proof dependencies:

(1) $\mathsf{CR}_{\mathrm{estbl}}(inv,r)$ means: $\models \mathsf{guard}(r)$ implies $\models \mathsf{WP}(\mathsf{action}(r),inv)$, i. e. one can prove that the guarded command r establishes invariant inv independent from the previous state.
(2) $\mathsf{CR}_{\mathrm{presv}}(inv,r)$ means: $\models inv$ implies $\models \mathsf{WP}(\mathsf{action}(r),inv)$, i. e. one can prove that the guarded command r preserves invariant inv. (This typically holds for guarded commands that do not touch variables appearing in inv.)
(3) $\mathsf{CR}_{\mathrm{aux}}(inv,r,F)$ means: there exists a formula $aux \in F$ such that $\models aux \wedge \mathsf{guard}(r)$ implies $\models \mathsf{WP}(\mathsf{action}(r),inv)$, i. e. one can prove that the guarded command r ensures invariant inv based on an auxiliary invariant $aux \in F$. In this case, there is an invariant dependency and we will store the corresponding pair $(\mathsf{CR}_{\mathrm{aux}},aux)$.

We use the abbreviation $\mathsf{CRS}(inv,r,F)$ for $\mathsf{CR}_{\mathrm{estbl}}(inv,r) \vee \mathsf{CR}_{\mathrm{presv}}(inv,r) \vee \mathsf{CR}_{\mathrm{aux}}(inv,r,F)$. It states that there is a proof dependency between inv, r, and F, and if it holds, then the conjunction of all formulas of F is an inductive invariant. (Note that individual formulas in F may not be inductive invariants.) We state this as a theorem:

THEOREM 5.2. *For a protocol $\mathscr{P} = (I,R)$, if $\models I \rightarrow inv$ and $\mathsf{CRS}(inv,r,F)$ for any formula $inv \in F$, and guarded command $r \in R$, then $\bigwedge F$ is an inductive invariant of $\mathscr{P}$.*

PROOF. Assume given an arbitrary guarded command $r \in R$. According to Def. 3.1, we need to show $\models (\bigwedge F) \wedge \mathsf{guard}(r) \rightarrow \mathsf{WP}(\mathsf{action}(r), \bigwedge F)$.

It is easy to see that $\mathsf{WP}(\mathsf{action}(r), \bigwedge F)$ holds if $\mathsf{WP}(\mathsf{action}(r),inv)$ holds for every $inv \in F$. So, let $inv \in F$ and state s be arbitrary. We assume that (1) $s \models (\bigwedge F) \wedge \mathsf{guard}(r)$ and we need to show (2) $s \models \mathsf{WP}(\mathsf{action}(r),inv)$. From the antecedents of the theorem we know $\mathsf{CRS}(inv,r,F)$. We distinguish the three cases for this tuple:

— If $\mathsf{CR}_{\mathrm{estbl}}(inv,r)$, then $s \models \mathsf{guard}(r)$ implies (2); but this is implied by (1).
— If $\mathsf{CR}_{\mathrm{presv}}(inv,r)$, then $s \models inv$ implies (2); but this is implied by (1).

— If $\mathsf{CR}_{\text{aux}}(inv, r, F)$, then we know that there exists a formula $aux \in F$ such that $s \models aux \wedge \mathsf{guard}(r)$ implies (2). Whatever aux may be, it is included in the conjunction $\bigwedge F$ in (1), so (2) follows. □

We emphasize that a single requirement $inv \in F$ is only an invariant and not necessarily inductive. By Thm. 5.2, if we can find a set of invariants F that satisfies the antecedents of this theorem, then $\bigwedge F$ is an inductive invariant, which obviously implies each $inv \in F$. From Prop. 3.3, we can then conclude that $\mathscr{P}$ satisfies each such requirement inv. In our work, we attack invariant checking of the parameterized protocol system by constructing such a set of formulas F that includes the original requirement.

Example 5.3. We continue to illustrate the above definitions on the mutual exclusion protocol. We first give a few auxiliary invariants for proving $\mathsf{mutualInv}(i_1, i_2)$. To specify that the variable x is false whenever node i stays in its critical section:

$$\mathsf{invOnXC}(i) \equiv \neg(x = \mathsf{true} \wedge a[i] = \mathsf{C})$$

To specify that x is false when the node exits from its critical section:

$$\mathsf{invOnXE}(i) \equiv \neg(x = \mathsf{true} \wedge a[i] = \mathsf{E})$$

To express that nodes i_1 and i_2 cannot stay in and exit from their critical sections at the same time:

$$\mathsf{aux1}(i_1, i_2) \equiv \neg(a[i_1] = \mathsf{C} \wedge a[i_2] = \mathsf{E})$$

To state that nodes i_1 and i_2 cannot exit from their critical sections at the same time:

$$\mathsf{aux2}(i_1, i_2) \equiv \neg(a[i_1] = \mathsf{E} \wedge a[i_2] = \mathsf{E})$$

Finally, the set of (original and auxiliary) invariants for the mutual exclusion protocol is denoted:

$$mutexinvs_N = \{\mathsf{mutualInv}(i_1, i_2)\,,\ \mathsf{invOnXC}(i_1)\,,\ \mathsf{invOnXE}(i_1)\,,\ \mathsf{aux1}(i_1, i_2)\,,\ \mathsf{aux2}(i_1, i_2) \mid i_1 \neq i_2, i_1 = 1, \ldots, N \text{ and } i_2 = 1, \ldots, N\}$$

We now continue to illustrate the three proof dependencies $\mathsf{CR}_{\text{estbl}}$, $\mathsf{CR}_{\text{presv}}$, and CR_{aux}. Consider the invariant $\mathsf{mutualInv}(i_1, i_2)$ and the guarded command $\mathsf{crit}(j)$ (with the understanding that $i_1, i_2, j \in \{1, \ldots, N\}$ and $i_1 \neq i_2$). We have the following proof dependencies.

— $\mathsf{CR}_{\text{presv}}(\mathsf{mutualInv}(i_1, i_2), \mathsf{crit}(j))$, with $i_1 \neq j \neq i_2$. This holds because $\mathsf{WP}(\mathsf{action}(\mathsf{crit}(j)), \mathsf{mutualInv}(i_1, i_2)) \equiv \mathsf{mutualInv}(i_1, i_2)$.
— $\mathsf{CR}_{\text{aux}}(\mathsf{mutualInv}(i_1, i_2), \mathsf{crit}(i_1), mutexinvs_N)$. This holds as $\mathsf{invOnXC}(i_2) \in mutexinvs_N$ and $\mathsf{invOnXC}(i_2) \wedge \mathsf{guard}(\mathsf{crit}(i_1)) \implies a[i_2] \neq \mathsf{C} \iff \neg(\mathsf{C} = \mathsf{C} \wedge a[i_2] = \mathsf{C}) \equiv \mathsf{WP}(\mathsf{action}(\mathsf{crit}(i_1)), \mathsf{mutualInv}(i_1, i_2))$.
— $\mathsf{CR}_{\text{aux}}(\mathsf{mutualInv}(i_1, i_2), \mathsf{crit}(i_2), mutexinvs_N)$. This holds as $\mathsf{invOnXC}(i_1) \in mutexinvs_N$ and $\mathsf{invOnXC}(i_1) \wedge \mathsf{guard}(\mathsf{crit}(i_2)) \implies a[i_1] \neq \mathsf{C} \iff \neg(a[i_1] = \mathsf{C} \wedge \mathsf{C} = \mathsf{C}) \equiv \mathsf{WP}(\mathsf{action}(\mathsf{crit}(i_2)), \mathsf{mutualInv}(i_1, i_2))$.

