A Learning-based Framework for Automatic Parameterized Verification

Yongjian Li
Chinese Academy of Sciences,
Institute of Software,
The State Key Laboratory of
Computer Science,
Beijing, China

Jialun Cao
Chinese Academy of Sciences,
Institute of Software,
The State Key Laboratory of
Computer Science,
Beijing, China

Jun Pang
University of Luxembourg
FSTC & SnT
Esch-sur-Alzette, Luxembourg

***Abstract*—Parameterized verification is shown to be a complicated and undecidable problem. The challenge of parameterized verification lies in how to construct appropriate invariants. Designing algorithms to find such invariants automatically has become an active research area since the last decade. With the advent of some recent works, automatically finding invariants has become possible, but most of these invariants are unreadable, making them difficult to be understood by protocol designers and researchers. Therefore, we propose an automatic framework that learns a set of readable and simple invariants to support in protocol design. It takes advantage of association rule learning, and combines the learning algorithm with parameterized verification. It is noteworthy that the gap between machine learning algorithms and parameterized verification seems to be huge, as they rely on statistical learning and symbolic reasoning, respectively. Our framework, however, builds a bridge through association rules and invariants, making their combination possible. Besides, we also propose an invariant-guided strengthening paradigm, providing an innovative perspective to existing abstraction-strengthening methods. Our framework has been successfully applied to several benchmarks, including an industrial-scale protocol *FLASH*.**

***Index Terms*—Formal methods, parameterized verification, machine learning, association rule learning, invariant learning**

I. Introduction

Parameterized concurrent systems are underlying architectures of modern computing systems [1], such as mutual exclusion protocols, bus protocols in embedded systems, networking communication, cache coherence protocols, web services and sensor networks [2]. In general, a parameterized protocol $P(N)$ is defined as a protocol that consists of N instances, where the term *parameterized* refers to the fact that the size of the system is a parameter of the verification problem [2]. The task of verifying such systems is to show their correctness regardless of the number of instances. Although model checking techniques [3]–[5] can verify properties for finite instances of such protocols, as the scale and number of instance of the protocols increase, model checkers reach their limits in both time and memory consumption [1], resulting in the correctness of these protocols of arbitrary sizes unable to be proved. In fact, this problem has been proved to be undecidable [6].

Nevertheless however, due to its practical importance, this problem has kept attracting considerable interests from both model checking and theorem proving communities throughout the last few decades [7]. Many approaches have been proposed in the literature [1], [8]–[15]. At an early stage, model checking for finite-state parameterized protocols was an alternative solution for this problem [16]–[18]. With this approach being well studied, its limitation was also revealed: the time and memory consumption of model checking is unacceptable as the size of the protocols being increased. For instance, to verify the safety properties of Stanford *FLASH* multiprocessor architecture [19] with five instances it took the model checker Murphi more than 24 hours and 20 GB of memory [1]. Although the cut-off technique has been developed in the literature, trying to conclude the protocols' correctness based on small instantiations [20], [21], it still can run into serious problems when the cut-off value itself is too large [22]. In the next stage, some approaches have been proposed to scale for large systems, including compositional verification [23], abstraction methods such as parameterized abstraction [10]–[12], [24], predicate abstraction [25], environment abstraction [26], and inductive assertions [9]. However, all these methods require human effort to provide hand-crafted invariants and guidance, which are often error-prone, especially when the protocol description is long and complex [11].

In the last decade, how to find sufficient and appropriate invariants automatically has gradually been an active research area [15], [25], [27]–[30]. The concept of "invisible invariants" has been proposed in 2001 [8]. It is an automatic technique for parameterized verification. In this method, auxiliary invariants are computed in a finite system instance to aid inductive invariant checking. Combining this idea with parameterized abstraction, Lv *et al.* used a small protocol instance to compute candidate invariants [7]. However, the invariants found by these works are "raw" boolean predicates, which are hard to understand. Later, a SMT-based model checker Cubicle has been proposed [15] and developed [1]. In their works, the BRAB algorithm has been introduced, which can automatically infer invariants and generate Why3-proof certificates for SMT-solvers. It was the first tool that proves automatically the safety properties of *FLASH*. Although these works can automatically verify parameterized protocols, the invariants they found are not readable and understandable for researchers,

let alone outsiders of the research area. Therefore, our target is to develop a framework, which can not only automatically verify parameterized protocols, but also find invariants that are understandable and useful for protocol designers.

