Mechanizing the CMP Abstraction for Parameterized Verification

YONGJIAN LI, Institute of Software, Chinese Academy of Sciences, China
BOHUA ZHAN, Institute of Software, Chinese Academy of Sciences, China
JUN PANG, University of Luxembourg, Luxembourg

Parameterized verification is a challenging problem that is known to be undecidable in the general case. CMP is a widely-used method for parameterized verification, originally proposed by Chou, Mannava and Park in 2004. It involves abstracting the protocol to a small fixed number of nodes, and strengthening by auxiliary invariants to refine the abstraction. In most of the existing applications of CMP, the abstraction and strengthening procedures are carried out manually, which can be tedious and error-prone. Existing theoretical justification of the CMP method is also done at a high level, without detailed descriptions of abstraction and strengthening rules. In this paper, we present a formally verified theory of CMP in Isabelle/HOL, with detailed, syntax-directed procedure for abstraction and strengthening that is proven correct. The formalization also includes correctness of symmetry reduction and assume-guarantee reasoning. We also describe a tool AutoCMP for automatically carrying out abstraction and strengthening in CMP, as well as generating Isabelle proof scripts showing their correctness. We applied the tool to a number of parameterized protocols, and discovered some inaccuracies in previous manual applications of CMP to the FLASH cache coherence protocol.

CCS Concepts: • **Theory of computation** → **Invariants**; **Program verification**; **Program analysis**; **Automated reasoning**; **Abstraction**.

Additional Key Words and Phrases: Parameterized verification, model checking, theorem proving, Isabelle/HOL, invariants, cache coherence protocols

ACM Reference Format:
Yongjian Li, Bohua Zhan, and Jun Pang. 2024. Mechanizing the CMP Abstraction for Parameterized Verification. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 141 (April 2024), 27 pages. https://doi.org/10.1145/3649858

1 INTRODUCTION

Parameterized concurrent systems, such as cache coherence protocols, have significant applications in many practical areas. Verifying the correctness of such systems has attracted considerable interest from model checking and theorem proving communities [1]. The challenge of parameterized verification is that one needs to check the correctness of the system for an arbitrary number of instances. This has been proved to be an undecidable problem in general [2].

Many approaches have been proposed for parameterized verification in the literature [3, 6, 8, 13, 14, 18, 22, 28, 31, 34, 36, 37, 41–43]. Among them, the CMP method, proposed by Chou, Mannava and Park in [8] in 2004, building on the work of McMillan [28], has been widely used to verify large-scale industrial cache coherence protocols, including Intel's Chipset and FLASH protocols [39].

Authors' addresses: Yongjian Li, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences , Beijing, China; Bohua Zhan, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences , Beijing, China; Jun Pang, Department of Computer Science, University of Luxembourg, Esch-sur-Alzette, Luxembourg.

ACM 2475-1421/2024/4-ART141
https://doi.org/10.1145/3649858

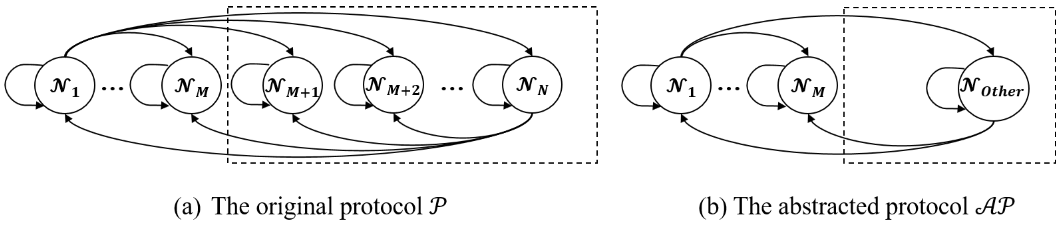

Fig. 1. An illustration of the idea of parameter (node) abstraction in the CMP method: For a parameterized protocol $\mathcal{P}$ with N nodes, the basic idea is to retain M nodes and abstract the remaining nodes as a single node $\mathcal{N}_{other}$. The abstracted protocol is denoted as $\mathcal{AP}$.

A theoretical justification for CMP at a high level is given in [22], but without describing the abstraction procedure in full detail, nor with formalization in a theorem prover.

The central idea of the CMP method can be explained as follows. Let $\mathcal{P}$ be a parameterized protocol, and let $\mathcal{P}(N)$ be the instantiation of the protocol with N nodes. The goal is to verify whether some invariant *Inv* is satisfied by $\mathcal{P}(N)$ for any N. This is achieved using the following procedure:

- A cut-off value M (usually very small, e.g. two or three) is set. For any protocol instance $\mathcal{P}(N)$ with $N > M$ nodes, we can construct an abstract model $\mathcal{AP}$ by abstracting the rules, consisting of M regular nodes and a special *other* node, which gives a conservative abstraction of the original protocol. This is illustrated in Figure 1.
- The abstracted model $\mathcal{AP}$, now with a finite number of states, is verified using a model checker. If model checking produces a counterexample, it may be a real counterexample to the original protocol, or a spurious one due to the abstraction. In the latter case, an auxiliary invariant (also called a *noninterference lemma*), is constructed by analyzing the counterexample (usually by hand). This auxiliary invariant is then used to strengthen the original protocol in order to rule out the counterexample. After strengthening, the protocol is abstracted and model-checked again in another round of CMP, as illustrated in Figure 2. Strengthening in CMP is justified by an assume-guarantee style argument.
- The above iteration of abstraction and strengthening is repeated until either a real counterexample is found, or model checking on the abstracted protocol is successful.

1.1 Related Work

Chou, Mannava, and Park proposed the CMP method informally and demonstrated its application on two examples: the German and FLASH protocols [8]. Much inspiration came from the work of McMillan on compositional model checking of parameterized systems [27, 28]. Krstić gave a formal description of the method at a high level and proved its correctness [22]. The CMP method is further developed by Talupur et al. [39], who proposed a technique for using high-quality invariants derived from message flows. This work also gives a more modular justification of CMP, independent of the abstraction procedure. Abdulla et al. provided an overview of techniques for parameterized verification including the CMP method [10]. In practice, CMP has been used extensively for verifying real-world cache coherence protocols. As an alternative to the exhaustive simulation of concrete protocols, or fully manual proofs by interactive theorem proving, the CMP method can be considered as a lightweight formal approach. On the one hand, it is able to prove correctness for arbitrary number of nodes. On the other hand, the auxiliary invariants that are needed for

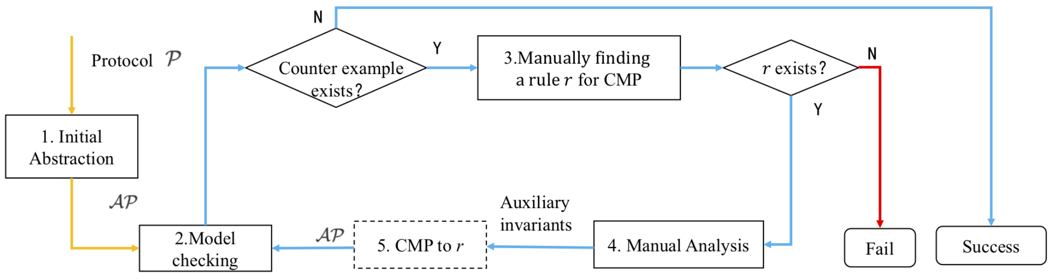

Fig. 2. Illustration of the workflow of CMP. For some parameterized rule r of $\mathcal{P}$, some auxiliary invariants are provided and used to strengthen the guard of r, and then the strengthened rule is abstracted. Usually the CMP procedures are done manually, which is highlighted by a dashed box.

applying CMP are usually much simpler than the fully inductive invariants, and can be more readily provided by system designers.

There are many other methods for parameterized verification. For example, Chen et al. employed a meta-circular assume-guarantee technique to reduce the complexity of verifying finite instances of protocols [6, 7]. There are also many methods that focus on generating invariants automatically, including techniques such as invisible invariants [3, 16, 36], indexed predicates [23], interpolant-based invariant generation [29], and split invariants [11], all of which try to deduce invariants with the aid of model checkers or theorem provers. In particular, Conchon et al. introduced a BRAB algorithm which is implemented in an SMT-based model checker. It computes over-approximations of backward reachable states that are checked to be unreachable in the parameterized system [13]. Li et al. proposed a method to automatically generate auxiliary invariants from a small reference instance of protocols and construct a parameterized formal proof in Isabelle [24]. Padon et al. designed a tool Ivy which graphically shows a counterexample to induction which guides human to find inductive invariants [33]. Padon et al. also proposed a modular proof methodology to produce decidable verification conditions [40]. Decidability greatly improves predictability of proof automation, resulting in a more practical verification approach which is used to verify implementations of distributed protocols.

1.2 Motivation of Our Work

Despite CMP's extensive application in parameterized verification, formalization of CMP, with detailed description of the procedures and proof of correctness in a theorem prover, is still absent. Currently, in most applications of CMP, abstraction and strengthening are conducted by manual editing of the description of the protocol. This can be error-prone for two reasons. First, the theoretical justification of the method may have overlooked small side-conditions and details of the abstraction and strengthening rules. Second, even if the theoretical justification is entirely correct on paper, manual editing of the protocol may not faithfully reflect the theoretical procedure. Another approach to applying the CMP method is to attempt to find auxiliary invariants and carry out abstraction and strengthening completely automatically. In some ways this approach is even more error-prone, as humans are no longer checking whether the steps taken make sense intuitively, and any mistake in the implementation can lead to glaring errors. Considering the above difficulties, it is desirable to have explicit, formally stated rules for strengthening and abstraction, as well as verifying them in a theorem prover. This has been recognized as a first priority in [8, Section 5], and also highlighted as an important research problem in [22].

Unlike predicate abstraction as used in counterexample-guided abstraction refinement [9], parameter abstraction on array-based systems, such as cache coherence protocols, has its own special features and difficulties. An array-based system's state is defined by a global state as well as local states indexed by node. During abstraction to M nodes, all nodes with index $i > M$ will be abstracted into *other*, and all local states on nodes $i > M$ are abstracted away. The main issue then is how to abstract predicates that involve indices and local states for nodes $i > M$. Previous work only identifies a predicate on index i and abstracts it into True or False depending on whether it occurs in a positive or negative context, and such identification and abstraction is performed manually. Detailed description and soundness of this abstraction process is not at all obvious. In particular, as we discover in our work, there are intricate side-conditions (called "safe" conditions[1]) that are required for soundness. No previous work has formally discussed these side conditions and their role in proving CMP's correctness.

1.3 Our Contributions

In this paper, we present a formally verified theory of the CMP method in the interactive theorem prover Isabelle/HOL [32]. In particular, we fully verify the theoretical justification of the CMP method, including the use of assume-guarantee, symmetry arguments, and correctness of the abstraction procedure. As a natural outcome of the verification, we obtain an automatic, syntax-directed procedure for abstraction and strengthening that is proven correct in Isabelle/HOL, in the sense that correctness of the output protocol by model checking implies correctness of the original protocol. For expressing the transformation procedures, we define a syntax for protocols that closely corresponds to a subset of Murphi [15], and is expressive enough to handle the common benchmarks for the CMP method. We also implemented a tool AutoCMP for automatically carrying out abstraction and strengthening, as well as generating Isabelle proof scripts verifying their correctness. We applied our tool to four case studies, including the FLASH cache coherence protocol. For the three smaller case studies, the output of the transformation matches previous hand-edited results. For the FLASH protocol, we uncovered a few mistakes in the previous manual abstraction process. Verification of FLASH using CMP still succeeds after correcting these errors.

In summary, our contributions in this paper are as follows.

- We formalize the theoretical justification of CMP in the Isabelle theorem prover, including concepts of permutation and symmetry, symmetry reduction, and assume-guarantee reasoning.
- We describe syntax-directed procedures for abstraction and strengthening in CMP and prove the correctness of these two procedures. In the process, we clarify some tricky side conditions in the abstraction rules, proposing new concepts of *safe expressions* and *safe formulas*, which are necessary to guarantee the soundness of abstraction.
- Based on our syntax-guided description of abstraction and strengthening procedures in CMP, we implement a tool AutoCMP for carrying out these procedures automatically. The tool can also produce proof scripts in Isabelle showing correctness of abstraction and strengthening on each protocol. The overall theorem proved in Isabelle states that the parameterized protocol is correct on the assumption that the abstracted model is correct as checked by Murphi.
- We applied our work to analyze four case studies on CMP, including the FLASH cache-coherence protocol in [8], and discovered some mistakes in the original abstraction.

Organization of the paper. Section 2 introduces the preliminaries. We describe the formalization of permutation and symmetry in Section 3. The two central procedures in CMP: strengthening and abstraction, are formalized in Section 4 and Section 5, respectively. Section 6 presents the

[1]The term "safe" has special meaning here, different from the concept of safety properties in model checking.

formalization of the main theorem to justify the CMP method. Section 7 describes the AutoCMP tool, and presents a summary of experiments on four protocols. Section 8 concludes our paper with discussion of directions for future work.