In Example 5.3, $\mathsf{CR}_{\text{estbl}}(inv, r)$ and $\mathsf{CR}_{\text{presv}}(inv, r)$ can be checked straightforwardly by a theorem prover. $\mathsf{CR}_{\text{aux}}(inv, r, mutexinvs_N)$ 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 existential quantifier. We notice that two things are needed to be done to guide a theorem prover to automatically check $\mathsf{CRS}(inv, r, mutexinvs_N)$: (1) the case distinction which rule parameters $j_1, j_2, \ldots$ are to be identified with which invariant parameters $i_1, i_2, \ldots$; (2) the choice among the three kinds of proof dependencies (and, for CR_{aux}, the relevant auxiliary invariant).

5.2. Finding Concrete Auxiliary Invariants

If neither $\mathsf{CR}_{\mathrm{estbl}}$ nor $\mathsf{CR}_{\mathrm{presv}}$ holds for a requirement *req* and a guarded command r, then a new auxiliary invariant *newInv* will be constructed, which will make the proof dependency $\mathsf{CR}_{\mathrm{aux}}$ hold. The second condition of Def. 3.1 is equivalent to $inv \wedge \mathsf{guard}(r) \rightarrow \mathsf{WP}(\mathsf{action}(r), inv)$; this suggests to choose $newInv^{\mathrm{c}} \equiv \neg\mathsf{guard}(r^{\mathrm{c}}) \vee \mathsf{WP}(\mathsf{action}(r^{\mathrm{c}}), req^{\mathrm{c}})$. However, this formula is in general rather complex, and below we describe a way to find simpler formulas.

Recall that the invariant property we consider in this paper has the form $\neg \bigwedge_n f_n$, where each f_n is an atomic formula or predicate. Thus, $\mathsf{WP}(\mathsf{action}(r^{\mathrm{c}}), req^{\mathrm{c}})$ is a simple concrete formula of the same shape. If $\neg\mathsf{guard}(r^{\mathrm{c}})$ also has this shape, the formula $\neg\mathsf{guard}(r^{\mathrm{c}}) \vee \mathsf{WP}(\mathsf{action}(r^{\mathrm{c}}), req^{\mathrm{c}})$ can again be transformed into it.

We denote the obtained formula by $\neg \bigwedge_{n=1}^{k} f_n^{\mathrm{c}}$. Candidates for the concrete auxiliary invariant are formulas $\neg f$, where f is a conjunction of a few of the $f_1^{\mathrm{c}}, \ldots, f_k^{\mathrm{c}}$. Observe that f allows more states than the original formula and thus may not be an inductive invariant. Thus, we apply model checking to find a simple formula $\neg f$.

We now illustrate the above steps using a simple example:

Example 5.4. Assume given the concrete requirement $\mathsf{mutualInv}(1,2) = \neg(a[1] = \mathsf{C} \wedge a[2] = \mathsf{C})$ and the concrete action $\mathsf{crit}(1) \equiv (a[1] = \mathsf{T} \wedge x = \mathsf{true} \rhd \{a[1] := \mathsf{C}, x := \mathsf{false}\}$. The weakest precondition is $\neg(\mathsf{C} = \mathsf{C} \wedge a[2] = \mathsf{C})$. The resulting disjunction $\neg\mathsf{guard}(\mathsf{crit}(1)) \vee \mathsf{WP}(\mathsf{action}(\mathsf{crit}(1)), \mathsf{mutualInv}(1,2))$ is equivalent to $\neg(a[1] = \mathsf{T} \wedge x = \mathsf{true} \wedge \mathsf{C} = \mathsf{C} \wedge a[2] = \mathsf{C})$.

We now have four conjuncts from which we can choose to create a simple inductive invariant candidate. It could perhaps be simplified further to $\neg(a[1] = \mathsf{T} \wedge x = \mathsf{true})$ or $\neg(x = \mathsf{true} \wedge a[2] = \mathsf{C}) \equiv \mathsf{invOnXC}(2)$. We model-check a concrete instance of the protocol with three nodes to find an invariant for this instance; while this is not exact, it at least gives us a likely candidate for a general invariant. We conclude that the latter formula is sufficient, so we store the invariant dependency pair $(\mathsf{CR}_{\mathrm{aux}}, \mathsf{newInv}(i_2))$ (with respect to the property $\mathsf{mutualInv}(j_1, j_2)$). The concrete candidate invariant is generalized to $\mathsf{newInv}(i) \equiv \mathsf{invOnXC}(i)$ and added to the set of invariants that need proof.

5.3. A semi-algorithm

In this subsection, we present a semi-algorithm for finding proof dependencies as well as concretized candidates for strengthening. We handle one pair of a concretized requirement req^{c} and a concretized guarded command r^{c} at a time. There is a set *invs* of auxiliary invariants found until now. The main step checks the proof dependencies of $(req^{\mathrm{c}}, r^{\mathrm{c}})$ and finds a new invariant if one is needed.

The algorithm does the following: after computing the weakest precondition $wp^{\mathrm{c}} \equiv \mathsf{WP}(\mathsf{action}(r^{\mathrm{c}}), req^{\mathrm{c}})$, it takes further operations according to the following cases:

(1) If $wp^{\mathrm{c}} \equiv req^{\mathrm{c}}$, meaning that statement $\mathsf{action}(r^{\mathrm{c}})$ does not change req^{c}, then the proof dependency for $(req^{\mathrm{c}}, r^{\mathrm{c}})$ is recorded as $\mathsf{CR}_{\mathrm{presv}}$. At this moment there are no new invariants to be added.
For example, let $r^{\mathrm{c}} = \mathsf{crit}(3)$ and $req^{\mathrm{c}} \equiv \mathsf{mutualInv}(1,2)$. Then $wp^{\mathrm{c}} \equiv req^{\mathrm{c}}$, so $\mathsf{CR}_{\mathrm{presv}}$ will be recorded.
(2) If $\mathsf{guard}(r^{\mathrm{c}}) \rightarrow wp^{\mathrm{c}}$ is found to be a tautology, then no new invariant is created, and the new proof dependency for $(req^{\mathrm{c}}, r^{\mathrm{c}})$ is recorded as $\mathsf{CR}_{\mathrm{estbl}}$. We implemented the tautology check in invFinder by sending the fomula to the SMT solver Z3. For example, let $r^{\mathrm{c}} = \mathsf{crit}(2)$, $req^{\mathrm{c}} \equiv \mathsf{invOnXC}(1)$, Then $wp^{\mathrm{c}} \equiv \neg(\mathsf{false} = \mathsf{true} \wedge a[1] = \mathsf{C}) \Longleftrightarrow \mathsf{true}$. Obviously, $\mathsf{guard}(r^{\mathrm{c}}) \rightarrow wp^{\mathrm{c}}$ holds, so $\mathsf{CR}_{\mathrm{estbl}}$ will be recorded.
(3) If neither of the above two cases holds, then a new auxiliary invariant *newInv* will be constructed, which will make the proof relation $\mathsf{CR}_{\mathrm{aux}}$ hold. We first construct $\neg \bigwedge_{n=1}^{k} f_n^{\mathrm{c}} \Longleftrightarrow \neg\mathsf{guard}(r^{\mathrm{c}}) \vee wp^{\mathrm{c}}$ and then consider all its subformulas starting from smaller ones: $\neg f_1$,