To achieve this goal, the invariants need to satisfy certain requirements. First, the form of the invariants needs to be readable and somehow straightforward. Second, the invariants should be used to verify parameterized protocol directly. Third, we hope that the underlying theory of verification is not too complicated, so that the entire verification can be understood by both researchers and outsiders. The underlying theory we exploit is the "parameterized abstraction and guard strengthening" method, also known as the "CMP" method [10], [11]. It has been widely applied to verify large-scale and industrial cache coherence protocols, including Intel's Chipset and *FLASH* protocols, and at the same time, its main idea is relatively simple. For a parameterized protocol $P(N)$, CMP constructs an abstract model $AP(m)$ (m is usually small) to simulate the behavior of the original protocol. Thus, if the properties can be verified with $AP(m)$, then the correctness of the original protocol can be inferred. The soundness of this method is proved in [12]. The difficulty of automatizing CMP lies in how to construct sufficient invariants. In the original paper, the construction is guided by counterexamples and error traces generated by Murphi. Yet, this process highly relies on human guidance, and is hard to automatize. Therefore, instead we propose an invariant-guided strengthening paradigm, which avoids analyzing counterexamples and error traces, so the human effort part can be omitted. The invariants learned from association rule learning can be used in the verification process directly. As a consequence, the aforementioned requirements are satisfied in our framework: it learns invariants and verifies parametrized protocols automatically, and meanwhile, the invariants it infers can help researchers get better insight for protocols and guide their design.

Main contributions. Our contributions in the current paper can be summarized as follows.

- We develop an automatic framework which can verify parameterized protocols automatically and provide readable auxiliary invariants for both protocol designers and researchers.
- We build a bridge between learning algorithms and parameterized verification. To the best of our knowledge, learning algorithms, which is mainly based on statistical reasoning, have never been combined with parameterized verification based on symbolic reasoning. Through finding auxiliary invariants, we link these two areas, and achieve an initial success. We hope that our work can shed lights on the further combination of parameterized verification and machine learning.
- We successfully apply our framework to several benchmarks. Especially, we have verified the *FLASH* protocol automatically.

Paper structure. The rest of this paper is organized as follows. We first review related works in Section II. In Section III, we introduce preliminaries used in this paper. Section IV provides an overview of our framework. In the following section (Section V), we give detailed implementation of the framework. Section VI shows the experiment result, followed by conclusions and future work in Section VII.

II. Related Work

There have been many studies in the field of parameterized verification [1], [7]–[11], [14], [15], [31]. The standard way of verifying a system or protocol is to enumerate its entire state space [32]. However, this method reaches its limits in both time and memory consumption [1]. Thus, some approaches have been proposed to be scalable for large systems, such as compositional [23] and abstraction model checking techniques [11]. McMillan *et al*. proposed the "parameter abstraction and guard strengthening" technique to maintain the desired properties in the abstract system [10]–[12]. Later, Talupur *et al*. accelerated this technique using high-quality invariants which are derived from message flows [24]. Chen *et al*. employed a meta-circular assume/guarantee technique to reduce the complexity of verifying finite instances of parameterized protocols [13], [33]. However, these methods require hand-crafted invariants, which is often error-prone especially when the protocol description is long and complicated [11].

In the last decade, an active research area is to design automatic methods to find good quality invariants. Arons *et al*. proposed the concept of "invisible invariants", which are computed in a finite system instance to aid inductive invariant checking [9]. Inspired by this idea and "parameter abstraction and guard strengthening", Lv *et al*. [7] used a small instance of a parameterized protocol as a "reference instance" to compute candidate invariants. These approaches attempt to automatically find invariants, yet the invisible invariants are "raw" boolean formulas, which are BDDs computed by TLV (a variant of the BDD-based SMV model checker) and hard to understand. Conchon *et al*. developed a SMT-based model checker Cubicle together with a new algorithm BRAB. It computes over-approximations of backward reachable states that are checked to be unreachable in the parameterized system [15]. Their method is the first one that proves automatically safety properties of the *FLASH* protocol. However, its generalization was limited because it cannot be generalized to other general-purpose theorem provers such as Isabelle [34] or Coq [35]. Li *et al*. proposed a novel method to automatically generate auxiliary invariants from a small reference instance of protocols and construct a parameterized formal proof in the theorem prover Isabelle [30]. The form of auxiliary invariants they found is more straightforward and understandable when compared with all the previous works.