2 PRELIMINARIES

2.1 Expressions and Formulas

We first describe the syntax and semantics for specifying protocols and their invariants. The syntax for values c, variables v, expressions e and formulas f are given as follows.

$$\begin{array}{lcl} c & ::= & \mathsf{enum} \mid \mathsf{b} \mid \mathsf{n} \mid \mathsf{X}_c \\ v & ::= & a \mid v[\mathsf{n}] \mid \mathsf{X}_{var} \\ e & ::= & v \mid c \mid f\,?\,e_1 : e_2 \mid \mathsf{X}_{exp} \\ f & ::= & e_1 = e_2 \mid \neg f_1 \mid f_1\, op\, f_2 \mid \bigwedge\limits_{i=1}^{N} f(i) \mid \bigwedge\limits_{i=1}^{N} i \neq \mathsf{n} \longrightarrow f(i) \mid \bigvee\limits_{i=1}^{N} f(i) \mid \mathsf{Chaos} \mid \mathsf{X}_{form} \end{array}$$

Values are either enumeration values with its own type, Boolean values (True or False), or natural number values. The enumeration values are used to specify control states, natural number values are intended only for specifying indices of nodes. State variables come in two types: global variables (in the form a), and local variables of each node (in the form $v[n]$). Expressions are either variables, values, or if-then-else forms (written as $f\,?\,e_1 : e_2$).

Formulas include equality between two expressions, the usual boolean connectives (here op is one of $\{\wedge, \vee, \longrightarrow\}$), and two forall-operators: quantifying over all nodes, and quantifying over all nodes except node j. While the latter can be expressed in terms of the former and the other boolean operations, we include it separately as they are handled in a special way during the abstraction transformation. In order to formalize and reason about expressions depending on unobservable values on an abstracted node in CMP, we also explicitly introduce X_c, X_{var}, X_{exp} and X_{form} to denote "unspecified" constants, variables, expressions, and formulas. These make it feasible to formally state the abstraction procedure for expressions at the syntactic level later in Section 5.

A state s is a mapping from variables to values. If there are N nodes, then a value is associated to each global variable a and each local variable $v[i]$ for $1 \leq i \leq N$. We use $s(v)$ for the value of variable v in state s, and $s(e)$ for the evaluation of expression e on s. We use $s \models f$ to denote that formula f is evaluated as true in s.

2.2 Statements and Protocols

One key aspect in the description of protocols considered here is the parallel semantics of assignment. That is, all statements in the body of a rule are considered to be executed in parallel, rather than sequentially as in usual programming languages. Hence, we adopt a parallel notation for statements. The syntax is given as follows.

$$S \;\;::=\;\; \mathsf{skip} \mid v := e \mid S_1 \,\|\, S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \|_{i=1}^{N} S(i) \mid \|_{i=1}^{N} i \neq \mathsf{n} \longrightarrow S(i)$$

The meaning of $\|_{i=1}^{N} S(i)$ is to perform all statements $S(i)$ for $1 \leq i \leq N$ in parallel. Similarly, $\|_{i=1}^{N} i \neq \mathsf{n} \longrightarrow S(i)$ performs all statements $S(i)$ for $1 \leq i \leq N$, except for n. We assume a well-formedness condition on statements, saying that each variable is assigned at most once in every statement (this condition can be easily checked syntactically). We use $\mathsf{trans}(a, s)$ to denote the result of executing statement a on state s.

A protocol rule is of the form $g \rhd a$, where the guard g is a formula and the action a is a statement. The interpretation of the rule is the following: if $s \models g$, then the rule can fire on state s, carrying it to state $\mathsf{trans}(a, s)$.

A protocol $\mathcal{P}$ can thus be specified by a pair $\langle I, R\rangle$, where I is a set of formulas describing the initial conditions, and R is a set of protocol rules. The *reachable states* of a protocol are defined inductively as follows: every state that satisfies all initial conditions are reachable. If s is reachable and there is a rule $r \in R$ carrying s to s', then s' is reachable. For the correctness proof of the CMP method, it is essential to define the concept of *reachability in n steps*. This is defined as an inductive predicate reachUpTo as follows.

$$\begin{array}{rcl} \forall f \in I.\, s \models f & \Longrightarrow & \mathsf{reachUpTo}(I, R, 0, s) \\ [\![\mathsf{reachUpTo}(I, R, n, s), g \rhd a \in R, s \models g]\!] & \Longrightarrow & \mathsf{reachUpTo}(I, R, n+1, \mathsf{trans}(a, s)) \end{array}$$

We define $\mathsf{Reach}(\mathcal{P})$ to be the entire reachable state set of a protocol $\mathcal{P} = \langle I, R\rangle$, that is, all s satisfying $\mathsf{reachUpTo}(I, R, i, s)$ for some i. A protocol $\mathcal{P}$ is said to satisfy an invariant f, which is denoted by $\mathcal{P} \models f$, if all states in $\mathsf{Reach}(\mathcal{P})$ satisfy f.

In this paper, it is essential to distinguish the syntax of expressions, formulas, and statements from their semantics, as the strengthening and abstraction procedures to be defined later will be applied at the syntactic level. We use the usual equality sign $\cdot = \cdot$ to denote syntactic equality. We use $f_1 \sim_{form} f_2$ to denote two formulas are semantically the same, that is they hold on the same set of states. Likewise, we use $a_1 \sim_{stm} a_2$ to denote two statements are semantically the same, that is they have the same behavior on all states. Semantic equivalence of two rules $g_1 \rhd a_1 \sim_{rule} g_2 \rhd a_2$ can then be defined as $(g_1 \sim_{form} g_2) \wedge (a_1 \sim_{stm} a_2)$.

2.3 Parameterized Protocols

So far, we have defined protocols as sets of protocol rules. In this paper, we will focus on protocols parameterized by a single integer N, standing for the number of nodes. Such protocols are described using parameterized formulas and rules.

A parameterized formula is a function $f(i_1, i_2, \ldots, i_m)$ from a tuple of node indices to formulas. Similarly, a parameterized rule is a function $r(i_1, i_2, \ldots, i_k)$ from a tuple of node indices to rules. Given the number of nodes N and a parameterized formula f, we write f^N for the set of formulas containing $f(i_1, i_2, \ldots, i_m)$ for all $1 \le i_1, \ldots, i_m \le N$. Likewise, we write r^N for the set of rules generated by r. For the case studies, we only need formulas and rules with at most two indices for specifying protocols.

2.4 Running Example: Mutual Exclusion

We used the mutual exclusion protocol (as considered in [12]) as a running example throughout this paper. The protocol describes a parameterized system where N nodes share a resource and need to ensure mutually exclusive access to the resource. Each node can be in one of four states: I (Idle), T (Try to enter critical region), C (in the Critical region) and E (Exit the critical region). The state of node i will be represented by a local variable $st[i]$. Mutually exclusive access is guaranteed by a lock, represented by a global Boolean variable x.

The parameterized rules of the mutual exclusion protocol are given as follows:

$$\begin{array}{rl} \mathsf{try}(i) \equiv & st[i] = \mathsf{I} \rhd st[i] := \mathsf{T} \\ \mathsf{crit}(i) \equiv & x \wedge st[i] = \mathsf{T} \rhd st[i] := \mathsf{C} \parallel x := \mathsf{False} \\ \mathsf{exit}(i) \equiv & st[i] = \mathsf{C} \rhd st[i] := \mathsf{E} \\ \mathsf{idle}(i) \equiv & st[i] = \mathsf{E} \rhd st[i] := \mathsf{I} \parallel x := \mathsf{True} \end{array}$$

For example, the second rule $\mathsf{crit}(i)$ states that any node i can transition from state T into state C only if the global lock x is True, and x is set to False simultaneously with the transition. The initial conditions of the protocol has two parts, stating that x is initially True, and each node is in state I.

$$\mathsf{init}_1 \equiv x = \mathsf{True} \quad \text{and} \quad \mathsf{init}_2(N) \equiv \bigwedge_{i=1}^{N} st[i] = \mathsf{I}$$

The mutual exclusion protocol $\mathcal{M}$ is then described by collecting together the above initial conditions and parameterized rules:

$$\mathcal{M}(N) = \langle \{\mathsf{init}_1, \mathsf{init}_2(N)\}, \mathsf{try}^N \cup \mathsf{crit}^N \cup \mathsf{exit}^N \cup \mathsf{idle}^N \rangle$$

The invariant of the protocol is given by a parameterized formula as follows, stating that any two distinct nodes cannot be in the critical section (state C) at the same time.

$$\mathsf{inv}(N, i) \equiv st[i] = \mathsf{C} \longrightarrow \bigwedge_{k=1}^{N} k \neq i \longrightarrow st[k] \neq \mathsf{C}.$$

And the full invariant to be verified is:

$$\mathcal{M}(N) \models \bigwedge_{i=1}^{N} \mathsf{inv}(N, i)$$

While the description of the above protocol is relatively simple, it already shows potential difficulties with verification. Standard model checking techniques are able to check the protocol for any fixed number of nodes N, but are unable to verify the invariant for any N. Theorem proving can potentially verify the protocol for any N, but may require complex invariants that are difficult to find manually, especially by system designers who may be unfamiliar with formal methods. The CMP method lies between the above two techniques. As we will show in the remainder of the paper, a single auxiliary invariant is sufficient to strengthen the protocol, such that its abstracted form can be verified corrected using the Murphi model checker.

3 PERMUTATION AND SYMMETRY

The CMP method is applied to the class of protocols whose nodes' behaviors are symmetric with each other. In this section, we formalize this meaning of symmetry, together with associated notions of permutation action and symmetry reduction. This mostly follows the description in [22], but we contribute the formalization of these concepts in Isabelle/HOL. Starting in this section, each definition and theorem have their correspondences in Isabelle, but for readability we still state them in more mathematical form.

3.1 Actions of Permutations

First, we describe the permutation action on protocols and specifications. Let N be the number of nodes, and π be a permutation on $\{1, \ldots, N\}$. For any $1 \leq i \leq N$, we write $\pi(i)$ for the result of applying π to i. We can then lift the application of π to our definitions of values, variables, expressions, formulas, statements and rules. In most cases the lifting is straightforward. We list the non-trivial cases as follows.

$$\begin{array}{rcll}
\mathsf{Perm}_c(\pi, \mathsf{n}) & \equiv & \pi(\mathsf{n}) & \text{if } c \text{ is an index } \mathsf{n} \\
\mathsf{Perm}_{var}(\pi, v[\mathsf{n}]) & \equiv & v[\pi(\mathsf{n})] & \text{if } v \text{ is a local variable} \\
\mathsf{Perm}_{form}(\pi, \bigwedge_{i=1}^{N} f(i)) & \equiv & \bigwedge_{i=1}^{N} \mathsf{Perm}_{form}(\pi, f(i)) & \\
\mathsf{Perm}_{form}(\pi, \bigwedge_{i=1}^{N} i \neq \mathsf{n} \longrightarrow f(i)) & \equiv & \bigwedge_{i=1}^{N} i \neq \mathsf{n} \longrightarrow \mathsf{Perm}_{form}(\pi, f(i)) & \\
\mathsf{Perm}_{stm}(\pi, \|_{i=1}^{N} S(i)) & \equiv & \|_{i=1}^{N} \mathsf{Perm}_{stm}(\pi, S(i)) & \\
\mathsf{Perm}_{stm}(\pi, \|_{i=1}^{N} i \neq \mathsf{n} \longrightarrow S(i)) & \equiv & \|_{i=1}^{N} i \neq \mathsf{n} \longrightarrow \mathsf{Perm}_{stm}(\pi, S(i)) &
\end{array}$$

We say a set F of formulas is *symmetric*, if for every $f \in F$ and permutation π, there exists a formula $f' \in F$ such that $\mathsf{Perm}_{form}(\pi, f) \sim_{form} f'$. Likewise we define symmetry for sets of statements and protocol rules. Finally, we say a protocol presentation $\mathcal{P} = \langle I, R\rangle$ is *symmetric*, if I is a symmetric set of formulas, and R is a symmetric set of protocol rules (see also [22]).

We now define the actions of permutations on states. This involves using the permutation to transform both indices in local variables and indices in values. The definition is given as follows, note the inverse application of π on indices of local variables.

$$\begin{aligned} \mathsf{Perm}_{st}(\pi, s)(a) &\equiv \mathsf{Perm}_c(\pi, s(a)) \\ \mathsf{Perm}_{st}(\pi, s)(v[\mathsf{n}]) &\equiv \mathsf{Perm}_c(\pi, s(v[\pi^{-1}(\mathsf{n})])) \end{aligned}$$

The following lemmas state that our definitions of actions on formulas, expressions, and states are consistent with each other.