Table I: A fragment of the output of invFinder

concrete invariant	concrete guarded command	proof dependency
mutualInv(1,2)	crit(1)	CR_{aux}, invOnXC(2)
	crit(2)	CR_{aux}, invOnXC(1)
	crit(3)	$\mathsf{CR}_{\text{presv}}$
invOnXC(1)	crit(1)	$\mathsf{CR}_{\text{estbl}}$
	crit(2)	$\mathsf{CR}_{\text{estbl}}$

$\neg f_2, \ldots; \neg(f_1 \wedge f_2), \neg(f_1 \wedge f_3), \ldots; \neg(f_1 \wedge f_2 \wedge f_3), \ldots$ For each subformula, we guess whether it is an invariant. This can be implemented by sending the formula to some model checker, which checks whether a small reference instance of the protocol satisfies it. While this may be unsound (there may be larger instances that violate the subformula), it at least produces a likely candidate auxiliary invariant. (This is why the output of invFinder will need to be checked by Isabelle later. We have never encountered a case where Isabelle would not succeed in our examples; we will come back to this point in the conclusion.) Our implementation calls the model checker NuSMV, except if the latter cannot find the set of reachable states RS; then it calls Murphi. After choosing the auxiliary invariant, we have to *generalize* it. That means, we change its concrete process identifiers to parameter variables $i_1, i_2, \ldots$ (For details, see the next section.) After generalization, the auxiliary invariant is added to the set of requirements that need to be verified.

This main step will be repeated until proof dependencies for all pairs (req^{c}, r^{c}) have been checked or a pair (req^{c}, r^{c}) is found for which no proof dependency applies. The former terminates the search for auxiliary invariants successfully, while the latter indicates that the requirement req^{c} cannot be handled by our method, likely because it is not actually an invariant. The efficiency of our algorithm to make the searching of invariants converge as soon as possible lies in two aspects: (1) the chosen invariant candidate is a correct invariant in a reference instance; (2) the content of the chosen candidate is as small as possible because we start trying with the smallest candidates.

Example 5.5. We extend Example 5.4 in the previous section. Assume given the concrete requirement mutualInv(1,2), combined with the three concrete guarded commands crit(1), crit(2), and crit(3). The resulting outputs of the cases described above is shown in Table I. In the table, each line records a concrete invariant, a concrete guarded command and the proof dependency. Note that the first line, with proof dependency CR_{aux}, introduces an auxiliary invariant invOnXC(2). Its generalized form, invOnXC(i), is added to the set of candidate invariants, and we immediately concretize it again to check whether it is an inductive invariant. These checks are documented in the last two lines of Table I.

6. GENERALIZING CONCRETE INVARIANTS

Intuitively, generalization means that a concrete formula or guarded command is generalized into a symbolic formula or command. We generalize guarded commands in the context of a requirement and therefore have to add parameter constraints to describe the relations between the parameters of the requirement and those of the guarded command.

The simple case is to generalize a requirement formula, executed when an auxiliary invariant inv^{c} has been found. As we always assumed that different parameters are assigned different node identities, we can define one parameter for each node identity that appears in inv^{c}. We choose parameters $i_1, i_2, \ldots$ in the order in which node identities appear in the formula from left to right, so that the generalization of $inv(1,2)$ becomes the same as the generalization of $inv(4,3)$.

The second kind of generalization is to generalize a guarded command r^{c} in the context of a requirement $req(i_1, i_2, \ldots, i_m)$ and its concretization $req(1, 2, \ldots, m)$. Again, we can use the assumption that different parameters are assigned different node identities. The guarded

Table II: The result of generalizing lines in Table I

invariant	guarded command	parameter constraint	proof dependency
$\mathsf{mutualInv}(i_1,i_2)$	$\mathsf{crit}(j)$	$j = i_1$	$\mathsf{CR_{aux}}, \mathsf{invOnXC}(i_2)$
		$j = i_2$	$\mathsf{CR_{aux}}, \mathsf{invOnXC}(i_1)$
		$j \neq i_1 \wedge j \neq i_2$	$\mathsf{CR_{presv}}$
$\mathsf{invOnXC}(i)$	$\mathsf{crit}(j)$	$j = i$	$\mathsf{CR_{estbl}}$
		$j \neq i$	$\mathsf{CR_{estbl}}$

command is a concretization of a general form $r(j_1, j_2, \ldots, j_\ell)$. Whenever we see that some parameter j has been instantiated by the node identity $n \in \{1,2,\ldots,m\}$, we add the parameter constraint $j = i_n$. When, however, the parameter j has been instantiated with some value > m, we add the parameter constraint $\bigwedge_{n=1}^{m} j \neq i_n$. The conjunction of all parameter constraints is added to the information that is passed to proofGen.

Example 6.1. We show how to generalize the information from Table I. In the first line of that table, we have concrete invariant $\mathsf{mutualInv}(1,2)$, which is generalized to $\mathsf{mutualInv}(i_1,i_2)$. The guarded command in that table line is $\mathsf{crit}(1)$, which is generalized to $\mathsf{crit}(j)$. Because the parameter value for j in the concretisation is equal to the concrete value for i_1, we add the parameter constraint $j = i_1$. The proof dependency $\mathsf{CR_{aux}}$ remains the same, but the associated auxiliary invariant $\mathsf{invOnXC}(2)$ is generalized to $\mathsf{invOnXC}(i_2)$ with the parameter chosen based on the concrete value.

The other lines in Table I are handled similarly. The result of all generalizations is shown in Table II.

7. APPLICATION TO FLASH

The FLASH protocol is a publicly recognized challenging benchmark in the field of parameterized verification. The first full verification of safety properties of FLASH was achieved by Park and Dill [1996], who proved interactively the safety properties of FLASH using PVS; in particular, they provided the needed inductive invariants manually. Chou et al. [2004] adopted parameter abstraction and guard strengthening to verify safety properties of FLASH. Talupur and Tuttle [2008] extended this work by deriving high-quality invariants from message flows and using these invariants to accelerate the above method. McMillan [2001] applied compositional model checking and used Cadence SMV to the verification of both safety and liveness properties of FLASH. Conchon et al. [2012; 2013] have applied the tool Cubicle to the verification FLASH, which is theoretically based on SMT model checking techniques.

In most of the treatments mentioned above, auxiliary invariants are provided manually depending on deep human insight into the FLASH protocol. Only Cubicle [Conchon et al. 2012] automatically found auxiliary invariants; it searches auxiliary invariants backward by a heuristics-guided algorithm with the help of an oracle (a reference instance of the protocol). Conchon et al. [2015] produce Why3-proof certificates for SMT-solvers, not for general-purpose theorem prover like Isabelle or Coq. However, they proved the induction $\bigwedge_{j=1}^{n} inv_j(X) \wedge \tau(X,X') \Longrightarrow \bigwedge_{j=1}^{n} inv_j(X')$ as one monolithic formula, without making use of proof dependencies or similar information. (Here X is the set of state variables and $\tau(X,X')$ is the transition relation. In our terminology, $\bigwedge_{j=1}^{n} inv_j(X)$ is an inductive invariant.) Such a proof obligation is quite big and complex, so the SMT solver needs a strong ability to find suitable instances (corresponding to our parametric constraints) of the quantified formulas.