As a summary, we can see that the trend in paramterized verification, in particularly for invariant learning, is to switch from manual construction to automatic detection. As the automatic methods being developed, the form of auxiliary invariants is required to be more readable and understandable in order to make contributions not only in formal verification, but also in the fields of the protocol design and further research.

III. PRELIMINARIES

In this section, we present how to specify parameterized protocols using one example, and briefly introduce the CMP method for parameterized verification and the Apriori algorithm for association rule learning. In the end, we describe the idea of symmetry reduction in model checking.

A. Parameterized protocols

Let $P(N)$ be a *parameterized protocol*. If the parameter N is assigned an exact value m, $P(m)$ becomes an instance of $P(N)$. A complete protocol mainly consists of three parts: type declarations and initialization, transition rules and desired safety properties (i.e., in the form of invariants). To better introduce these concepts, we take the *Mutual Exclusion* protocol[1] written in the model checker Murphi's language as an example. In the rest of the paper, this protocol will serve as a running example. The protocol describes a parameterized system where N nodes share a resource and follow the protocol to ensure its mutual exclusive access. Each node maintains a record to mark its state and store the cached data. The node state can be I (idle), T (try to enter critical region), C (enter the Critical region) and E (exit the critical region). Each node acts under the guidance of transition rules in the protocol. The mutual exclusive access to the shared resource is guaranteed by a semaphore x, which acts as a lock, ensuring only one node can occupy the resource. The resource, in this case, is some data in the memory ($memDATA$). Besides, an auxiliary data ($auxDATA$) is used to keep a copy of the latest data. Now, we elaborate three main parts of this protocol.

```
-- Configuration parameters --
Const
   NODENUM : 2;
   DATANUMS: 2;

-- Type declarations --
Type
   NODE: 1..NODENUM;
   DATA: 1..DATANUMS;
   state: enum
         {I,T,C,E};
   status: record
      st: state;
      data: DATA;
   end;

-- State variables --
Var
   n: array [NODE] of state;
   x: boolean;
   auxDATA: DATA;
   memDATA: DATA;

ruleset d: DATA do
startstate
   for i: NODE do
      n[i]:= I;
      n[i].data:=d;
   endfor;
   x:= true;
   auxDATA:= d;
   memDATA:=d;
endstartstate;
endruleset;
```

The above first part defines the variables that occur in the protocol and their initial assignments. We can see that in this example, the parameter is initially set to be 2, which means there are 2 nodes in this instance. Variable n is an array-based variable, which denotes 2 nodes and each of them contains a $state$ type. This kind of variables are local variables, compared with global variables such as x, $auxDATA$ and $memDATA$.

[1]To better explain our framework, we add data controls to the naive mutual exclusion protocol [36].

```
ruleset i: NODE do
rule "rule_name"
  guard part   -- conjunction of predicates
==>
  action part   -- a set of statements
endrule;
endruleset;
```

Transition rulesets and rules are crucial in protocol specification. A transition rule mainly includes two parts: **guard** and **action**. If the predicates in the guard are satisfied, the statements in the action part can be executed. We can see that the guard part is conjunction of predicates (e.g., see rule "Crit" below), while the action part consists of a set of assignments. There are five transition rules in the ruleset of the *Mutual Exclusion* protocol.