Lemma 3.1. *Given a permutation π on $\{1, \dots, N\}$, evaluation of expressions and formulas on states commutes with symmetry action:*

$$\begin{aligned} \mathsf{Perm}_{st}(\pi, s)(\mathsf{Perm}_{exp}(\pi, e)) &= \mathsf{Perm}_c(\pi, s(e)) \\ \mathsf{Perm}_{st}(\pi, s) \models \mathsf{Perm}_{form}(\pi, f) &\longleftrightarrow s \models f \end{aligned}$$

Likewise, applying a permutation action commutes with applying transition associated to a statement:

$$\mathsf{Perm}_{st}(\pi, \mathsf{trans}(a, s)) = \mathsf{trans}(\mathsf{Perm}_{stm}(\pi, a), \mathsf{Perm}_{st}(\pi, s)).$$

3.2 Proving Symmetry

For applications of CMP to real-world protocols, we need to ensure that the protocol definition is in fact symmetric. Symmetry is by no means automatically guaranteed. For example, the original version of the FLASH protocol makes use of a home node, which breaks symmetry among nodes. To apply CMP to FLASH in a rigorous way, the protocol needs to be modified to make any information about the home node into global variables.

In our work, we prove a number of lemmas that allow us to quickly derive symmetry of protocol presentations in Isabelle. Most of these lemmas are straightforward. The only tricky ones are those involving forall-formulas and forall-statements. To state the lemmas precisely, we first define the following properties on parameterized formulas (the definitions can be easily extended to the case of more than two parameters).

Definition 3.2. A parameterized formula f on one parameter i is symmetric if for any N and permutation π on $\{1, \dots, N\}$, we have $\mathsf{Perm}_{form}(\pi, f(i)) \sim_{form} f(\pi(i))$ for all $1 \leq i \leq N$. Likewise, a parameterized formula f on two two parameters is symmetric if $\mathsf{Perm}_{form}(\pi, f(i, j)) \sim_{form} f(\pi(i), \pi(j))$ for all $1 \leq i, j \leq N$.

Lemma 3.3. *The following results hold, linking symmetry of parameterized formulas and symmetry of generated sets. If a parameterized formula $f(i)$ is symmetric, then f^N is a symmetric set of formulas. Likewise for the case of two parameters. If a parameterized formula $f(i, j)$ is symmetric, then the formulas $\bigwedge_{i=1}^{N} f(i, j)$ and $\bigwedge_{i=1}^{N} i \neq j \longrightarrow f(i)$ parameterized on j are symmetric.*

Lemma 3.4. *The following results are compositional laws of symmetry on formula constructors.*

1. *If a parameterized formula $f(i)$ and $g(i)$ is symmetric, then $f(i)$ op $g(i)$ is a symmetric set of formulas. Likewise for the case of two parameters. Here, we have op $\in \{\wedge, \vee, \longrightarrow\}$.*
2. *If a parameterized formula $f(i)$ is symmetric, then $\neg f(i)$ is also symmetric.*

(3) If a parameterized formula $f(i,j)$ is symmetric, then the formulas $\bigwedge_{i=1}^{N} f(i,j)$ and $\bigwedge_{i=1}^{N} i \neq j \rightarrow f(i)$ parameterized on j are symmetric.

Using these lemmas, as well as more trivial results about taking unions and operation on formulas, automation in Isabelle is able to conclude symmetry of guard conditions and invariants in our examples. A similar development can be performed for sets of statements.

Definition 3.5. A parameterized statement S on one parameter i is symmetric if for any N and permutation π on $\{1,\ldots,N\}$, we have $\mathsf{Perm}_{stm}(\pi, S(i)) \sim_{stm} S(\pi(i))$ for all $1 \leq i \leq N$. Likewise, a parameterized statement r on two parameters is symmetric if $\mathsf{Perm}_{stm}(\pi, f(i,j)) \sim_{stm} r(\pi(i), \pi(j))$ for all $1 \leq i, j \leq N$.

LEMMA 3.6. *The following results hold, linking symmetry of parameterized statements and symmetry of generated sets. If a parameterized statement $S(i)$ is symmetric, then S^N is a symmetric set of statements. Likewise for the case of two parameters. If a parameterized statement $S(i,j)$ is symmetric, then we have the statements $\|_{i=1}^{N} S(i,j)$ and $\|_{i=1}^{N} i \neq j \rightarrow S(i,j)$ parameterized on j are symmetric.*

Symmetry of parameterized family of rules $g(i) \triangleright a(i)$ is reduced to showing the symmetry of families $g(i)$ and $a(i)$, respectively.

Definition 3.7. A parameterized rule $g \triangleright S$ on one parameter i is symmetric if g and S is symmetric for all $1 \leq i \leq N$. Likewise, we can define a parameterized rule with two parameters. A symmetric set of rules can also be defined as above. If I and R are a symmetric set of formulas and protocol rules respectively, then we say that $\mathcal{P} = \langle I, R \rangle$ is a *symmetric presentation* of the protocol.

3.3 Symmetry Reduction

We now consider the principle of *symmetry reduction*. The general technique of symmetry reduction has been developed in [19–21, 38] to address the state explosion problem for model checking of protocols with symmetry properties. The basic idea is simple but effective: it uses an equivalence relation between states to explore only one state per equivalence class. The CMP method also makes use of a form of symmetry reduction, reducing model checking for an arbitrary number of nodes to model checking a small fixed number of nodes.

In our context, symmetry reduction states that if a protocol has a symmetric presentation, then it satisfies a formula f if and only if it satisfies all permutations of f. The statement of this principle, in a form close to its formalization in Isabelle, is as follows.

LEMMA 3.8. *Given a protocol $\mathcal{P}$ with a symmetric presentation, and a permutation π on $\{1,\ldots,N\}$. If a state s is reachable in n steps, then the permuted state $\mathsf{Perm}_{st}(\pi, s)$ is also reachable in n steps. More precisely:*

$$\forall s.\ \mathsf{reachUpTo}(I, R, n, s) \longrightarrow \mathsf{reachUpTo}(I, R, n, \mathsf{Perm}_{st}(\pi, s))$$

This leads to the following theorem.

THEOREM 3.9 (SYMMETRY REDUCTION). *Given a protocol $\mathcal{P}$ with a symmetric presentation, and a permutation π on $\{1,\ldots,N\}$. Then for any formula f, $s \models f$ for any reachable state s if and only if $s \models \mathsf{Perm}_{form}(\pi, f)$ for any reachable state s. In other words, if f is an invariant of $\mathcal{P}$, then any formula f' such that $f' \sim_{form} \mathsf{Perm}_{form}(\pi, f)$ is also an invariant.*

Running example. Let us revisit the mutual exclusion protocol. It is easy to check syntactically that it is symmetric. By symmetry reduction, we can reduce the property to be verified into a simpler form. Let $\mathsf{inv}'(i,j) \equiv i \neq j \wedge st[i] = \mathsf{C} \rightarrow st[j] \neq \mathsf{C}$. By symmetry arguments, $\mathcal{M}(N) \models \bigwedge_{i=1}^{N} \mathsf{inv}(N,i)$ iff $\mathcal{M}(N) \models \mathsf{inv}'(1,2)$. Therefore, we only need to check this simpler form of the invariant.

4 GUARD STRENGTHENING

The idea of guard strengthening is to add an auxiliary invariant to the guard of certain rules, in order to refine the resulting abstracted protocol by constraining its allowable behaviours. We begin by presenting the guard strengthening procedure that is only appropriate in the *semantic* sense, that is, yielding a formula that is semantically equal to the desired one, but is not appropriate in the ensuing abstraction. The simple strengthening procedure is as follows:

Definition 4.1 (Simple strengthening). Given an auxiliary invariant f, strengthening of protocol rule $g \triangleright a$ by f is defined as:

$$\mathsf{strengthen}(g \triangleright a, f) \equiv (g \wedge f) \triangleright a$$

Strengthening a rule by a finite set of auxiliary invariants means applying strengthening one-by-one using invariants in this set. In particular, strengthening a protocol rule by an empty set $\emptyset$ of auxiliary invariants leaves the rule unmodified.

Given two protocols $\mathcal{P} = \langle I, R\rangle$ and $\mathcal{P}' = \langle I, R'\rangle$ with the same initial conditions I, we say $\mathcal{P}'$ is *guard-strengthened* by a set F of auxiliary invariants from $\mathcal{P}$, if any rule $r' \in R'$ is guard-strengthened from a rule $r \in R$ using a subset of F. The following lemma contains the core of the assume-guarantee principle justifying the CMP method.

Lemma 4.2 (Correctness of strengthening). *Let $\mathcal{P} = \langle I, R\rangle$, $\mathcal{P}' = \langle I, R'\rangle$, f is an invariant to be checked, and F is a set of auxiliary invariants s.t. $f \in F$. From the following assumptions:*

(1) $\mathcal{P}'$ is guard-strengthened by the set F from $\mathcal{P}$.
(2) $\mathcal{P}' \models f'$ for any $f' \in F$.

We can conclude that for any s, $s \in \mathsf{Reach}(\mathcal{P})$ implies $s \in \mathsf{Reach}(\mathcal{P}')$ and $s \models f'$ for any $f' \in F$. This means $\mathsf{Reach}(\mathcal{P}) \subseteq \mathsf{Reach}(\mathcal{P}')$ and $\mathcal{P} \models f'$ for any $f' \in F$. In particular, the protocol $\mathcal{P}$ also satisfies the invariant f.

The above simple form of guard strengthening is given in most presentations of the CMP method. Unfortunately, this strengthening rule is not sufficient on the syntactic level. As an example, consider our mutual exclusion example. The auxiliary invariant that is needed can be written as:

$$\mathsf{strExit}(N, i) \equiv st[i] = \mathsf{E} \longrightarrow \bigwedge_{j=1}^{N} j \neq i \longrightarrow st[j] \neq \mathsf{C} \wedge st[j] \neq \mathsf{E}$$

It states that if one node is in state E, then no other node can be in either state C or E. This auxiliary invariant should be used to strengthen the idle rule. If the above simple strengthening procedure is used, the strengthened idle rule would be (adding the invariant to the guard of the rule):

$$st[i] = \mathsf{E} \wedge (st[i] = \mathsf{E} \longrightarrow \bigwedge_{j=1}^{N} j \neq i \longrightarrow st[j] \neq \mathsf{C} \wedge st[j] \neq \mathsf{E}) \triangleright \{st[i] := \mathsf{I}, x := \mathsf{True}\}$$

However, the rule in the current form cannot be abstracted correctly using the syntax-directed procedure in Section 5. The additional operation that is needed is to use the condition $st[i] = \mathsf{E}$ already in the guard of idle to discharge the premise of the auxiliary invariant. We thus formalize this idea below.

Definition 4.3. The function $\mathsf{removeImplies}(f, g)$ removes implication in f if its premise can be implied by g:

$$\begin{aligned} \mathsf{removeImplies}(f_1 \longrightarrow f_2, g) &\equiv (\text{if } g \Rightarrow_{form} f_1 \text{ then } f_2 \text{ else } f_1 \longrightarrow f_2) \\ \mathsf{removeImplies}(f, g) &\equiv f \quad \text{if } f \text{ is not in implies form} \end{aligned}$$

Here $g \Rightarrow_{form} f_1$ means g implies f_1 semantically. This can be approximated by some syntactic checks. In our case studies, it is sufficient to perform the check that each conjunct of f_1 occurs (syntactically) as a conjunct in g.

Definition 4.4 (Refined strengthening). Given an auxiliary invariant f, refined strengthening of rule $g \rhd a$ by f is defined as:

$$\mathsf{strengthen}_2(g \rhd a, f) \equiv (g \wedge \mathsf{removeImplies}(f, g)) \rhd a.$$

It is clear that the two strengthening procedures are semantically equivalent:

Lemma 4.5. *For any auxiliary invariant f and rule $g \rhd a$, we have:*

$$\mathsf{strengthen}(g \rhd a, f) \sim_{rule} \mathsf{strengthen}_2(g \rhd a, f).$$

As a consequence, the statement of Lemma 4.2 also holds with strengthen *replaced by* $\mathsf{strengthen}_2$.

Running example. As an example, the result of applying $\mathsf{strengthen}_2$ with $\mathsf{strExit}(N, i)$ on rule idle is:

$$st[i] = \mathsf{E} \wedge (\bigwedge_{j=1}^{N} j \neq i \longrightarrow st[j] \neq \mathsf{C} \wedge st[j] \neq \mathsf{E}) \rhd \{st[i] := \mathsf{I}, x := \mathsf{True}\},$$

It adds the conclusion of $\mathsf{strExit}(N, i)$ to the guard of idle, since the assumption $st[i] = \mathsf{E}$ of $\mathsf{strExit}(N, i)$ already appears in the guard. This form of strengthening is now ready for abstraction.