Thanks to the three kinds of proof dependency relation, our method allows to decompose this big induction into smaller proofs for single proof dependencies, which are communicated as hints to the theorem prover. Therefore, the proof is easy and does not need any help from Isabelle's sledgehammer or SMT solvers in the background.

We summarize Park and Dill [1996]'s description of the FLASH protocol: It is a directory-based cache coherence protocol which supports a large number of distributed processing n-

odes. Each node contains a processor, caches, and a portion of the main memory. The distributed nodes communicate with each other through asynchronous messages in a point-to-point network. The state of a cached copy is in either *invalid, shared* (readable), or *exclusive* (readable and writable). Each cache line-sized block in memory is associated with one directory header which keeps information about the cache line. For a main memory address, the node on which that piece of memory is physically located is called the home; the other nodes are called remote. The home node maintains all the information about its portion of main memory in the corresponding directory headers. The "three-hop flow" occurring in a READ-transaction is the most innovative feature of the FLASH protocol: a remote node can transfer modified cached data directly to another remote node without detour through the home node (so the request-and-reply cycle takes three instead of four hops); the first node informs the home in a "sharing write-back" message about the modified data and sharing status.

We reused and adapted the formalisation of FLASH by Conchon et al. [2013] for Murphi. It models the state variables of the home processor indirectly by global variables such as *A_home*. In contrast, Chou et al. [2004]'s formalisation models the home processor just as one of the processors, so some array entry *A*[*Home*] contains the state variable corresponding to the home processor. We follow the modelling idea of Conchon et al. [2013] because this allows us to treat indices of array variables fully symmetrically, which enables our generalization strategy. However, the FLASH protocol with data paths is complex and NuSMV is unable to handle candidate invariants with data paths because NuSMV cannot generate a BDD for the reachable state set of (an instance of) the FLASH protocol with data paths. for example, an instance with four nodes (including the home node) and one bit of main memory in the home node is still too complex to compute the set of reachable states of the full model with data paths (on a parallel computing server with 384 GiB of RAM memory). Adding one bit of main memory also implies that every processor has a bit of cache content, the home processor may have another copy of the data, and messages between nodes also carry data, so overall there would be about 10 additional bits in the state space per bit of main memory. In fact, this problem of state space explosion is the main obstacle for the application of existing model-checking based solutions to the parameterized verification of FLASH. For instance, the reachable state set of a protocol instance is required in existing work: Arons et al. [2001] need the reachable state set to compute the so-called "invisible inductive invariants" for deductive theorem proving, while Lv et al. [2007] need it to strengthen the guards of rules for a counterexample-guided refinement of an abstract protocol. Exactly because the reachable state set for a proper instance of the FLASH protocol is not available, these two approaches fail to verify FLASH.

In order to overcome the obstacle of constructing the full state space, invFinder provides techniques to simplify protocols by data abstraction and allows to try another model checker on the abstract protocol. Based on the Murphi model of FLASH, we constructed a simplified NuSMV-model: we abstracted (or removed) all the variables and operations on data. Figure 2 illustrates our data abstraction by comparing the type UNI_MSG and guarded command "NI_Remote_PutX" in the original and simplified versions. If a guarded command only deals with data, it can be omitted in the simplified protocol.

Our tool invFinder accepts a command line option to indicate a simplified (abstract) set of guarded commands and to indicate that another tool (NuSMV) should be called to generate its reachable state set and judge a candidate invariant formula. Here we emphasize that the reachable state set of the simplified FLASH protocol instance with four nodes (including the home node) and no main memory can be generated by NuSMV in the aforementioned computing server. This is sufficient for invFinder to find all the invariant formulas without data.

However, if variables on data occur in a candidate formula, we also need to guide invFinder to deal with it. For instance, "Sta.Dir.Dirty = **false** & !Sta.MemData = Sta.CurrData" cannot be simply judged via the simplified NuSMV-model; as the variable "Sta.MemData"

```
UNI_MSG: record
  Cmd: UNI_CMD;
  Proc: NODE;
  HomeProc: boolean;
  Data: DATA;
end;

ruleset dst: NODE do
  rule "NI_Remote_PutX"
    Sta.UniMsg[dst].Cmd = UNI_PutX &
    Sta.Proc[dst].ProcCmd = NODE_GetX
  ==>
    begin
      Sta.UniMsg[dst].Cmd := UNI_None;
      Sta.Proc[dst].ProcCmd := NODE_None;
      Sta.Proc[dst].InvMarked := false;
      Sta.Proc[dst].CacheState := CACHE_E;
      Sta.Proc[dst].CacheData := Sta.UniMsg[dst].Data;
    end;
  endrule;
endruleset;
```

(a) original model with data

```
UNI_MSG: record
  Cmd: UNI_CMD;
  Proc: NODE;
  HomeProc: boolean;

end;

ruleset dst: NODE do
  rule "NI_Remote_PutX"
    Sta.UniMsg[dst].Cmd = UNI_PutX &
    Sta.Proc[dst].ProcCmd = NODE_GetX
  ==>
    begin
      Sta.UniMsg[dst].Cmd := UNI_None;
      Sta.Proc[dst].ProcCmd := NODE_None;
      Sta.Proc[dst].InvMarked := false;
      Sta.Proc[dst].CacheState := CACHE_E;

    end;
  endrule;
endruleset;
```

(b) simplified model without data

Fig. 2: Data abstraction in FLASH

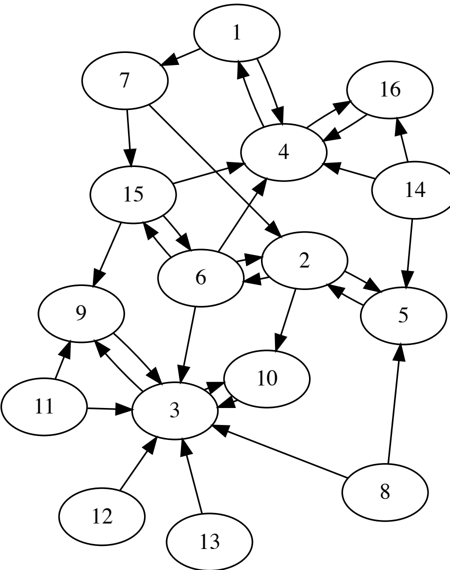

Fig. 3: Invariant dependencies between the first 16 invariants

is undefined, NuSMV will report an error. At this moment, invFinder calls Murphi. Murphi walks through a full protocol instance (for this formula, with four nodes and one bit of data) and decides on the formula containing data variables. Here we adopt a heuristic: if Murphi has been running to check the property up to a time-out without finding a counter-example, then the checked formula will be regarded as an invariant. This is another point where our method relies on heuristics, but this is not surprising because verification of the protocols with guarded choices is undecidable [Bloem et al. 2016]. The heuristic may introduce a false invariant, but as the method anyway only produces an invariant candidate (recall that the formula may be false in a larger instance), this need not bother us now; Isabelle will provide a guaranteed-sound proof only if it exists. If such a false invariant is generalized and occurs in the parameterized invariant set, the generated proof script cannot be checked in Isabelle, but we did not encounter this situation in FLASH or in any other example.