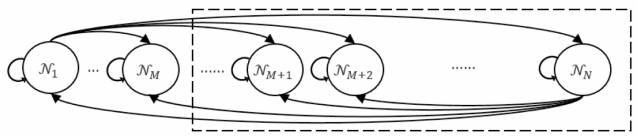
```
ruleset i:NODE do
  rule "Try"
    n[i].st = I ==>
    n[i].st:= T;
  endrule;

  rule "Crit"
    n[i].st = T & x = true
    ==>
    n[i].st:= C;
    n[i].data:= memDATA;
    x:= false;
  endrule;

  rule "Exit"
    n[i].st = C ==>
    n[i].st:= E;
  endrule;

  rule "Idle"
    n[i].st = E ==>
    n[i].st:= I;
    x:= true;
    memDATA:= n[i].data;
  endrule;

  rule "Store"
    n[i].st = C ==>
    auxDATA:= data;
    n[i].data:= data;
  endrule;
  endruleset;
```

In the third part, desired properties such as safety properties can be stated. Once they are not satisfied by the protocol, a counterexample and its error trace will be generated by a model checker (i.e., Murphi in the current paper). It means the design of protocol needs to be adjusted to prevent such a situation from happening. The safety properties of the *Mutual Exclusion* protocol are:

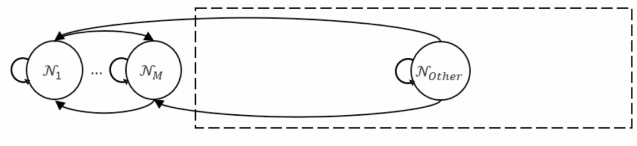
```
invariant "CntrlProp"
  forall i:NODE do
  forall j:NODE do
   i!=j->
   (n[i].st=C->n[j].st!=C)
  endfor; endfor;

invariant "DataProp"
  forall i:NODE do
   n[i].st=C->
   n[i].data=auxDATA
  endfor;
```

We can see the above description that there are two properties in this protocol. The first one states that it is not allowed for any two nodes to enter the critical region at the same time. While the second requires once a node is in the critical region, the data needs to be synchronized to $auxDATA$ as well. This is because $auxDATA$ is used to record the latest data.

Given the above *Mutual Exclusion* protocol, the model checker Murphi will enumerate the entire state space explicitly until no new reachable state can be explored or the properties fail to hold on the protocol. Here, the term *reachable state* is not the same with node's state we defined in the first part, it refers to one possible assignment for all the variables. In this paper, for convenience, we will simply refer to the "reachable state" as "state". Relatively, a set of all possible reachable states is regarded as reachable state set (abbrv. $RS(P)$).



(a) The original protocol P



(b) The abstracted protocol AP

Fig. 1. **Illustration of the CMP method.** For a parameterized protocol P with N nodes, the basic idea is to retain M nodes and abstract the rest nodes $(N_{M+1}, \ldots, N_N)$ as a single node N_{Other}. The abstracted protocol is denoted as AP.

Murphi starts from the initial states, which is prescribed by initialization in the first part. Then, the Murphi randomly finds a transition rule whose guard is satisfied and executes its action part. This "find-execute" process will be iterated until either a property is violated or no new reachable state can be found.

B. The CMP method

The "Parameterized abstraction and guard strengthening" method, also known as CMP [10], [11] has been well studied and applied to verify industrial protocols. Its main idea is rather clear and easy to understand, as shown in Figure 1. For a parameterized protocol P with N nodes, CMP retains M nodes (e.g., $M = 2$), and abstracts the remaining nodes $(N_{M+1}, \ldots, N_N)$ as a single node N_{Other}. In general, the abstracted protocol AP may fail to satisfy the desired properties because it is too 'permissive' [11], so Murphi will generate a counterexample and the error trace. The researchers need to analyze the error trace and construct an auxiliary invariant to restrict the behavior of N_{Other}. This process stops when the AP eventually satisfies the properties as well as all the constructed auxiliary invariants.

The detailed implementation of the CMP method can be divided into two steps: parameterized abstraction and guard strengthening. We use the *Mutual Exclusion* protocol to explain how the method works.