For the other three rules try, crit, and exit, we strengthen them by the empty set $\emptyset$, then we get intermediate parameterized protocol rules as follows:

$$\begin{array}{rll}
\mathsf{try}_{ref}(i) \equiv & st[i] = \mathsf{I} & \rhd\ st[i] := \mathsf{T} \\
\mathsf{crit}_{ref}(i) \equiv & x \wedge st[i] = \mathsf{T} & \rhd\ st[i] := \mathsf{C} \parallel x := \mathsf{False} \\
\mathsf{exit}_{ref}(i) \equiv & st[i] = \mathsf{C} & \rhd\ st[i] := \mathsf{E} \\
\mathsf{idle}_{ref}(i) \equiv & st[i] = \mathsf{E} \wedge (\bigwedge_{j=1}^{N} j \neq i \longrightarrow st[j] \neq \mathsf{C} \wedge st[j] \neq \mathsf{E}) & \rhd\ \{st[i] := \mathsf{I}, x := \mathsf{True}\}
\end{array}$$

We collect these rules to form the strengthened mutual exclusion protocol $\mathcal{M}_{ref}$ as follows:

$$\mathcal{M}_{ref}(N) = \langle \{\mathsf{init}_1, \mathsf{init}_2(N)\}, \mathsf{try}_{ref}^{N} \cup \mathsf{crit}_{ref}^{N} \cup \mathsf{exit}_{ref}^{N} \cup \mathsf{idle}_{ref}^{N} \rangle$$

5 ABSTRACTION

The abstraction procedure is the most difficult part of formally describing the CMP method. The procedure should satisfy the following properties:

- The abstraction should be fine-grained. That is, it should be no more conservative than necessary, so it does not produce false negative results. We show that a type system for protocol descriptions is necessary to achieve this fine-grainedness (Section 5.1).
- The abstraction should be syntax-guided, so it can be implemented as an automatic procedure. We describe this syntax-guided procedure, along with some non-trivial side conditions that are necessary for soundness in Section 5.2.
- The abstraction should be sound, which guarantees that the abstracted protocol can simulate the original protocol. We define this simulation property and state the soundness theorem in Section 5.3.

5.1 A Simple Type System

We first explain why a type system on protocol descriptions is necessary to define fine-grained abstractions. Consider the protocol rule $x \neq y \triangleright z := \mathsf{c}$, where x, y, z are variables, and c is a constant. Without type information of the variables, we will not know whether $x \neq y$ can be preserved when node indices $\mathsf{n} > M$ are abstracted into $M + 1$, so we have to abstract the rule into $\mathsf{True} \triangleright z := \mathsf{c}$. However, if x and y are known as boolean or enumerating type, we know $x \neq y$ will be preserved by abstraction of node indices, so we can perform the more fine-grained abstraction of this rule into $x \neq y \triangleright z := \mathsf{c}$.

First, we introduce a typing environment *env* which maps variables to basic types: *Enum*, *Bool*, *Index*, and X_{type}. Here, X_{type} is designed for reasoning about types of unknown objects such as X_{var}. Based on *env*, we can derive the type of an expression, denoted by $env \Vdash e : t$; and define what it means for a formula f, a statement S, and a rule r to be well-typed by *env*, denoted by $env \Vdash f$, $env \Vdash S$ and $env \Vdash r$, respectively. The term X_c is assigned type X_{type}. For a variable v, its type is assigned to its predefined type $env(v)$. For an ite-expression $f\,?\,e_1 : e_2$ to have type t, both e_1 and e_2 need to have the same type t, and f must be well-typed. The other typing rules are self-evident. The full set of rules are given in Table 1.

Table 1. The simple type system

$\dfrac{}{env \Vdash \mathsf{enum} : Enum}$	$\dfrac{}{env \Vdash \mathsf{b} : Bool}$
$\dfrac{}{env \Vdash i : Index}$	$\dfrac{}{env \Vdash \mathsf{X}_c : \mathsf{X}_{type}}$
$\dfrac{env(v) = t}{env \Vdash v : t}$	$\dfrac{env \Vdash e_1 : t; env \Vdash e_2 : t, env \Vdash f}{env \Vdash (f\,?\,e_1 : e_2) : t}$
$\dfrac{env \Vdash e_1 : t; env \Vdash e_2 : t}{env \Vdash e_1 = e_2}$	$\dfrac{env \Vdash f}{env \Vdash \neg f}$
$\dfrac{env \Vdash f_1; env \Vdash f_2}{env \Vdash f_1\ op\ f_2}$	$\dfrac{\forall i. env \Vdash f(i)}{env \Vdash \bigwedge_{i=1}^{N} f(i)}$
$\dfrac{\forall i. env \Vdash f(i)}{env \Vdash \bigwedge_{i=1}^{N} i \neq \mathsf{n} \longrightarrow f(i)}$	$\dfrac{\forall i. env \Vdash f(i)}{env \Vdash \bigvee_{i=1}^{N} f(i)}$
$\dfrac{}{env \Vdash \mathsf{skip}}$	$\dfrac{env \Vdash x : t; env \Vdash e : t}{env \Vdash \mathsf{x} := \mathsf{e}}$
$\dfrac{env \Vdash S_1; env \Vdash S_2}{env \Vdash S_1 \parallel S_2}$	$\dfrac{env \Vdash b; env \Vdash S_1; env \Vdash S_2}{env \Vdash \text{if } b \text{ then } S_1 \text{ else } S_2}$
$\dfrac{\forall i. env \Vdash S(i)}{env \Vdash \parallel_{i=1}^{N} S(i)}$	$\dfrac{\forall i. env \Vdash S(i)}{env \Vdash \parallel_{i=1}^{N} i \neq \mathsf{n} \longrightarrow S(i)}$
$\dfrac{env \Vdash g; env \Vdash S}{env \Vdash g \triangleright S}$	

We next introduce the concept of a state of the protocol satisfying a given typing environment.

Definition 5.1. A state s is well-typed by a type environment env, denoted $\mathsf{fitEnv}(s, env)$, if any variable v in s such that $env(v) \neq \mathsf{X}_{type}$ satisfies $env \Vdash s(v) : env(v)$.

The next lemma formalizes an induction principle on the well-typed properties of reachable states of a protocol.

LEMMA 5.2. *Fix a protocol $\mathcal{P} = \langle I, R \rangle$. Suppose the following two assumptions hold.*

(1) For any initial state s satisfying I, we have $\mathsf{fitEnv}(s, env)$*;*
(2) For any rule $r \in R$ and state s satisfying $s \models \mathsf{guard}(r)$, if in addition $\mathsf{fitEnv}(s, env)$*, then* $\mathsf{fitEnv}(\mathsf{trans}(\mathsf{action}(r), s), env)$*.*

Then for any reachable state s of $\mathcal{P}$, we have $\mathsf{fitEnv}(s, env)$*.*

While the type system in our case is relatively simple, it is a necessary part of defining and proving correctness of abstraction. Hence type checking need to be performed on the protocol description, with proof generation by the tool AutoCMP like for all other parts of the procedure.

5.2 Syntax-guided Abstraction

In this section, we present a formalization of the abstraction procedure. This is the central part of the CMP method. The basic idea is as follows. Given a fixed M, for an instantiation of the protocol with $N > M$ nodes, we can abstract it to one with M ordinary nodes plus a special *other* node. Following the formalization in Isabelle, we use $M + 1$ to represent the index of the *other* node. Abstraction for formulas (guards) are *over-approximations*, so that the original protocol and the abstracted protocol observe a *simulation* relation: any transition in the original protocol corresponds to a transition in the abstracted protocol, but the abstracted protocol may allow more behaviors.

There are many subtleties in the abstraction rules due to this asymmetry. For example, if we have an equality guard $i = j$ between two indices, then it is safe to abstract it to $\mathsf{Abs}_{exp}(i) = \mathsf{Abs}_{exp}(j)$ (the abstraction function Abs_{exp} on expressions, to be defined below, transforms i to $M + 1$ if $i > M$ and leaves it unchanged otherwise). However, it is not valid to transform a guard $i \neq j$ into $\mathsf{Abs}_{exp}(i) \neq \mathsf{Abs}_{exp}(j)$, because if i and j take distinct values greater than M, then $i \neq j$ holds but $\mathsf{Abs}_{exp}(i) \neq \mathsf{Abs}_{exp}(j)$ does not hold.

We now define the action of abstraction on values, variables, expressions, formulas and rules. In stating the abstraction rules, we use X_{var}, X_c, X_{exp} and X_{form} to propagate information that is unknown. For values, abstraction of indices is by cutoff at M. Other kinds of values are unchanged:

$$\begin{array}{rcll}
\mathsf{Abs}_c(M, c) & \equiv & c & \text{if } c \text{ is of the form } \mathsf{enum}, \mathsf{b} \text{ or } \mathsf{X}_c \\
\mathsf{Abs}_c(M, \mathsf{n}) & \equiv & M+1 & \text{if } \mathsf{n} > M \\
\mathsf{Abs}_c(M, \mathsf{n}) & \equiv & \mathsf{n} & \text{if } \mathsf{n} \leq M
\end{array}$$

For variables, local variable $v[i]$ with index $i > M$ is abstracted to X_{var}. All other variables are preserved.

$$\begin{array}{rcll}
\mathsf{Abs}_{var}(M, a) & \equiv & a & \text{if } a \text{ is a global variable} \\
\mathsf{Abs}_{var}(M, v[\mathsf{n}]) & \equiv & v[\mathsf{n}] & \text{if } \mathsf{n} \leq M \text{ otherwise } \mathsf{X}_{var} \\
\mathsf{Abs}_{var}(M, \mathsf{X}_{var}) & \equiv & \mathsf{X}_{var} &
\end{array}$$

We now consider abstraction of expressions and formulas. For this, we first state some preliminary concepts, concerning boundedness and safety of expression and formulas. Intuitively, an expression or formula is *bounded on i* if all indices occurring in it equal i. A formula is *safe* if abstraction of the formula gives an equivalence (not just an over-approximation). All these conditions are defined by recursion, so they can be checked in a syntactic manner.

Definition 5.3. We define the following conditions on expressions and formulas.

- An expression e is a bound variable on i (written $\mathsf{bound}_{var}(i, e)$) if it is a variable of the form a or $v[i]$.
- An expression e is a bound expression (written $\mathsf{bound}_{exp}(i, e)$) if it is an enumerating value, boolean value, or the index i. A formula f is a bound formula on i (written $\mathsf{bound}_{form}(i, f)$) if it is a propositional combination of equalities of the form $e_1 = e_2$, where e_1 is a bound variable on i, and e_2 is a bound expression, or vice versa.
- Safety for M of variables x (written $\mathsf{safe}_{var}(x, M)$), expressions e (written $\mathsf{safe}_{exp}(env, M, e)$), and formulas f (written $\mathsf{safe}_{form}(env, M, f)$) are defined as follows:

$$
\begin{array}{rcll}
\mathsf{safe}_{var}(x, M) & \equiv & \mathsf{True} & \text{if } x \text{ is a global variable} \\
\mathsf{safe}_{var}(a[\mathsf{n}], M) & \equiv & \mathsf{True} & \text{if } \mathsf{n} \leq M \text{ else False} \\
\mathsf{safe}_{exp}(env, M, c) & \equiv & \mathsf{True} & \text{if } env \Vdash c : Enum \\
 & & & \ \text{or } env \Vdash c : Bool \text{ or } env \Vdash c : Index \wedge c \leq M \\
\mathsf{safe}_{exp}(env, M, v) & \equiv & \mathsf{True} & \text{if } \mathsf{safe}_{var}(M, v) \text{ and} \\
 & & & (env(v) = Enum \text{ or } env(v) = Bool) \\
\mathsf{safe}_{exp}(env, M, f?e_1 : e_2) & \equiv & \mathsf{True} & \text{if } \mathsf{safe}_{exp}(env, M, e_1) \wedge \mathsf{safe}_{exp}(env, M, e_2) \\
 & & & \wedge\ \mathsf{safe}_{form}(env, M, f) \\
\mathsf{safe}_{form}(env, M, e_1 = e_2) & \equiv & \mathsf{True} & \text{if } env \Vdash e_1 : Index \wedge \mathsf{safe}_{exp}(env, M, e_2) \wedge (\exists i. e_2 = i) \wedge \\
 & & & ((\exists v. e_1 = v \wedge \mathsf{safe}_{var}(v, M)) \vee (\exists i. e_1 = i)) \vee \\
 & & & env \Vdash e_2 : Index \wedge \mathsf{safe}_{exp}(env, M, e_1) \wedge (\exists i. e_1 = i) \wedge \\
 & & & ((\exists v. e_2 = v \wedge \mathsf{safe}_{var}(v, M)) \vee (\exists i. e_2 = i)) \vee \\
 & & & (env \Vdash e_1 : Bool \vee env \Vdash e_1 : Enum) \wedge \\
 & & & \mathsf{safe}_{exp}(env, M, e_1) \wedge \mathsf{safe}_{exp}(env, M, e_2) \\
\mathsf{safe}_{form}(env, M, \neg f) & \equiv & \mathsf{True} & \text{if } \mathsf{safe}_{form}(env, M, f) \\
\mathsf{safe}_{form}(env, M, f_1\ op\ f_2) & \equiv & \mathsf{True} & \text{if } \mathsf{safe}_{form}(env, M, f_1) \text{ and } \mathsf{safe}_{form}(env, M, f_2) \\
\mathsf{safe}_{form}(env, M, Chaos) & \equiv & \mathsf{True} &
\end{array}
$$

With these definitions, we are ready to define abstraction on expressions e and formulas f. Most of the rules are straightforward, the main idea being to propagate unknowns, and recursively apply abstraction to negations only if the formula involved is safe. We state the key rules below (here we assume $M \leq N$).