We have successfully verified the FLASH protocol. Our verification product is a formal proof with a set of inductive invariants. To the best of knowledge, this work for the first time automatically generates a proof of safety properties of a full version of FLASH in a theorem prover without manual provision of auxiliary invariants. Besides, the invariants, being the shortest sufficient ones that have been found, have a clean and neat semantics; they gave us deep insight into the formalisation of FLASH, which we copied from [Conchon et al. 2013] with only superficial modifications.

In the FLASH protocol, invFinder constructed a total of 162 invariants with several thousands of proof dependencies CR_{aux} between them. Often, there are multiple dependencies between the same pair of invariants, based on a case distinction. As an illustration, we show the dependencies between the first 16 invariants in Fig. 3. The graph contains an arrow $inv1 \longrightarrow inv2$ whenever at least one proof that the post-state of some guarded command satisfies $inv2$ uses proof dependency CR_{aux} with the auxiliary invariant $inv1$ in the pre-state of the transition. Cycles in this graph do not mean that our proof is circular.

For our verification of Flash, we extended our modelling language by adding if–then–else (ite) expressions, and we extended our proof dependency relations to handle the corresponding case distinctions, by an additional constraint similar to the parameter constraint in Table II. Let's see an example in Fig. 4. Here a **for** statement is used to express that all the nodes in this protocol update their state variables synchronously. In the body of the **for** loop, there is an **if** statement. We first rewrite the **if** statement to ite expressions of the form:

```
Sta.Dir.InvSet[p] := ite(... | (Sta.Dir.HeadVld & ... & !Sta.Dir.HomeHeadPtr), true, false);
Sta.InvMsg[p].Cmd := ite(... | (Sta.Dir.HeadVld & ... & !Sta.Dir.HomeHeadPtr),
                                                                  INV_Inv, INV_None);
```

Then, we distinguish cases depending on whether a state satisfies the condition in the ite expression or not. For example, for invariant 60, we have a CR_{aux} dependency in states that satisfy Sta.Dir.HeadVld & ... & !Sta.Dir.HomeHeadPtr, and we have a $\mathsf{CR}_{\text{estbl}}$ dependency otherwise. This is noted as

$$\mathsf{CR}_{\text{aux}}(inv60[i], \text{NI_Local_GetX_PutX_10_Home}[j], inv14[i], i \neq j \wedge \text{Sta.Dir.HeadVld} \,\&\, \ldots)$$
$$\mathsf{CR}_{\text{estbl}}(inv60[i], \text{NI_Local_GetX_PutX_10_Home}[j], i \neq j \wedge \neg(\text{Sta.Dir.HeadVld} \,\&\, \ldots))$$

Accordingly, invFinder may create a finer-grained case distinction for one (invariant, guarded command) pair. As these new case distinctions can be handled similarly to parameter constraints, they pose no significant difficulty.

8. OTHER EXPERIMENTS

We have implemented our approach in a prototype paraVerifier [Li et al. 2015] and used it to check some other examples: typical bus-snoop benchmarks such as MESI and MOESI, as well as directory-based benchmarks such as German. The MESI protocol (also known as the Illinois protocol) is an invalidate-based cache coherence protocol, and it is one of the

```
ruleset src: NODE do
  rule "NI_Local_GetX_PutX_10_Home"
    Sta.UniMsg[src].Cmd = UNI_GetX & Sta.UniMsg[src].HomeProc & !Sta.Dir.Dirty &
    Sta.Dir.HeadVld & Sta.Dir.HeadPtr = src & !Sta.Dir.HomeHeadPtr & Sta.Dir.HomeShrSet &
    !Sta.Dir.Pending & !Sta.Dir.Local
  ==>
    begin
      for p: NODE do
        if p != src & ((Sta.Dir.ShrVld & Sta.Dir.ShrSet[p]) |
                  (Sta.Dir.HeadVld & Sta.Dir.HeadPtr = p & !Sta.Dir.HomeHeadPtr)) then
          Sta.Dir.InvSet[p] := true;
          Sta.InvMsg[p].Cmd := INV_Inv;
        else
          Sta.Dir.InvSet[p] := false;
          Sta.InvMsg[p].Cmd := INV_None;
        endif;
        Sta.Dir.ShrSet[p] := false;
      endfor;
      Sta.Dir.Pending := true;
      ...
      Sta.UniMsg[src].Data := Sta.MemData;
    end;
  endrule;
endruleset;
```

Fig. 4: Extension of modelling language in FLASH

most common protocols which support write-back caches [Papamarcos and Patel 1984]. Each cache line is in one of four states, which form the letters of the acronym MESI. It requires cache-to-cache transfer on a miss if the block resides in another cache. MOESI is a cache coherence protocol implemented in the AMD64 architecture [Devices 2013]. In addition to the four common MESI protocol states, there is a fifth "Owned" state representing data that is both modified and shared. This avoids the need to write modified data back to main memory before sharing it. The German protocol is a directory-based cache coherence protocol, which was posted as a challenge to the formal verification community by German [2004]. Germanish is a simplified version of German.

The detailed code and experiment data can be found at the website of our tool [Li and Duan 2016]. Each experiment directory includes the input files to paraVerifier, the invariant sets, and the Isabelle proof scripts. We ran paraVerifier to perform these experiments on a server with an Intel Xeon processor, 8 GiB memory and 64-bit Linux 3.15.10. NuSMV and Murphi were executed on a computing server with 32 Intel Xeon processors, 384 GiB memory and 64-bit Linux 2.6.32. Experiment results are summarized in Table III. The #invariants column contains the total number of invariants (original and auxiliary). The time column shows the wall clock time from the start of invFinder until the tool reported the successful verification of the found inductive invariant. The memory columns separately show the memory used by the client (running invFinder) and the computing server (running NuSMV and Murphi to find invariant candidates).

9. CONCLUSION AND FUTURE WORK

Our approach to parameterized verification is the first one that successfully applied a theorem prover (Isabelle) without human intervention to verify an industry-scale parameterized protocol, namely the FLASH protocol. Especially, the focus of our approach is on making the theorem-proving based verification of parameterized protocols automatic.

Table III: Verification results on benchmarks

Protocol	#commands	#invariants	time (s)	memory (MB)	
				client	server
mutualEx	4	5	3.25	7.3	—
MESI	4	3	2.47	11.5	84
MOESI	5	3	2.49	13.2	85
Germanish	6	3	2.9	7.8	90
German	13	52	38.67	14	135
FLASH_nodata	60	152	280	26	2.400
FLASH_data	62	162	510	26	4.800

To achieve this goal, we made the following contributions. First, we distinguished three kinds of proof dependencies as special cases of the induction proof rule to provide the theorem prover with additional information on what tactics to use. Second, we automatically distinguish the relevant cases for each (invariant, guarded command) pair, depending on the relations between parameters and possibly on whether the state satisfies a condition used in an ite expression. Third, we guess sensible candidate invariants using a model checker to ensure that the candidate holds at least in a small concrete instance. Fourth, to deal with large models like the FLASH protocol, paraVerifier uses a combination of a concrete and an abstract model to accelerate this checking of candidate invariants.