1) Parameterized abstraction: This step abstracts the original protocol P to an abstracted protocol AP manually. To be more specific, for a transition rule, CMP will firstly instantiate the parameterized indexes (e.g., i and j) as an abstracted node $Other$, which is an unobservable node that simulates the behavior of abstracted nodes $N_{M+1}, \ldots, N_N$. Then, it abstracts the local variables by eliminating them from predicates in guard and statements in actions. After that, the abstracted rules together with P consist of a new abstracted protocol AP. Take the rule "Crit" as an example. First, the parameter i is instantiated to $Other$, then the predicates and statements relating to $Other$ will be removed. So, after abstraction, only the global variable x will remain. If after abstraction, there is no statement left in action part of the rule, the transition rule will be removed as well. Therefore, after this step, the transition rules in the *Mutual Exclusion* protocol will be abstracted to:

```
rule "ABS_Crit"
    x = true ==>
    x:= false;
endrule;

rule "ABS_Idle"
    true ==> x:= true;
endrule;
```

```
rule "ABS_Store"
    true ==>
    x:= false;
    auxDATA:= data;
endrule;
```

The guard of rules "ABS_Idle" and "ABS_Store" is set to be true because there is no predicate left after abstraction. After this step, we can see that the abstracted rules are too permissive. For example, the global variable x, which serves as a critical lock, can be set to available without any condition (see rule "ABS_Idle"). Thus, the guards of these rules need to be strengthened.

2) Guard strengthening: In this step, the "counterexample-guided" strengthening paradigm is executed iteratively [11]. It depends on the model checker Murphi to check the protocol instance, and once a counterexample occurs, researchers will analyze its error trace and come up with an auxiliary invariant to prevent it from happening.

C. Association rule learning

Association rules learning is a classical data mining algorithm [37]. It was designed to mine correlations between items in large databases. Let $D = \{t_1, t_2, \ldots, t_n\}$ be a set of transactions called the database and $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m items called itemset, each $t_k \in D$ contains a subset of the items in I. An association rule is composed by two different itemsets X and Y, where X is called antecedent or left-hand-side (LHS) and Y consequent or right-hand-side (RHS). It is defined as an implication of the form $X \rightarrow Y$ where $X, Y \in I$. In this paper, we follow the original definition in [37] to restrict Y only containing one item.

In order to select association rules from the set of all possible rules, constraints on various measures of significance and frequency are used. The best-known constraints are minimum thresholds on support and confidence. *Support* is an indication of how frequently the itemset appears in the dataset. The formula is defined as follow:

$$S(X) = P(X) = \mid \{t \in D \mid X \subseteq t\} \mid / \mid T \mid \qquad (1)$$

where X denotes an itemset, $|t|$ denotes the number of transaction that contains X, $|T|$ represents the number of transaction in the database. *Confidence* is an indication of how often the association rule has been found to be true in the database. It is defined as:

$$C(X \rightarrow Y) = S(X, Y)/S(X). \qquad (2)$$

Mining association rules are usually required to satisfy both a user-specified minimum support and a user-specified minimum confidence. Given the minimum support and minimum confidence, there are usually two steps to learn the expected

association rules: (1) mining frequent itemsets whose *Support* is greater than the minimum support, and (2) mining association rules whose *Confidence* is greater than the minimum confidence. In this paper, we adopt the Apriori algorithm [37] to perform this learning process.

D. Symmetry reduction

In automatic verification of finite-state systems, the general process is to enumerate all the reachable states of the system. Yet, the major problem of this method is the state explosion problem, i.e., the size of the state space grows rapidly with the size of the system [38]. To address this problem, symmetry reduction techniques have been developed [38]. The basic idea is simple but effective: it utilizes the equivalence relation between states to explore only one state per equivalence class. Take the *Mutual Exclusion* protocol as an example. For two processes A and B, the state where A is in the critical section and B is waiting is equivalent to the state in which B is in the critical section and A is waiting [38].

With the help of symmetry reduction, often the states space can be reduced by over 90%, and the computational time can be reduced by more than 40% according to statistical results provided by [38]. Below we list the comparison of the numbers of reachable states before (#states-b) and after (#states-a) applying symmetry reduction in Table I. It is rather clear that the usefulness of symmetry reduction technique increases with the increasing size of original reachable sets. Thus, we adopt this technique in our framework as well.

TABLE I
COMPARISON ON SCALES OF REACHABLE STATES SETS

Protocol	#instance	#states-b	#states-a
MOESI	2	23	6
MESI	2	8	5
MutualEx	2	12	7
MutData	2	88	23
GERMAN	2	43,422	852
FLASH	3	764,100,000	1,350,226

IV. AN OVERVIEW OF OUR FRAMEWORK

In this section, we give an overview of our proposed framework for automatic parameterized verification. There are two main phases in our framework, as shown in Figure 2.