$$
\begin{array}{rcll}
\mathsf{Abs}_{exp}(env, M, c) & \equiv & \mathsf{Abs}_{c}(M, c) & \text{for value } c \\
\mathsf{Abs}_{exp}(env, M, v) & \equiv & \mathsf{Abs}_{var}(M, v) & \text{for variable } v \text{ with } \mathsf{Abs}_{var}(M, v) \neq \mathsf{X}_{var} \\
 & \equiv & \mathsf{X}_{exp} & \text{otherwise} \\
\mathsf{Abs}_{exp}(env, M, f?e_1 : e_2) & \equiv & \begin{array}{l} \mathsf{Abs}_{form}(env, M, f)? \\ \mathsf{Abs}_{exp}(env, M, e_1) : \\ \mathsf{Abs}_{exp}(env, M, e_2) \end{array} & \text{if } \begin{array}{l} \mathsf{Abs}_{form}(env, M, f) \neq \mathsf{X}_{form} \wedge \\ \mathsf{Abs}_{exp}(env, M, e_1) \neq \mathsf{X}_{exp} \wedge \\ \mathsf{Abs}_{exp}(env, M, e_2) \neq \mathsf{X}_{exp} \wedge \\ \mathsf{safe}_{form}(env, M, f) \end{array} \\
 & \equiv & \mathsf{X}_{exp} & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Abs}_{form}(env, M, e_1 = e_2) & \equiv & \mathsf{X}_{form} \text{ if } \mathsf{Abs}_{exp}(M, e_1) = \mathsf{X}_{exp} \\
& & \text{ or } \mathsf{Abs}_{exp}(env, M, e_2) = \mathsf{X}_{exp} \\
& \equiv & \mathsf{Abs}_{exp}(env, M, e_1) = \mathsf{Abs}_{exp}(env, M, e_2) \text{ otherwise} \\
\mathsf{Abs}_{form}(env, M, \neg f) & \equiv & \neg \mathsf{Abs}_{form}(env, M, f) \text{ if } \mathsf{safe}_{form}(env, M, f) \\
& \equiv & \mathsf{X}_{form} \text{ otherwise} \\
\mathsf{Abs}_{form}(env, M, f_1 \wedge f_2) & \equiv & \mathsf{Abs}_{form}(env, M, f_2) \text{ if } \mathsf{Abs}_{form}(env, M, f_1) = \mathsf{X}_{form} \\
& \equiv & \mathsf{Abs}_{form}(env, M, f_1) \text{ if } \mathsf{Abs}_{form}(env, M, f_2) = \mathsf{X}_{form} \\
& \equiv & \mathsf{Abs}_{form}(env, M, f_1) \wedge \mathsf{Abs}_{form}(env, M, f_2) \text{ otherwise} \\
\mathsf{Abs}_{form}(env, M, f_1 \vee f_2) & \equiv & \mathsf{X}_{form} \text{ if } \mathsf{Abs}_{form}(env, M, f_1) = \mathsf{X}_{form} \\
& & \text{ or } \mathsf{Abs}_{form}(env, M, f_2) = \mathsf{X}_{form} \\
& \equiv & \mathsf{Abs}_{form}(env, M, f_1) \vee \mathsf{Abs}_{form}(env, M, f_2) \text{ otherwise} \\
\mathsf{Abs}_{form}(env, M, \mathsf{X}_{form}) & \equiv & \mathsf{X}_{form} \\
\mathsf{Abs}_{form}(env, M, \mathsf{Chaos}) & \equiv & \mathsf{True} \\
\mathsf{Abs}_{form}(env, M, \bigwedge_{i=1}^{N} F) & \equiv & \bigwedge_{i=1}^{M} F \text{ if } M \leq N \text{ and } \mathsf{bound}_{form}(i, F(i)) \\
& \equiv & \mathsf{X}_{form} \text{ otherwise} \\
\mathsf{Abs}_{form}(env, M, \bigwedge_{i=1}^{N} i \neq j \longrightarrow F(i)) & \equiv & \bigwedge_{i=1}^{M} i \neq \mathsf{n} \longrightarrow F(i) \text{ if } M \leq N \text{ and } \mathsf{n} \leq M \text{ and} \\
& & \mathsf{bound}_{form}(i, F(i)) \\
& \equiv & \bigwedge_{i=1}^{M} F \text{ if } M \leq N \text{ and } \mathsf{n} > M \text{ and } \mathsf{bound}_{form}(i, F(i)) \\
& \equiv & \mathsf{X}_{form} \text{ otherwise} \\
\mathsf{Abs}_{form}(env, M, \bigvee_{i=1}^{N} F) & \equiv & \mathsf{X}_{form}
\end{array}
$$

In particular, the $\mathsf{safe}_{form}(f)$ condition is used in the abstraction of negation formula $\neg f$ and ite-expression $f \mathbin{?} e_1 : e_2$, which applies the abstraction at the next level only if f is safe, otherwise $\neg f$ and $f \mathbin{?} e_1 : e_2$ are abstracted into X_{form} and X_{exp} respectively. The condition $\mathsf{bound}_{form}(i, F(i))$ is used to restrict the syntax of $F(i)$ when abstracting the quantified formulas $\bigwedge_{i=1}^{N} F$ and $\bigwedge_{i=1}^{N} i \neq \mathsf{n} \longrightarrow F(i)$.

As a quick check, we see how the above definition prevents the invalid abstraction of an inequality between two indices $i \neq j$, represented as $\neg(i = j)$. In the rule for $\mathsf{Abs}_{form}(M, \neg f)$, we check whether f is a safe form. However, neither i nor j are bound variables or bound expressions, hence $\neg(i = j)$ is abstracted to X_{form}. On the other hand, $i = j$ is abstracted to $\mathsf{Abs}_{exp}(i) = \mathsf{Abs}_{exp}(j)$, as neither side is abstracted to X_{exp}.

Abstraction on states simply combines abstraction of variables and constants. It can be defined directly as follows:

$$
\begin{array}{rcll}
\mathsf{Abs}_{st}(M, s)(a) & \equiv & \mathsf{Abs}_c(M, s(a)) & \text{if } a \text{ is a global variable} \\
\mathsf{Abs}_{st}(M, s)(v[i]) & \equiv & \mathsf{Abs}_c(M, s(v[i])) & \text{if } i \leq M \text{ otherwise } \mathsf{X}_c \\
\mathsf{Abs}_{st}(M, s)(\mathsf{X}_{var}) & \equiv & \mathsf{X}_c &
\end{array}
$$

LEMMA 5.4. *The following properties hold for environments, bound expressions, bound formulas, and safe formulas, if for a state s such that* $\mathsf{fitEnv}(s, env)$*:*

(1) *For any bound expression e such that $env \Vdash e : t$ for some $t \neq \mathsf{X}_{type}$, and the state s, we have* $\mathsf{Abs}_{st}(M, s)(e) = s(e)$.
(2) *For any bound formula f on i with $i \leq M$ and $env \Vdash f$, we have $s \models f \longleftrightarrow \mathsf{Abs}_{st}(M, s) \models \mathsf{Abs}_{form}(env, M, f)$.*

(3) *For any formula f such as* $\mathsf{safe}_{form}(env, M, f)$ *and* $env \Vdash f$, *we have* $s \models f \longleftrightarrow \mathsf{Abs}_{st}(M, s) \models \mathsf{Abs}_{form}(env, M, f)$.

Abstraction of expressions and formulas commutes with abstraction of states, as given in the following lemma. Note the abstraction of formulas gives an *over-approximation.*

LEMMA 5.5. *Given* $M \leq N$ *and expression* e, *if* $\mathsf{Abs}_{exp}(M, e) \neq \mathsf{X}_{exp}$, *and for any state* s, $\mathsf{fitEnv}(s, env)$ *and* $env \Vdash e : t$ *for some* $t \neq \mathsf{X}_{type}$,

$$\mathsf{Abs}_{st}(M, s)(\mathsf{Abs}_{exp}(env, M, e)) = \mathsf{Abs}_c(s(e)).$$

If a formula f *satisfies* $\mathsf{Abs}_{form}(M, f) \neq \mathsf{X}_{form}$, *for any state* s, *if* $\mathsf{fitEnv}(s, env)$ *and* $env \Vdash f$, *then*

$$s \models f \longrightarrow \mathsf{Abs}_{st}(M, s) \models \mathsf{Abs}_{form}(env, M, f).$$

For abstraction of statements, we further define the notion of boundedness of assignments. The condition $\mathsf{boundAssign}(i, S)$ means the statement does not contain forall-statements, and all assignments in the statement are to local variables of a node i. This is the condition that should be satisfied for the body $S(i)$ of a forall-statement. Then, the *well-formedness* of a statement is defined recursively. We show the these cases below.

$$\begin{array}{rcl}
\mathsf{wellform}_{stm}(env, M, v := e) & \equiv & \forall M.\ \mathsf{Abs}_{var}(env, M, v) \neq \mathsf{X}_{var} \longrightarrow \\
 & & \mathsf{Abs}_{exp}(env, M, e) \neq \mathsf{X}_{exp} \\
\mathsf{wellform}_{stm}(env, M, S_1 \parallel S_2) & \equiv & \mathsf{wellform}_{stm}(env, M, S_1) \wedge \mathsf{wellform}_{stm}(env, M, S_2) \\
\mathsf{wellform}_{stm}(env, M, \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) & \equiv & \mathsf{safe}_{form}(env, M, b) \wedge \mathsf{wellform}_{stm}(env, M, S_1) \wedge \\
 & & \mathsf{wellform}_{stm}(env, M, S_2) \\
\mathsf{wellform}_{stm}(env, M, \|_{i=1}^{N} S(i)) & \equiv & \forall i.\ \mathsf{boundAssign}(i, S(i)) \\
 & & \wedge \mathsf{wellform}_{stm}(env, M, S(i)) \\
\mathsf{wellform}_{stm}(env, M, \|_{i=1}^{N} i \neq \mathsf{n} \longrightarrow S(i)) & \equiv & \forall i.\ \mathsf{boundAssign}(i, S(i)) \\
 & & \wedge \mathsf{wellform}_{stm}(env, M, S(i))
\end{array}$$

The explanations of these well-formedness conditions are as follows. For a single assignment, if the left side does not abstract to an unspecified variable, then the right side must not abstract to an unspecified expression. This would forbid, for example, assigning a global variable to an expression that depends on information in node i, as the abstraction for $i > M$ would not know what value to assign. For forall-statements, we require all assignments in $S(i)$ to assign a variable $v[i]$ to an expression that depends only on node i. If assignments $S(i)$ involve other nodes, then reducing N to M would not be safe as the first M nodes may involve assignments on node i with $i > M$.

A rule $r = g \rhd S$ is well-formed, denoted by $\mathsf{wellform}_{rule}(env, M, r)$, if its statement S s.t. $\mathsf{wellform}_{stm}(env, M, S)$.

Now we present rules for abstracting statements. The key cases are shown below.

$$\begin{array}{rcl}
\mathsf{Abs}_{stm}(env, M, v := e) & \equiv & \mathsf{if}\ \mathsf{Abs}_{var}(env, M, v) = \mathsf{X}_{var}\ \mathsf{then\ skip} \\
 & & \mathsf{else}\ v := \mathsf{Abs}_{exp}(env, M, e) \\
\mathsf{Abs}_{stm}(env, M, S_1 \parallel S_2) & \equiv & \mathsf{let}\ S_1' = \mathsf{Abs}_{stm}(env, M, S_1)\ \mathsf{in} \\
 & & \mathsf{let}\ S_2' = \mathsf{Abs}_{stm}(env, M, S_2)\ \mathsf{in} \\
 & & \mathsf{if}\ S_1' = \mathsf{skip\ then\ if}\ S_2' = \mathsf{skip\ then\ skip\ else}\ S_2'\ \mathsf{else} \\
 & & \mathsf{if}\ S_2' = \mathsf{skip\ then}\ S_1'\ \mathsf{else}\ S_1' \parallel S_2' \\
\mathsf{Abs}_{stm}(env, M, \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) & \equiv & \mathsf{let}\ S_1' = \mathsf{Abs}_{stm}(env, M, S_1)\ \mathsf{in} \\
 & & \mathsf{let}\ S_2' = \mathsf{Abs}_{stm}(env, M, S_2)\ \mathsf{in} \\
 & & \mathsf{if}\ S_1' = \mathsf{skip} \wedge S_2' = \mathsf{skip\ then\ skip\ else} \\
 & & \mathsf{if}\ \mathsf{Abs}_{form}(env, M, b)\ \mathsf{then}\ S_1'\ \mathsf{else}\ S_2' \\
\mathsf{Abs}_{stm}(env, M, \|_{i=1}^{N} S) & \equiv & \|_{i=1}^{M} S \\
\mathsf{Abs}_{stm}(env, M, \|_{i=1}^{N} i \neq \mathsf{n} \longrightarrow S(i)) & \equiv & \|_{i=1}^{M} i \neq \mathsf{n} \longrightarrow S(i)\ \mathsf{if}\ \mathsf{n} \leq M\ \mathsf{otherwise}\ \|_{i=1}^{M} S
\end{array}$$

Here, assignments are removed (abstracted to skip) if they assign to variables that have been abstracted away. All other assignments are kept. The abstraction rule for forall-statements changes the range to M.