As we demonstrated in this work, we found an effective way to automate the (traditionally, interactive) process of formal theorem proving. Interactive proving is not scalable to complex protocols like FLASH even if we already have the set of inductive invariants and the proof dependencies. We can run Isablle to check these proof scripts in batch mode (not the interactive mode: the proof is too large). Checking the proof scripts of FLASH needs about 14 hours. Theorem proving can guarantee the rigorousness of the verification results, while automation releases the burden of human interaction.

In the future, we will extend our work in the following three directions: (1) we want to deal with general safety properties (not only invariant properties) and liveness properties, which also play an important role in realistic protocols to guarantee that good things will eventually happen; (2) we want to develop more powerful techniques to find invariants by adopting machine learning to tackle the obstacle of constructing a SMV problem instance. For instance, for a large protocol, we can gather practical test data and obtain knowledge from the data by machine learning, and use the knowledge to decide whether a guessed candidate is an invariant; (3) While we never encountered it in our examples, invFinder provides no strict guarantee that the resulting set of auxiliary invariant candidates provides an inductive invariant. If Isabelle, on another example, would be unable to prove the correctness of some invariant, it could point out which invariant is problematic. Then, invFinder could backtrack to that invariant and try to find a different invariant from the subformulas of $\neg \bigwedge_{n=1}^{k} f_n$. Adding the automation of this loop seems possible, but it would require us to find interesting examples where this phenomenon occurs.

REFERENCES

Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. 2013. All for the Price of Few. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 476–495.

Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. 2001. Parameterized Verification with Automatically Computed Inductive Assertions?. In *Computer Aided Verification: CAV (LNCS)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.), Vol. 2102. Springer, Berlin, 221–234. DOI: http://dx.doi.org/10.1007/3-540-44585-4_19

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Pr., Cambridge, Mass..

Kai Baukus, Yassine Lakhnech, and Karsten Stahl. 2002. Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In *Verification, Model Checking, and Abstract Interpretation: VMCAI (LNCS)*, Agostino Cortesi (Ed.), Vol. 2294. Springer, Berlin, 317–330. DOI:http://dx.doi.org/10.1007/3-540-47813-2_22

Nikolaj Bjørner, Anca Browne, and Zohar Manna. 1997. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science* 173, 1 (1997), 49–87. DOI:http://dx.doi.org/10.1016/S0304-3975(96)00191-0

Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2016. Decidability in Parameterized Verification. *SIGACT News* 47, 2 (June 2016), 53–64. DOI:http://dx.doi.org/10.1145/2951860.2951873

Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Formal Methods in Computer-Aided Design: FMCAD (LNCS)*, Alan J. Hu and Andrew K. Martin (Eds.), Vol. 3312. Springer, Berlin, 382–398. DOI:http://dx.doi.org/10.1007/978-3-540-30494-4_27

Edmund M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. MIT Pr., Cambridge, Mass..

Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. 2012. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems. In *Computer Aided Verification: CAV (LNCS)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, Heidelberg, 718–724. DOI:http://dx.doi.org/10.1007/978-3-642-31424-7_55

Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. 2013. Invariants for Finite Instances and Beyond. In *FMCAD 2013: Formal Methods in Computer-Aided Design*. IEEE, [s. l.], 61–68. DOI:http://dx.doi.org/10.1109/FMCAD.2013.6679392

Sylvain Conchon, Alain Mebsout, and Fatiha Zaïdi. 2015. Certificates for Parameterized Model Checking. In *The 20th International Symposium on Formal Methods, June 24-26, 2015, Proceedings (FM 05)*. 126–142. DOI:http://dx.doi.org/10.1007/978-3-319-19249-9_9

Advanced Micro Devices. 2013. AMD64 Architecture Programmer's Manual. (2013).

Michael Dooley and Fabio Somenzi. 2016. Proving Parameterized Systems Safe by Generalizing Clausal Proofs of Small Instances. In *Computer Aided Verification: CAV (LNCS)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, [s. l.], 292–309. DOI:http://dx.doi.org/10.1007/978-3-319-41528-4_16

E. Allen Emerson and Vineet Kahlon. 2003. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In *12th Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings (CHARME 03)*. Springer-Verlag, Berlin, Heidelberg, 247–262. DOI:http://dx.doi.org/10.1007/978-3-540-39724-3_22

Steven M. German. 2004. Tutorial on verification of distributed cache memory protocols. In *5th International Conference on Formal Methods in Computer-Aided Design*. IEEE, Austin, Texas, USA.

Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. 2008. Towards SMT Model Checking of Array-Based Systems. In *The 4th International Joint Conference on Automated Reasoning (IJCAR '08)*. Springer-Verlag, Berlin, Heidelberg, 67–82. DOI:http://dx.doi.org/10.1007/978-3-540-71070-7_6

Silvio Ghilardi and Silvio Ranise. 2010. Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. 6 (10 2010).

Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic Cutoff Detection in Parameterized Concurrent Programs. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 645–659. DOI:http://dx.doi.org/10.1007/978-3-642-14295-6_55

Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. 1994. The Stanford FLASH multiprocessor. In *Computer Architecture [ISCA]*. IEEE Comp. Soc. Pr., Los Alamitos, Calif., 302–313. DOI:http://dx.doi.org/10.1109/ISCA.1994.288140

Y. Li and K. Duan. 2016. The software tool ParaVerifier. (2016). Available from https://github.com/paraVerifier/paraVerifier.

Yongjian Li, Kaiqiang Duan, Yi Lv, Jun Pang, and Shaowei Cai. 2016. A novel approach to parameterized verification of cache coherence protocols. In *34th IEEE International Conference on Computer Design (ICCD)*. IEEE, Piscataway, NJ, 560–567. DOI:http://dx.doi.org/10.1109/ICCD.2016.7753341

Yongjian Li, Jun Pang, Yi Lv, Dongrui Fan, Shen Cao, and Kaiqiang Duan. 2015. ParaVerifier: An automatic framework for proving parameterized cache coherence protocols. In *Automated Technology for Verification and Analysis: ATVA (LNCS)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.), Vol. 9364. Springer, Cham, 207–213. DOI:http://dx.doi.org/10.1007/978-3-319-24953-7_15

Yi Lv, Huimin Lin, and Hong Pan. 2007. Computing Invariants for Parameter Abstraction. In *Fifth ACM & IEEE International Conference on Formal Methods and Models for Co-Design: MEMOCODE '07*. IEEE, Piscataway, NJ, 29–38. DOI:http://dx.doi.org/10.1109/MEMCOD.2007.371252

K. L. McMillan. 2001. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Correct Hardware Design and Verification Methods: CHARME (LNCS)*, Tiziana Margaria and Tom Melham (Eds.), Vol. 2144. Springer, Berlin, 179–195. DOI:http://dx.doi.org/10.1007/3-540-44798-9_17

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer, Berlin. DOI:http://dx.doi.org/10.1007/3-540-45949-9

Sudhindra Pandav, Konrad Slind, and Ganesh Gopalakrishnan. 2005. Counterexample Guided Invariant Discovery for Parameterized Cache Coherence Verification. In *Correct Hardware Design and Verification Methods: CHARME (LNCS)*, Dominique Borrione and Wolfgang Paul (Eds.), Vol. 3725. Springer, Berlin, 317–331. DOI:http://dx.doi.org/10.1007/11560548_24

Mark S. Papamarcos and Janak H. Patel. 1984. A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. *SIGARCH Comput. Archit. News* 12, 3 (Jan. 1984), 348–354. DOI:http://dx.doi.org/10.1145/773453.808204

Seungjoon Park and David L. Dill. 1996. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *SPAA '96: ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, 288–296. DOI:http://dx.doi.org/10.1145/237502.237573

Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems: TACAS (LNCS)*, Tiziana Margaria and Wang Yi (Eds.), Vol. 2031. Springer, Berlin, 82–97. DOI:http://dx.doi.org/10.1007/3-540-45319-9_7

Amir Pnueli and Elad Shahar. 1996. A platform for combining deductive with algorithmic verification. In *Computer Aided Verification: CAV (LNCS)*, Rajeev Alur and Thomas A. Henzinger (Eds.), Vol. 1102. Springer, Berlin, 184–195. DOI:http://dx.doi.org/10.1007/3-540-61474-5_68

Murali Talupur and Mark R. Tuttle. 2008. Going with the Flow: Parameterized Verification Using Message Flows. In *The 2008 International Conference on Formal Methods in Computer-Aided Design (FMCAD '08)*. IEEE Press, Piscataway, NJ, USA, 10:1–10:8.