Part 1. Learning invariants. This phase aims at learning auxiliary invariants by association rule learning. There are three main steps in this part, including data collection and preprocessing, learning association rule and selecting auxiliary invariants. At first, an instance of protocol is given to the model checker to generate the reachable state set. If the instance fails when verifying the safety properties, then the overall verification stops. Otherwise, the reachable state set is converted to a dataset which is suitable for association rule learning. This step is crucial and innovative (Section V-A). Because of it, the learning algorithm can learn sufficiently many invariants of high quality. Next, the association rule learning is applied to this dataset, and a set of association rules can be learned (Section V-B). Note that these association rules are not necessarily invariants of the protocol, so a selection step is needed with the help of a model checker (Section V-C).

Part 2. Strengthening and abstraction. Unlike the CMP method, in this part, the guard strengthening (Section V-D) is performed before parameterized abstraction (Section V-E). Besides, different from the origianl "counterexample-guided" paradigm, we propose an "invariant-guided" strengthening paradigm, which avoids the laborious human effort and repeated strengthening process. After strengthening and abstraction, the resulting abstracted protocol is subjected to the model checker (Section V-F). Note that the invariants which are used to strengthen the guard of rules also need to be verified as true invariants. If the protocol passes the verification step, then the overall parameterized verification has finished. Otherwise, the resulting protocol is still too "permissive", which means more auxiliary invariants are needed, so the next round of invariant learning will start (moving to step 1 of the whole framework).

V. IMPLEMENTATION DETAILS

In this section, we present the technical details of our framework. We use the *Mutual Exclusion* protocol as a running example to explain its realization.

A. Data Collection and Preprocessing

We collect the reachable state set of the *Mutual Exclusion* protocol (with *NODE_NUM* = 2, *DATA_NUM* = 2) from the model checker Murphi and list a few of its states below.

State 1:	State 2:	State 3:	State 4:
n[1].st:I	n[1].st:I	n[1].st:T	n[1].st:I
n[1].data:1	n[1].data:2	n[1].data:1	n[1].data:1
n[2].st:I	n[2].st:I	n[2].st:I	n[2].st:T
n[2].data:1	n[2].data:2	n[2].data:1	n[2].data:1
x:true	x:true	x:true	x:true
auxDATA:1	auxDATA:2	auxDATA:1	auxDATA:1
memDATA:1	memDATA:2	memDATA:1	memDATA:1

Recall the concepts of "transaction" and "itemset" in Section III-C, for the reachable state set, if each state represents a transaction, then what about itemset? Intuitively, every pair of variable-value (e.g., *n[1].st* = I) can be regarded as an item, thus all possible pairs make up the itemset. However, this method may ignore some crucial relations. For example, for the variable *auxDATA*, its exact value does not really matter, but its relation with other variables like *n[1].data* is important. In other words, for some variables, the significance of the relation with other variables goes beyond its exact assignment. Given this intuition, we need to take pairs of variables into account. Nevertheless, assume there are $n = 15$ variable, then the possible combinations will be $n(n-1)/2 = 105$, while not all of them are really useful. Thus, we need to figure out how to construct a sufficient itemset effectively.

After a few attempts, we find that the predicates in the guards of protocols rules are ideal items. They are atomic, and given a reachable state they are decidable. Furthermore, we extend this idea to predicates in the safety properties as well. The entire searching process works as follows:

$$\mathsf{WP}(A, f) \equiv f[v_1 \mapsto e_1, \ldots, v_n \mapsto e_n]$$

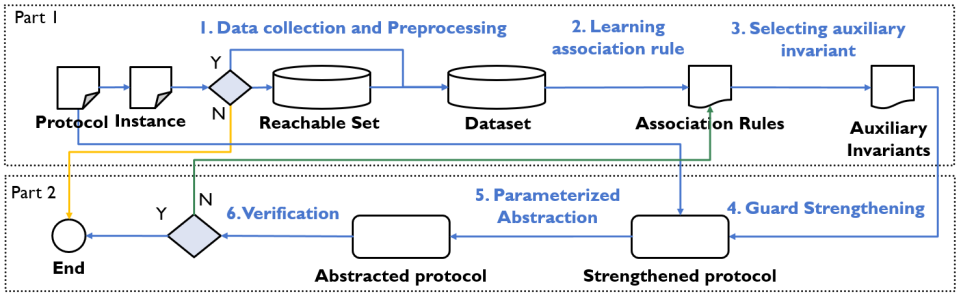


Fig. 2. An illustration of our framework.