The following lemma shows that abstraction of statements commutes with abstraction on states:

Lemma 5.6. *Given $M \leq N$ and a statement S that satisfies* $\mathsf{wellform}_{stm}(env, N, S)$*, $env \Vdash S$ and* $\mathsf{fitEnv}(s, env)$*. The following equality holds:*

$$\mathsf{Abs}_{st}(M, \mathsf{trans}(S, s)) = \mathsf{trans}(\mathsf{Abs}_{stm}(env, M, S))(\mathsf{Abs}_{st}(M, s)).$$

Based on the abstraction of formulas and statements, we can define abstraction of a rule.

$$\mathsf{Abs}_{rule}(env, M, g \triangleright a) \equiv \mathsf{Abs}_{form}(env, M, g) \triangleright \mathsf{Abs}_{stm}(env, M, a).$$

Running example. Now we use the abstraction operator Abs_{rule} on the rules in $\mathcal{M}_{ref}(N)$, we have the following results:

$$\begin{array}{rcll}
\mathsf{Abs}_{rule}(env, M, \mathsf{r}_{ref}(i)) & \equiv & \mathsf{r}_{ref}(i) & \text{if } i \leq M \\
\mathsf{Abs}_{rule}(env, M, \mathsf{try}_{ref}(i)) & \equiv & \mathsf{skip\_rule} & \text{if } i > M \\
\mathsf{Abs}_{rule}(env, M, \mathsf{crit}_{ref}(i)) & \equiv & \mathsf{abs\_crit} & \text{if } i > M \\
\mathsf{Abs}_{rule}(env, M, \mathsf{exit}_{ref}(i)) & \equiv & \mathsf{skip\_rule} & \text{if } i > M \\
\mathsf{Abs}_{rule}(env, M, \mathsf{idle}_{ref}(i)) & \equiv & \mathsf{abs\_idle} & \text{if } i > M
\end{array}$$

where $\mathsf{r}_{ref} \in \{\mathsf{try}_{ref}, \mathsf{crit}_{ref}, \mathsf{exit}_{ref}, \mathsf{idle}_{ref}\}$, $\mathsf{skip\_rule} \equiv \mathsf{True} \triangleright \mathsf{skip}$

$$\begin{array}{rl}
\mathsf{abs\_crit} \equiv x & \triangleright\ x := \mathsf{False} \\
\mathsf{abs\_idle}(M) \equiv \ (\bigwedge_{j=1}^{M} st[j] \neq \mathsf{C} \wedge st[j] \neq \mathsf{E}) & \triangleright\ x := \mathsf{True}
\end{array}$$

For $i \leq M$ abstraction on all rules changes nothing. For $i > M$, abstraction removes rules try and exit completely. The rule crit is abstracted into $x \triangleright x := \mathsf{False}$. This indicates that the only observable behavior of applying crit on a node with index $i > M$ is changing the global variable x from True to False. The transition on the state of the node is unobserved. For abstraction of the rule idle, the bound on the forall-condition in the guard is changed from N to M, indicating that only the condition on the first M nodes is observed. Note if we used the simple form of guard strengthening in Section 4, the entire guard would be lost as the condition $st[i] = E$ is abstracted to unknown, which would have rendered the strengthening useless. With the more refined strengthening, the key constraint in the guard is preserved, which is just what is needed to allow the abstracted protocol pass model checking.

The strengthened and abstracted protocol rules are collected into the protocol $\mathcal{AM}(M)$ as follows. This is the form of the protocol sent to model checking when applying CMP.

$$\begin{array}{rcl}
\mathcal{AR}(M) & \equiv & \{\mathsf{try}_{ref}(i) \mid i \leq M\} \cup \{\mathsf{crit}_{ref}(i) \mid i \leq M\} \cup \{\mathsf{exit}_{ref}(i) \mid i \leq M\} \cup \\
 & & \{\mathsf{idle}_{ref}(i) \mid i \leq M\} \cup \{\mathsf{abs\_crit}, \mathsf{abs\_idle}(M), \mathsf{skip\_rule}\} \\
\mathcal{AM}(M) & \equiv & \langle \{\mathsf{init}_1, \mathsf{init}_2(M)\}, \mathcal{AR}(M) \rangle
\end{array}$$

Discussion. We compare the abstraction procedure given here with existing work. The work of Krstic [22] gave a relatively formal description of the CMP method. Talupur et al. [39] described an abstract version of the CMP method, and showed how it can be combined with generation of auxiliary invariants from message flow diagrams. In both works, the description of the abstraction procedure is informal, and misses many of the required details. In [39], abstraction is characterized as replacing any condition involving processors greater than M with either True, False, or nondeterministic variable to overapproximate it, and removing any assignment to state variables with index greater than M. The work [22] gives more details, describing the changes made to each of four

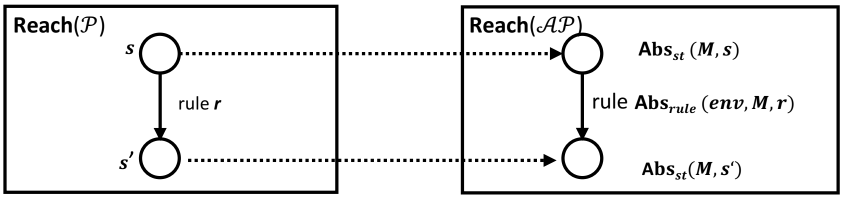

Fig. 3. An illustration of the commutation graph of the abstraction mapping.

forms of assignments: $v := e, f[p_i] := e, g[e] := p_j$ and $h[p_i] := p_j$, and defines the replacement of predicates depending on whether it appears positively or negatively in the condition. None of these existing descriptions of CMP discusses how to handle expressions of the form $i \neq j$ appearing in conditions, or how to abstract forall-conditions and forall-statements (as well as versions excluding a certain index). In contrast, we formally state the abstraction rules for all these cases, so the entire abstraction procedure is syntax-directed. This enables completely automatic abstraction that is also proved correct in Isabelle.

5.3 Simulation of Two Protocols

Based on the above definitions, we can show that the abstracted protocol represents a *simulation* of the original protocol, in the sense that any transition in the original protocol corresponds to a transition in the abstracted protocol (but the abstracted protocol may allow more transitions). This is formally defined as follows:

Definition 5.7. Given two protocol representations $\mathcal{P} = \langle I, R \rangle$ and $\mathcal{P}' = \langle I', R' \rangle$, and an abstraction map $\mathsf{Abs}_{st}(M, \cdot)$, we say $\mathcal{P}'$ is a *simulation* of $\mathcal{P}$, if the following two conditions hold:

(1) For any formula $f \in I$, there exists a formula $f' \in I'$ such that for any state s, $s \models f$ implies $\mathsf{Abs}_{st}(M, s) \models f'$.
(2) For any rule $(g \rhd a) \in R$, there exists a rule $(g' \rhd a') \in R'$, such that for any state s, if $s \models g$, then $\mathsf{Abs}_{st}(M, s) \models g'$ and $\mathsf{Abs}_{st}(M, \mathsf{trans}(a, s)) = \mathsf{trans}(a', \mathsf{Abs}_{st}(M, s))$, as illustrated in Figure 3.

We use $f(R)$ to denote the image of function f under set R. From the properties of abstraction on formulas and statements given in the previous section, we can formally verify the following result:

Lemma 5.8. *Given $M \leq N$ and a protocol presentation $\mathcal{P} = \langle I, R \rangle$, let $I' = \mathsf{Abs}_{form}(env, M, I)$, and $R' = \mathsf{Abs}_{rule}(env, M, R)$, $env \Vdash f$ for any $f \in I$, and $env \Vdash r$ for any $r \in R$. Then $\mathcal{P}' = \langle I', R' \rangle$ is a simulation of $\mathcal{P}$; furthermore, if $\mathsf{fitEnv}(s, env)$ for all $s \in \mathsf{Reach}(\mathcal{P})$, then $s \in \mathsf{Reach}(\mathcal{P})$ implies $\mathsf{Abs}_{st}(M, s) \in \mathsf{Reach}(\mathcal{P}')$.*

6 MAIN THEOREM

In this section, we formally state the main theorem for the correctness of CMP. One additional technical detail is necessary for this statement. We make use of two forms of auxiliary invariants. Model checking can only check invariants of the form $f'(i, j)$ for $i < j \leq M$ on the abstracted protocol, which is to be extended to $i, j \leq N$ by symmetry arguments on the strengthened protocol. Here, $f'(i, j)$ is symmetric with respect to the two parameters i and j. On the other hand, for strengthening the protocol before abstraction, the auxiliary invariants are required to be in the form $f(j, i)$, that is symmetric with respect to the single parameter i, and j is a dummy parameter. We call the former form (used in model checking) *invariants for observation*, and the latter form (used in strengthening) *invariants for strengthening*.

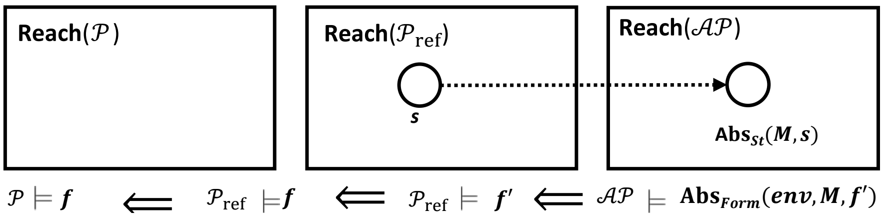
Reach($\mathcal{P}$)
Reach($\mathcal{P}_{\text{ref}}$)
Reach($\mathcal{AP}$)
s
$\mathbf{Abs}_{St}(M, s)$
$\mathcal{P} \models f \Longleftarrow \mathcal{P}_{\text{ref}} \models f \Longleftarrow \mathcal{P}_{\text{ref}} \models f' \Longleftarrow \mathcal{AP} \models \mathbf{Abs}_{Form}(env, M, f')$

Fig. 4. An illustration of the proof sketch of CMP.

We define a relation linking these two forms of auxiliary invariants as $\mathsf{strVSobs}(f, f', N)$, where f is the invariants for strengthening, f' is the invariants for observation.

$$
\begin{aligned}
&\mathsf{strVSobs}(f, f', N) \equiv \\
&(\exists ant\ cons.\ \text{both } ant(i) \text{ and } cons(j) \text{ are symmetric} \land \\
&(f(N, j, i) = (ant(i) \longrightarrow \bigwedge_{k=1}^{N} k \neq i \longrightarrow cons(k)) \lor \\
&f(N, i, j) = (ant(i) \longrightarrow \bigwedge_{k=1}^{N} k \neq i \longrightarrow cons(k))) \land \\
&f'(i, j) = (i \neq j \land ant(i)) \longrightarrow cons(j)
\end{aligned}
$$

For example, the auxiliary invariant $\mathsf{strExit}(N, i)$ in Section 4 is originally given in strengthening form as follows:

$$\mathsf{strExit}(N, j, i) \equiv st[i] = \mathsf{E} \longrightarrow \bigwedge_{k=1}^{N} k \neq i \longrightarrow st[k] \neq \mathsf{C} \land st[k] \neq \mathsf{E}.$$

The corresponding invariant for observation is:

$$\mathsf{obsExit}'(i, j) \equiv i \neq j \land st[i] = \mathsf{E} \longrightarrow st[j] \neq \mathsf{C} \land st[j] \neq \mathsf{E}.$$

For model checking, we only need to check $\mathsf{strExit}'(1, 2)$ in the abstracted protocol model. We also note that since the invariants for observation are *safe*, we can guarantee that the positive result of model checking invariants for observation in abstracted model implies the invariants for strengthening in parameterized strengthened protocol model. In general, $f'(i, j)$ being an invariant for $i < j \leq M$ implies $f(N, i, j)$ being an invariant for any $i \leq N, j \leq N$. This is stated in the following lemma, which will be applied to $\mathcal{P}_{ref}$ (the protocol after strengthening and before abstraction) in the proof of the main theorem.

Lemma 6.1. *If $\mathcal{P}$ is symmetric, and $\mathcal{P} \models f'(i, j)$, $i < j \leq M$, and* $\mathsf{strVSobs}(f, f', N)$*, then $\mathcal{P} \models f(i, j)$ for any $i \leq N, j \leq N$.*

Now we are ready to state the main theorem of the CMP theory.