A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. 2001. A Technique for Invariant Generation. In *Tools and Algorithms for the Construction and Analysis of Systems: TACAS (LNCS)*, Tiziana Margaria and Wang Yi (Eds.), Vol. 2031. Springer, Berlin, 113–127. DOI:http://dx.doi.org/10.1007/3-540-45319-9_9

$\mathsf{CR}_{\mathrm{estbl}}(inv,r,case1)$	moreover { assume b1 : *case1* have CR1 *r inv* proof (cut_tac a1 a2 b1, auto) qed then have CRS *r inv* (invariants N) by auto }
$\mathsf{CR}_{\mathrm{presv}}(inv,r,case2)$	moreover { assume b1 : *case2* have CR2 *r inv* proof (cut_tac a1 a2 b1, auto) qed then have CRS *r inv* (invariants N) by auto }
$\mathsf{CR}_{\mathrm{aux}}(inv,r,F,case3,f')$	moreover { assume b1 : *case3* have CR3 *r inv* (invariants N) proof (cut_tac a1 a2 b1, simp, rule_tac x = f' in exI, auto) qed then have CRS *r inv* (invariants N) by auto }

Fig. 5: The transformation from the symbolic proof dependencies to a proof

A. PROOFGEN: AUTOMATIC PROOF GENERATION

A.1. A base theory `paraTheory.thy`

The Isabelle proof script fed into Isabelle consists of two parts: (1) a base theory `paraTheory.thy` to formalize preliminary definitions for values, variables, states, rules, protocols, and reachable state set, weakest preconditions, all kinds of proof dependency relations, and consistency lemma. `paraTheory.thy` provides a public library for protocol analysis. It is manually done once, and will be imported in each case study. This theory comprises 1125 lines. Its execution in Isabelle needs only 31 seconds. (2) Theories for a case study like the simple mutual exclusion or FLASH protocols. They are automatically generated by proofGen, which will be illustrated in the remainder of this section.

A.2. From proof dependencies to a formal proof

In this section, we first focus on the core of the proof generation: generation of a lemma on the existence of the proof dependency relation between a parameterized rule and invariant formula based on the aforementioned generalized and symbolic dependency relations in Table II. Let us show an example lemma $\mathtt{critVsinv}_1$, which specifies and proves the existence of the proof dependency relation between $\mathsf{crit}(j_1)$, and $\mathsf{mutualInv}(i_1,i_2)$.

```
1 lemma critVsinv1:
2 assumes
  a1: ∃j1.j1 ≤ N ∧ r = crit j1 and
  a2: ∃i1i2.i1 ≤ N ∧ i2 ≤ N
  ∧i1 ≠ i2 ∧ f = inv1 i1 i2
3 shows CRS s f r (invariants N)
4 proof -
  from a1 obtain j1 where a1: j1 ≤ N ∧ r = crit j1
  by blast
  from a2 obtain i1 i2 where a2: i1 ≤ N
       ∧i2 ≤ N ∧ i1 ≠ i2 ∧ f = inv1 i1 i2
  by blast
5 have j1 = i1 ∨ j1 = i2 ∨ (j1 ≠ i1 ∧ j1 ≠ i2)
  by auto
6 moreover {assume b1: j1 = i1
7   have CR3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x = ¬(x = true ∧ a[i2] = C) in exI, auto) qed
8   then have CR s f r (invariants N) by auto}
9 moreover {assume b1: j1 = i2
10  have CR3 s f r (invariants N)
    proof(cut_tac a1 a2 b1, simp, rule_tac x = ¬(x = true ∧ a[i1] = C) in exI,auto) qed
11  then have CR s f r (invariants N) by auto}
12 moreover {assume b1: (j1 ≠ i1 ∧ j1 ≠ i2)
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
```

In the above lemma, 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 `j1` with which `r` is a rule obtained by instantiating `crit`; (2) assumption `a2` specifies that there exist actual parameters `i1` and `i2` with which `f` is a formula obtained by instantiating `inv1`. Line 3 is the proof goal to show CRS $s\ f\ r$ (invariants N).

Its proof is a typically case analysis. Line 4 shows two typical proof patterns forward-style which fixes local variables such as `j1` and new facts such as $\texttt{a1} : \texttt{iR1} \le \texttt{N} \wedge \texttt{r} = \texttt{critj1}$. 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 `j1` with `i1` and `i2`, which is in fact decided by the three cases listed in the three lines of the Table II. Lines 6–14 contain three `moreover` branches, which prove these sub cases one by one: Lines 6–8 prove the case where $\texttt{iR1} = \texttt{i1}$: Line 6 lists the assumption of the case $\texttt{j}_1 = \texttt{i}_1$, which is from the cloumn *case*; Line 7 first proves that the proof dependency relation CR_3 holds by supplying a symbolic formula invOnXC(i_2), which is from the column f' in the first line of Table II; From the conclusion at Line 7, Line 8 furthermore proves the proof dependency relation CRS holds. Lines 9–11 proves the case where $\texttt{iR1} = \texttt{i2}$, proof of which is similar to that of case 1; Lines 12–14 are for the case where neither $\texttt{iR1} = \texttt{i1}$ nor $\texttt{iR1} = \texttt{i2}$. Each proof of a sub case is done in a block $\texttt{moreoverb1} : \texttt{asm1proof1}$, the `ultimately` proof command in Line 15 concludes by summing up all the subcases.

The core parts of the generation of a lemma such as $\texttt{critVsinv}_1$ are as follows:

— Generation of proof command of case splitting like line 5, which can be derived by collecting all cases on the symbolic proof dependency relation between `crit` and $\texttt{inv}_1$ in Table II.
— Generation of each subproof commands for each sub cases. From the table of symbolic proof dependency relations like Table II, the generation strategy of the above proof can be illustrated intuitively in Fig. 5. The strategy transforms a line of the symbolic proof dependency relation into a paragraph of the subproof. In such a subproof, it chooses a proper subgoal

to prove among CR_1, CR_2 and CR_3, according to the kind of the proof dependency relation, then prove the goal CRS. For CR_1 and CR_2, the transformation is rather straightforward, otherwise the form f' is provided in the `exI` [2] tactic to instantiate the schema variable "?x" in order to tell Isabelle the formula which should be used to construct the CR_3 relation.