Theorem 6.2. *Given $2 \leq M \leq N$, F and F' are set of invariants for strengthening and observation, env is the type environment, $\mathcal{P} = \langle I, R \rangle$, $\mathcal{P}_{ref} = \langle I, R_{ref} \rangle$, $\mathcal{AP} = \langle AI, AR \rangle$, and if the following eight conditions hold:*

(1) $\mathcal{P}_{ref}$ is guard-strengthened by $\mathcal{P}$ with F;
(2) $\mathsf{Abs}_{rule}(env, M, R_{ref}) \subseteq AR$, and $\mathsf{Abs}_{form}(env, M, I) = AI$;
(3) for any $f' \in F'$, any $i < j \leq M$, $\mathcal{AP} \models \mathsf{Abs}_{form}(env, M, f'(i, j))$;
(4) for any $f \in F$, there is an $f' \in F'$ s.t. $\mathsf{strVSobs}(f, f', N)$ or $f = f'$;[2]
(5) for any $f' \in F'$, s.t. $\mathsf{safe}_{form}(env, M, f')$;

[2]For the running example, only $\mathsf{strVSobs}(f, f', N)$ is needed, but in the FLASH protocol, it also can be $f = f'$.

(6) for any s, $s \in \mathsf{Reach}(\mathcal{P}_{ref})$ implies $\mathsf{fitEnv}(s, env)$;
(7) $env \Vdash r$ and $\mathsf{wellform}_{rule}(env, M, r)$ for any $r \in R_{ref}$;
(8) $\mathcal{P}_{ref}$ is symmetric and f is symmetric for any $f \in F$.

Then for any $f \in F$, any $i \le N$, $j \le N$, $\mathcal{P} \models f(i, j)$.

Proof. For any state s such that $s \in \mathsf{Reach}(\mathcal{P}_{ref})$, from condition (2), by Lemma 5.8, we have $\mathsf{Abs}_{st}(s, env, M) \in \mathsf{Reach}(\mathcal{AP})$, which is illustrated by the dash right arrow in Figure 4. From condition (3), $\mathsf{Abs}_{st}(s, env, M) \models \mathsf{Abs}_{form}(env, M, f'(i, j))$ for any $f' \in F'$, where $i < j \le M$, with conditions (5) (6) (7), by Lemma 5.4(3), we have $s \models f'(i, j)$. That is to say, $\mathcal{P}_{ref} \models f'(i, j)$ for any $i < j \le M$ and this is illustrated by the first left arrow[3] in Figure 4. For any $f \in F$, by condition (4), we have a counter-part formula f' of f s.t. $\mathsf{strVSobs}(f, f', N)$, $f'(i, j) \in F'$. From (8), by Lemma 6.1, we have $\mathcal{P}_{ref} \models f(i, j)$ for any $i, j \le N$. This is illustrated by the second left arrow in Figure 4. With condition (1), by Lemma 4.2, $\mathcal{P} \models f(i, j)$. This is illustrated by the third left arrow in Figure 4. □

Running example. In order to verify $\mathcal{M}(N)$, we need to define $F \equiv \{\mathsf{strExit}(N, j, i) \mid 1 \le i, j \le N\} \cup \{\mathsf{inv}(j, i) \mid 1 \le i, j \le N\}$, and $F' \equiv \{\mathsf{obsExit}'(i, j) \mid 1 \le i < j \le N\} \cup \{\mathsf{inv}'(i, j) \mid 1 \le i < j \le N\}$, where $\mathsf{strExit}(N, j, i)$ and $\mathsf{inv}(j, i)$ are used to strengthen invariants, and $\mathsf{inv}(j, i)$ is the original invariant to be verified; $\mathsf{obsExit}'(1, 2)$ and $\mathsf{inv}'(1, 2)$ are the corresponding invariants for observation. $\mathcal{M}_{ref}(N)$ and $\mathcal{AM}(M)$ are the strengthened and abstracted mutual exclusion protocol. We also need to translate $\mathcal{AM}(M)$ into a form which can be accepted by Murphi, and use the model checker to verify whether any $f \in F'$ is an invariant. As the model checking result on $\mathcal{AM}$ is positive, the condition (3) is verified. Other conditions in the theorem are proved in Isabelle one by one. Therefore, we conclude any $f \in F$ is an invariant in $\mathcal{M}(N)$ by Theorem 6.2.

7 IMPLEMENTATION AND CASE STUDIES

All definitions and theorems presented in the above sections, including Theorem 6.2 which accounts for the final correctness statement of CMP, are formalized in the Isabelle proof assistant. The total line count for this formalization is 5,384 lines. Values, variables, expressions, formulas and statements are represented as Isabelle datatypes (abstract syntax trees), using lambda expressions for the quantified formulas and statements. The line count indicates that with these choices of representations, the eventual proof effort is rather modest for authors who are already familiar with Isabelle. However, the entire project still went on-and-off for about two years, with most of the time spent working out the details of the theory such as the abstraction process, type system, and various safe conditions. Hence we consider these to be one of the main contributions of the paper.

7.1 An Overview of The Verification Framework AutoCMP

As a natural outcome of the verification, we obtain an automatic, syntax-directed procedure for abstraction and strengthening, ultimately generating proofs that can be checked using Isabelle/HOL. We implemented a tool AutoCMP [26] in Python to automate the CMP method. The tool accepts user input in the form of the original protocol to be verified, list of auxiliary invariants, and which auxiliary invariants are applied to strengthen each rule. The core functionality of AutoCMP is the automation of guard strengthening and abstraction procedures. From the user inputs, it automatically generates the strengthened and abstracted protocol, as well as the invariants for observation in a new protocol file, which is sent to be model checked by Murphi. If model checking succeeds, the tool also automatically generates a proof script for Isabelle to certify the correctness

[3] The order is from right to left.

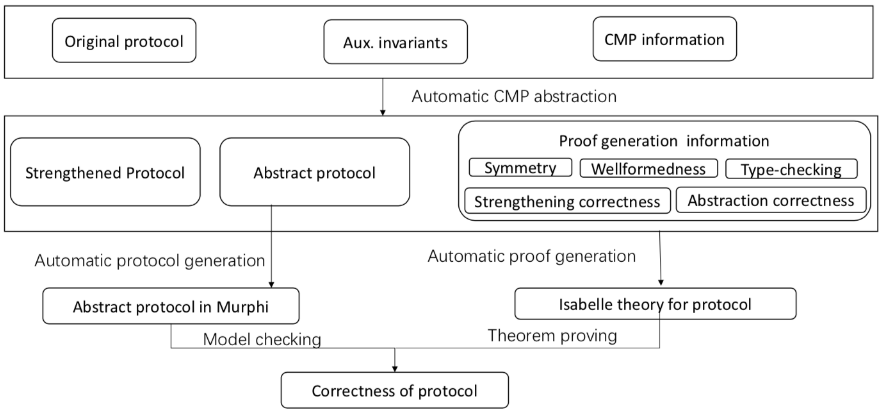

Fig. 5. Combination of model checking, abstraction and theorem proving in AutoCMP.

of running strengthening and abstraction procedures for the protocol (the implementation of the procedures in Python are not verified, as it is much easier to verify the generated results for each input protocol). The conclusion of the generated Isabelle script states that the parameterized protocol satisfies the original invariants to be checked, on the assumption of that the finite, abstracted protocol is correct. When combined with model checking by Murphi, this gives the correctness of the original protocol. An overview of the verificaiton framework is presented in Figure 5. The trusted code base (TCB) of AutoCMP include both the Isabelle theorem prover and the Murphi model checker. We also assume the consistency between the representation of the parameterized system in Isabelle and Murphi.

Proof generation in Isabelle is also implemented in a Python module. It depends on a set of utilities for construction and representation of Isabelle types, terms, and proof tactics, which can be output to formatted Isabelle text. This set of utilities can potentially be reused for other projects where automatic generation of Isabelle proof script is needed. Based on these, the main work of proof generation is constructing proofs showing symmetry and wellformedness of invariants and rules, as well as typing conditions on the protocol description. The strengthening and abstraction procedures are themselves defined and proved correct in Isabelle, so for proof generation, it suffices to prove that the output of Python implementation agrees with applying the procedures in Isabelle. Overall, proof generation is feasible because all our procedures, including those for checking symmetry, wellformedness, and typing conditions, are syntax-directed, so they can be achieved by fixed Isabelle automation tactics.

The above workflow assumes that information about auxiliary invariants are already available, for example when reproducing existing work, or checking results of automatic generation of auxiliary invariants by machine learning (such methods have been investigated by others, e.g. in [4, 5, 25], but incorporating these techniques is outside the scope of this work). In the practical scenario of verifying a new protocol by hand, the auxiliary invariants are initially unknown. Then AutoCMP enables a workflow as illustrated by Figure 6. The original protocol is first abstracted (without any strengthening) using AutoCMP and sent to model checking by Murphi. If model checking produces a counterexample, it is analyzed by the user to determine a rule that should be strengthened. The rule and auxiliary invariant chosen by the user are then added to the input to AutoCMP, which automatically strengthens and abstracts the protocol again. This process iterates until either a real counterexample is found, or when the user is unable to find a valid auxiliary

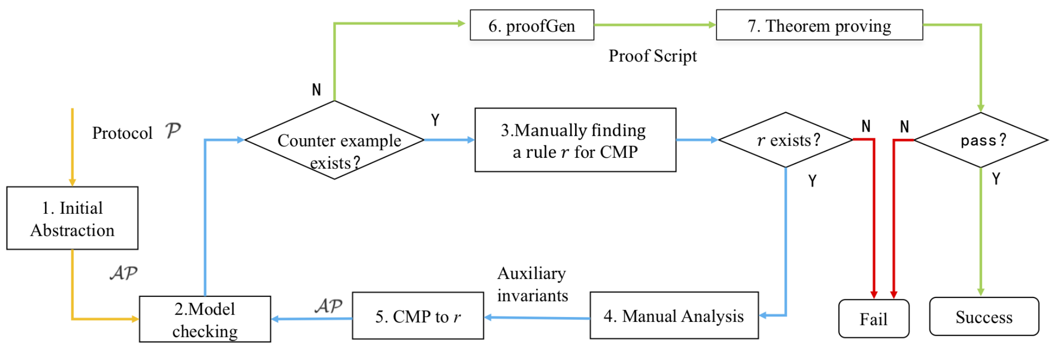

Fig. 6. An illustration of the workflow of AutoCMP to automate the CMP method. CMP procedures are done automatically, which is highlighted now by a concrete box.

invariant. The main advantage provided by AutoCMP compared to the traditional manual workflow is that strengthening and abstraction is completely automatic, saving the effort (as well as risk of making mistakes) in manually editing the protocol descriptions. Finally, upon success the tool also produces Isabelle proof scripts certifying the correctness of the entire process. There are two different failing cases. The first case leads from "r exists" to "Fail", meaning that no rule can be found for strengthening. This indicates that either the CMP method is not suitable for verifying the protocol, or additional changes (e.g. adding ghost variables) are needed before applying the CMP method. The second case leads from "pass?" to "Fail". This indicates that automatic proof generation for CMP has failed. This may occur during the development process, and more proof patterns need to be added if this occurs.[4] The CEGAR loop for introducing auxiliary invariants is the loop containing boxes 2, 3, 4 and 5, not including the "Fail" case.

7.2 Level of Automation Compared to The State-of-art

In this and the next section, we evaluate the AutoCMP tool on some typical benchmarks of cache coherence protocols. In comparison with earlier work applying the CMP method, we demonstrate both increased level of automation and precision in applying the rules.

We applied the tool to four protocols that commonly serve as benchmarks for the CMP method. The mutual exclusion protocol is used as a running example in this paper. The German and FLASH protocols are used as the main examples in [8]. The detailed code and experiment data can be found at the website of our verification framework [26]. Each experiment directory includes the input files to AutoCMP: the original Murphi model, the auxiliary invariants, and the strengthening information for each rule of the protocol. We also include the generated abstracted protocol and Isabelle proof scripts. All the experiments were conducted on an iMac workstation with an Intel i5 processor, 16 GiB memory and 64-bit macOs High Sierra. Statistics for each of the case studies are summarized in Table 2. The results show that the number of auxiliary invariants is relatively small (in the actual input files, the invariants are further merged, resulting in e.g. only six lemmas for the Flash protocol). Model checking using Murphi is relatively fast. Checking the generated proofs in Isabelle is more time consuming. However, these only need to be done once at the end of verification process.

[4]For the final version of the proof generator, this case is not expected to happen, and indeed it does not occur in the examples we tested.

Table 2. Statistics for case studies. The #rules column contains the total number of protocol rules. The #lines is the number of lines in the generated Isabelle proof script, and the #invariants is the number of invariants before merging. The column 'TP time' presents the time consumed by the theorem proving in Isabelle, while the column 'MC time' further indicates the amount of time spent on model checking the abtracted protocol.

Protocol	#rules	#lines	#invariants	TP time (seconds)	MC time (seconds)
mutualEx	4	740	2	5.04	0.01
MESI	4	955	6	8.957	0.01
German	12	1,826	3	43.69	0.01
FLASH	40	6,497	36	1,565.56	8.41

In [8], the parameter abstraction and guard strengthening procedures in CMP method are done manually. No mechanized proofs are provided to justify the foundation of CMP and the simulation of a parameterized protocol and the counterpart abstraction. For Flash protocol, the authors argued they need 0.5 day to do the above CMP work to work out an abstraction. Compared to purely interactive verification of concurrent systems (i.e. without using the CMP method), we only require auxiliary invariants, which are much less complicated than the fully inductive invariants. We are also able to automatically verify the sufficiency of auxiliary invariants through model checking.

In our work, the parameter abstraction and guard strengthening procedures in CMP method are done automatically (once a protocol and the auxiliary invariants are provided). Not only is the abstraction protocol generated automatically, but also an Isabelle proof script is generated to verify the abstraction relation between the abstraction result and the original parameterized protocol. The proof scripts can be checked automatically in Isabelle.

7.3 Increased Precision for Handling Complex Real-world Protocols

Among our benchmarks, FLASH is more complex than the others and closer to being a real-world protocol. Therefore, FLASH is a good test for any proposed method of parameterized verification: if the method works on FLASH, then there is a good chance that it will also work on many real-world cache coherence protocols.

The first verification of FLASH was achieved by Park and Dill in [35], by providing the needed inductive invariants manually. Chou et al. [8] introduced their CMP method to verify safety properties of FLASH. The application of CMP to the FLASH protocol is complicated by the fact that the protocol contains rules with two parameters (hence there are 4 cases of abstraction for these rules), and there are many auxiliary invariants, in several different forms. The model of the FLASH protocol used by Chou et al. contains 33 rules. There are three main invariants to be checked: mutual exclusion and two other data invariants. A total of 36 auxiliary invariants are used, which for simplicity are merged into five according to their antecedents.

We applied our abstraction and strengthening procedures to the FLASH model, and discovered that the original abstraction performed by Chou et al. contains two mistakes and one place where abstraction is applied in a highly irregular way. We briefly describe these problems below, and more details are available [26].

- In rules `NI_LOCAL_GET_GET` and `NI_LOCAL_GETX_GETX`, their guard contains the following conjunct $Dir.HeadPtr \neq src$, and this was abstracted to $Dir.HeadPtr \neq Other$. However, this is not a conservative abstraction as discussed in Section 5. We removed this conjunct from the abstraction of these two rules.
- In rules `NI_Local_GetX_PutX` and `NI_InvAck`, there are if-then-else statements where the condition is not a safe formula. The abstractions of these rules in [8] are not thus sound with respect

to our theory. We rewrote the rules by splitting them into multiple rules, and obtain sound abstractions.

- The abstraction for the rule `NI_ShWb`, while correct in hindsight, is highly irregular. The assigned values in the rule involves $p = ShWbMsg.Proc$. This is abstracted away, while adding a guard $ShWbMsg.Proc = Other$. We clarified the situation by changing the rule to be parameterized by an index *src*, and adding the guard $ShWbMsg.Proc = src$. This does not change the semantics of the model.

The two mistakes resulted in the abstracted model having a smaller state space than it should (158466 vs. 183596 for the corrected model). While model checking on the corrected model still passes (no errors are found in the increased state space), so the general method of applying CMP to FLASH protocol is valid, it does point to the danger of manually performing abstraction, which is eliminated by using the AutoCMP tool. For the rule `NI_ShWb`, while the original abstraction is correct, it may be difficult to understand for others reviewing the work. Using our tool clarifies the issue and make the abstraction process fully transparent.

8 CONCLUSION AND FUTURE WORK

In this work, we provided for the first time the rigorous formalization of CMP for parameterized verification in Isabelle/HOL. This includes concepts of symmetry, assume-guarantee reasoning, type-checking on the protocol descriptions, and syntax-directed procedures for abstraction and strengthening. All of the above have been proved correct in Isabelle. In particular, for the abstraction process, we introduced additional side conditions such as safe expressions and formulas. The use of a theorem prover like Isabelle forces the entire theory to be completely precise, and revealed more tricky details in CMP than what would appear on first sight. Our work results in a tool AutoCMP for automatically applying the CMP method. After providing the auxiliary invariants, the tool automatically performs strengthening and abstraction, as well as generation of proof scripts that can be checked in Isabelle. In the case studies, we applied the tool to four parameterized protocols, which led to the discovery of some mistakes in the original manual abstraction of the FLASH protocol. Although we illustrated the applicability of our mechanization only for cache coherence protocols, the CMP method itself can be applied to a broad range of parameterized protocols (e.g., distributed algorithms, multithreaded programs, hierarchical multicore protocols, etc.), so our verification framework AutoCMP can potentially be applied to these areas as well.

Formalization of CMP is surprisingly difficult, with no essential progress made since the problem was proposed by Chou, Mannava and Park in [8] in 2004. The proof is very challenging because CMP appears alarmingly circular, and the abstraction/strengthening process is not described in precise terms in the past. We address these problems with carefully chosen assume-guarantee arguments and abstraction methods based on type information. We believe our work provides experience in formalizing model checking techniques such as CMP, and many of the components, such as symmetry and assume-guarantee reasoning, may provides guidance on formalizing similar methodologies as well. Finally, we learn through the work that, as in the example of CMP, a method that is not formalized may always contain subtle, unrealized flaws.

There are a number of directions for possible future work. First, we would like to extend the formalism to be able to handle hierarchical multicore protocols [7, 30], which is known to require special care in the strengthening procedure for statements and introduce new dimension of symmetry. We also plan to link the formalization with verification of the model-checking process itself, for example along the lines of [17]. Finally, we will consider linking our tool with automatic methods of producing auxiliary invariants, for example, those based on machine learning.

9 DATA-AVAILABILITY STATEMENT

The development of AutoCMP can be found at [26]. It is also available as a verified artifact in https://zenodo.org/doi/10.5281/zenodo.10464461.

ACKNOWLEDGMENTS

This research was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320000 and XDA0320300. The authors would like to express their gratitude to the anonymous reviewers for their valuable and insightful feedback, which greatly contributed to improving the quality of the paper.

REFERENCES

[1] Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. 2018. Model Checking Parameterized Systems. In *Handbook of Model Checking*. 685–725.

[2] Krzysztof R. Apt and Dexter Kozen. 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inform. Process. Lett.* 22, 6 (1986), 307–309.

[3] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. 2001. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proc. 13th International Conference on Computer Aided Verification (CAV'01) (Lecture Notes in Computer Science, Vol. 2102)*. Springer, 221–234.

[4] Jialun Cao, Yongjian Li, and Jun Pang. 2018. L-CMP: An automatic learning-based parameterized verification tool (Tool Demo). In *Proc. 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. ACM Press, 892–895.

[5] J. Cao, Y. Li, and J. Pang. 2019. A learning-based framework for automatic parameterized verification. In *Proc. 37th IEEE International Conference on Computer Design (ICCD'19)*. IEEE CS, 450–459.

[6] Xiaofang Chen and Ganesh Gopalakrishnan. 2006. *A General Compositional Approach to Verifying Hierarchical Cache Coherence Protocols*. Technical Report. Technical Report, School of Computing, University of Utah.

[7] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. 2006. Reducing Verification Complexity of a Multicore Coherence Protocol using Assume/guarantee. In *Proc. 6th International Conference on Formal Methods in Computer Aided Design (FMCAD'06)*. IEEE Computer Society, 81–88.

[8] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. 2004. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Proc. 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04) (Lecture Notes in Computer Science, Vol. 3312)*. Springer, 382–398.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (2003), 752–794.

[10] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of Model Checking*. Springer.

[11] Ariel Cohen and Kedar S. Namjoshi. 2009. Local Proofs for Global Safety Properties. *Formal Methods in System Design* 34, 2 (2009), 104–125.

[12] Sylvain Conchon, David Declerck, and Fatiha Zaïdi. 2018. Cubicle-W: Parameterized Model Checking on Weak Memory. In *Proc. 9th International Joint Conference on Automated Reasoning (IJCAR'18) (Lecture Notes in Computer Science, Vol. 10900)*. 152–160.

[13] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. 2012. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper. In *Proc. 24th International Conference on Computer Aided Verification (CAV'12) (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 718–724.

[14] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. 2013. Invariants for Finite Instances and Beyond. In *Proc. 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD'13)*. IEEE Computer Society, 61–68.

[15] David L. Dill. 1996. The Mur*phi* Verification System. In *Proc. 8th International Conference on Computer Aided Verification (CAV'96) (Lecture Notes in Computer Science, Vol. 1102)*. Springer, 390–393.

[16] Michael Emmi, Rupak Majumdar, and Roman Manevich. 2010. Parameterized Verification of Transactional Memories. In *Proc. 31th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM Press, 134–145.

[17] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. 2013. A Fully Verified Executable LTL Model Checker. In *Proc. 25th International Conference on Computer Aided Verification (CAV'13) (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 463–478.

[18] Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. 2017. Thread Modularity at Many Levels: A Pearl in Compositional Verification. In *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM Press, 473–485.

[19] Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. 1984. Towards Reachability Trees for High-level Petri Nets. In *Proc. 6th European Workshop on Applications and Theory in Petri Nets (Lecture Notes in Computer Science, Vol. 188)*. Springer, 215–233.

[20] C. Norris Ip and David L. Dill. 1993. Efficient Verification of Symmetric Concurrent Systems. In *Proc. International Conference on Computer Design (ICCD'93)*. IEEE Computer Soceity, 230–234.

[21] C. Norris Ip and David L. Dill. 1996. Better Verification through Symmetry. *Formal Methods in System Design* 9, 1/2 (1996), 41–75.

[22] Sava Krstic. 2005. Parameterized System Verification with Guard Strengthening and Parameter Abstraction. *Proc. 4th Workshop on Automated Verification of Infinite State Systems (AVIS'05)* (2005).

[23] Shuvendu K. Lahiri and Randal E. Bryant. 2004. Constructing Quantified Invariants via Predicate Abstraction. In *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04) (Lecture Notes in Computer Science, Vol. 2937)*. Springer, 267–281.

[24] Yongjian Li, Kaiqiang Duan, Yi Lv, Jun Pang, and Shaowei Cai. 2016. A Novel Approach to Parameterized Verification of Cache Coherence Protocols. In *Proc. 34th IEEE International Conference on Computer Design (ICCD'16)*. IEEE Computer Society, 560–567.

[25] Yongjian Li and Zimin Li. 2022. ILCMP source code. https://gitee.com/dust_capacity_i/ILCMP.git

[26] Yongjian Li, Zimin Li, Bohua Zhan, and Jun Pang. 2023. autoCMP. https://github.com/forSubmission238/autoCMP.

[27] Kenneth L. McMillan. 1999. Verification of Infinite State Systems by Compositional Model Checking. In *Proc. 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99) (Lecture Notes in Computer Science, Vol. 1703)*. Springer, 219–234.

[28] Kenneth L. McMillan. 2001. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Proc. 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01) (Lecture Notes in Computer Science, Vol. 2144)*. Springer, 179–195.

[29] Kenneth L. McMillan. 2008. Quantified Invariant Generation using An Interpolating Saturation Prover. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08) (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 413–427.

[30] Kenneth L. McMillan. 2016. Modular Specification and Verification of a Cache-coherent Interface. In *Proc. 16th International Conference on Formal Methods in Computer-Aided Design (FMCAD'16)*. IEEE, 109–116.

[31] Kenneth L. McMillan. 2018. Eager Abstraction for Symbolic Model Checking. In *Proc. 30th International Conference on Computer Aided Verification (CAV'18) (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 191–208.

[32] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer.

[33] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM Press, 614–630.

[34] Sudhindra Pandav, Konrad Slind, and Ganesh Gopalakrishnan. 2005. Counterexample Guided Invariant Discovery for Parameterized Cache Coherence Verification. In *Proc. 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'05) (Lecture Notes in Computer Science, Vol. 3725)*. Springer, 317–331.

[35] Seungjoon Park and David L Dill. 1996. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proc. 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*. ACM, 288–296.

[36] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *Proc. 7th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01) (Lecture Notes in Computer Science, Vol. 2031)*. Springer, 82–97.

[37] Amir Pnueli and Elad Shahar. 1996. A Platform for Combining Deductive with Algorithmic Verification. In *Proc. 8th International Conference on Computer Aided Verification (CAV'96) (Lecture Notes in Computer Science, Vol. 1102)*. Springer, 184–195.

[38] Peter H Starke. 1991. Reachability analysis of Petri Nets using Symmetries. *Systems Analysis Modelling Simulation* 8, 4-5 (1991), 293–303.

[39] Murali Talupur and Mark R. Tuttle. 2008. Going with the Flow: Parameterized Verification using Message Flows. In *Proc. 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE Computer Society, 1–8.

[40] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems.

In *Proc. 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM Press, 662–677.

[41] Ashish Tiwari, Harald Rueß, Hassen Saïdi, and Natarajan Shankar. 2001. A Technique for Invariant Generation. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01) (Lecture Notes in Computer Science, Vol. 2031)*. Springer, 113–127.

[42] Pierre Wolper. 1986. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Proc. 13th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'86)*. ACM Press, 184–193.

[43] Pierre Wolper and Vinciane Lovinfosse. 1989. Verifying Properties of Large Sets of Processes with Network Invariants. In *Proc. 1st International Conference on Computer Aided Verification (CAV'89) (Lecture Notes in Computer Science, Vol. 407)*. Springer, 60–80.

Received 21-OCT-2023; accepted 2024-02-24