A.3. The structure of generated proofs

A formal model of a protocol in a theorem prover like Isabelle includes two parts: (1) the definitions of parameterized rules and invariants and initial states; (2) lemmas, and their proofs. Readers can refer to [Li and Duan 2016] for detailed illustration of the formal proof script. In the following discussion, we abbreviate a parameterized protocol instance with size N as N-parameterized instance.

A.3.1. Definitions. The part of definitions is as follows:

(1) Definitions of formally parameterized invariant formulas, which are generalized from concrete invariants. An actual parameterized invariant in a N-parameterized instance can be obtained by instantiating a formal invariant formula with symbolic indexes with model restriction. All actual invariant formulas in the N-parameterized are defined by a set `invariants N`;
(2) Definitions of formally parameterized rules which can be directly transformed from the internal model which is in turn compiled from Murphi rules of FLASH protocol. Actual parameterized rules can be defined similarly. All actual invariant formulas in the N-parameterized instance are defined by a set `rules N`;
(3) Definitions of specification of the initial states, which can be directly transformed from internal specification of initial state, which in turn is compiled from the `startstate` part of Murphi's code;

Let us give an example to illustrate the above definitions. A formal definition of a parameterized invariant formula of invOnXE in Isabelle, which is renamed as `inv4` by proofGen, is shown as follow:

```
definition inv4::nat ⇒ formula
inv4 i1 ≡ neg (andForm (eqn (IVar (Ident ''x'') ) (Const true))
              (eqn (IVar (Para ''n'' i1)) (Const E)) )
```

Definition of `pinv4` includes two parts: type and body. This definition is generalized according to the concrete one searched by invFinder. The type of `inv4` is `nat ⇒ formula`, which is decided by the number of actual parameters occurring in the concrete one. The body is simply transformed from the formula $(\neg((x = true) \wedge (n[iInv1] = E)))$ into an Isabelle dialect.

After defining all the formal definitions of parameterized invariant formulas, we can define `invariants N` which is the set of all the actual formulas of invariants in the $N-$parameterized instance, each element of which can be an invariant formula instance.

```
definition invariants::nat ⇒ formula set where:
invariants N≡ {f.
∃i1i2.i1 ≤ N ∧ i2 ≤ N ∧ i1 ≠ i2 ∧ f = inv1 i1 i2 ∨
∃i1.i1 ≤ N ∧ f = inv2 i1 ∨
∃i1i2.i1 ≤ N ∧ i2 ≤ N ∧ i1 ≠ i2 ∧ f = inv3 i1 i2∨
∃i1.i1 ≤ N ∧ f = inv4 i1 ∨
∃i1i2.i1 ≤ N ∧ i2 ≤ N ∧ i1 ≠ i2 ∧ f = inv5 i1 i2  }
```

[2] `exI` is an introduction rule for existential quantifier, and formally is $\frac{P(?x)}{\exists x.P(x)}$

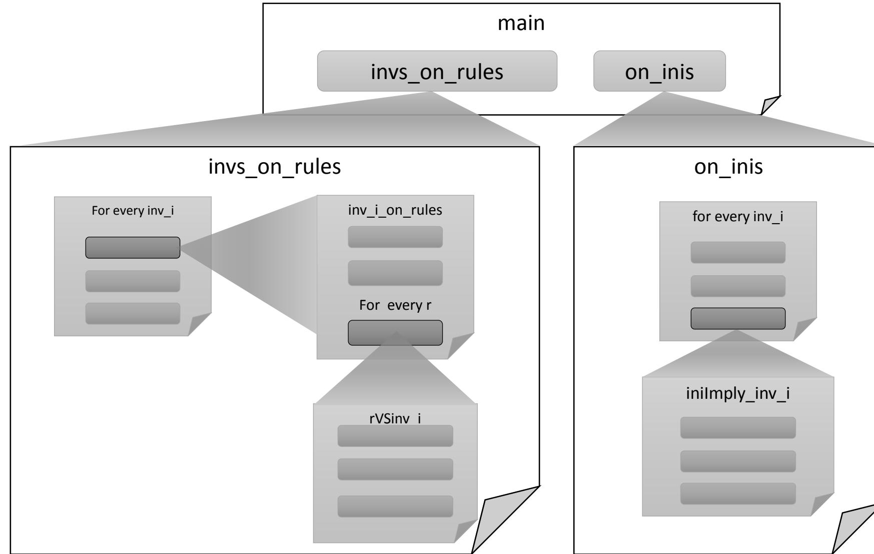

Fig. 6: The structure of the lemmas in an automatically generated proof.

In Isabelle proof script, we adopt the curried form of a function application, so we write $f\ a\ b\ c$, not uncurried form $f(a,b,c)$. In previous section, we adopt the uncurried form for readability. where `invariants N` is a set of invariant formulas to specify properties of a N-parameterized instance: each element in it is obtained by instantiating some formally defined parameterized invariant formula among `inv1 -inv5` with some actual parameters $i_{1-2} \leq N$. `invariants N` can be easily generated if the formal definition of each invariant formula is available. Here the number of parameters of each invariant formula is needed to know.

A.3.2. Lemmas. Roughly speaking, the parts of lemmas are modular and formal and detailed version of some informal paper lemmas listed in Section 5. The Hierarchy of the lemmas in a generated proof script is illustrated in Figure 6. We introduce them in a top-bottom order.

(1) Main lemma proves that any invariant in invariants N holds at any reachable state of the N-parameterized protocol instance. This lemma is according to paper Theorem 5.2. Its proof first applies the main theorema to obtain main subgoals: (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 by applying lemma on_inis. (2) For any invariant $inv \in$ (invariants N), any r in rule set rules N , the proof dependency relation CRS holds. This can be solved by applying lemma `invs_on_rules`.
(2) The lemma `invs_on_rules` specifies and proves the existence of the proof dependency relation CRS for any rule and any invariant. Its proof needs to do case analysis on the form of all invariants, and apply m `inv_i_on_rule`, where $0 < i \leq m$ and m is the number of invariants. For the simple mutual exclusion protocol, $m = 5$.
(3) A lemma such as `inv_i_on_rule` on the proof dependency relation CRS for all rules and an invariant inv_i. Its proof needs to do case analysis on the form of all rules, and call n `r_Vs_inv_i`, where $0 < i \leq n$ and n is the number of rules. For the simple mutual exclusion protocol, $n = 4$.

(4) The lemma such as `r_Vs_inv_i` proves the existence of the proof dependency relation CRS between a rule r and a parameterized invariant inv_i. There are $m \times n$ such lemmas , which is the product of the numbers of rules and invariants. For instance, the aforementioned `critVsinv1` is such a lemma, which we have introduced in detail in Section A.2.
(5) A lemma `on_inits` proves that all invariants hold at the initial state. Its proof needs to do case analysis on the form of all invariants, and apply m lemmas `iniImply_inv_i`, where $0 < i \leq m$.
(6) A Lemma such as `iniImply_inv_i` proves that an invariant `inv_i` holds at the initial state.

Except the lemmas of the fourth kind, the other kinds of lemmas are easy to generate because the bodies of these lemmas and their proofs are in a rather routine pattern. The first kind of lemma is in a fixed pattern. The second and third and fifth kinds of lemmas are typical proof structures of doing case analysis, which are implemented by using proof commands `moreover` and `ultimately`. The sixth kind of lemma can be solved by an `auto` command. We have discussed the generation of the fourth kind in Section A.2.