where WP is the weakest precondition operator, $\mathsf{WP}(A, f)$ substitutes each occurrence of a variable v_ℓ by the corresponding term e_ℓ in formula f. Initially, our algorithm computes

$$A_0 = \bigcup\{\mathsf{atoms}(f),\ f \in (\{guard(r) \mid r \in R\} \cup invs)\},$$

where $\mathsf{atoms}(f)$ returns atomic predicate of f, and R and $invs$ are the set of protocol rules and safety properties under verification; then it continues to compute

$$A_{i+1} = \{\mathsf{WP}(action(r),\ f),\ f \in A_i,\ r \in R\},$$

until $A_{i+1} = A_i$. Finally, the returned A_n (n being the final step) is the set of atomic predicates and can be used as items.

Then, the reachable state set can be transformed to a dataset where the values in the first line are the atomic predicates, and the following values in each row means the presence of the predicate in the corresponding state (row).

B. Learning association rules

We adopt the Apriori algorithm to learn association rules from the constructed dataset. We add a constrain K to the size of frequent itemset, which sets an upper bound. The algorithm is presented as below, which is almost the same as the original Apriori algorithm [37], [39].

Algorithm 1: $K - Apriori(D, \theta, K)$

Input: a dataset D, minimum support θ, maximum frequent itemset size K
Output: a set of frequent itemsets
- C_k: Candidate itemset of size k
- L_k: frequent itemset of size k
- L_1: {itemset}

1: **for** $k = 1; k \leq K; k++$ **do**
2: C_{k+1} = candidates generated from L_k;
3: **for** each $t \in D$ **do**
4: increment the count of all candidates in C_{k+1} that are contained in t
5: **end for**
6: $L_{k+1} \leftarrow$ candidates in C_{k+1} that satisfy θ
7: **end for**
8: **return** $\bigcup_k L_k$;

After performing the Apriori algorithm, a set of association rules can be learned for the *Mutual Exclusion* protocol, part of the rules are listed below:

```
asR1:n[1].st!= I & n[1].st = C -> x = false
asR2:n[1].st!= T & n[1].st = E -> x = false
asR3:n[1].st = T & n[2].st = T -> x = true
asR4:n[1].st = E & n[2].st = I -> n[1].data=auxDATA
asR5:n[1].st!= T & n[2].st = C -> x = false
asR6:n[1].st!= C & n[2].st = E -> n[2].data=auxDATA
```

In our framework, we need to set the size of frequent itemset for the sake of feasibility and reducing time consumption. Empirically, we have carried out a series of experiments with different sizes of K starting from 2, and the experiments show that the association rules that learned from 3-frequent itemset are always sufficient. Besides, when increasing the size of frequent itemset, it takes too much time to learn the association rules, this leads us to set K as small as possible (3 in the current paper). Second, we need to figure out what is the best minimum support for protocols. It is generally practical to set minimum support as small as possible, i.e., 0.0, so that all possible frequent itemset will be considered. Experiments show that the minimum support rate does not largely affect the verification results for small protocols, while its effect on larger protocols cannot be ignored – with larger rates the protocols might fail to be strengthened and verified with the learned association rules. For minimum confidence, it needs to ensure the certainty of association rules, so we set it as 1.0 to guarantee the highest certainty of the association rules.

C. Selecting auxiliary invariants

In the previous step, the minimum confidence of association rules is set to be 1.0, this does not indicate that they are invariants of the protocol. There are two reasons. First, the dataset from which association rules are learned is transformed from the reachable state set of a small protocol instance, so the rules may not be true for larger instances. For example, see "asR3" in Section V-B, this association rule has 100% confidence as there are only 2 nodes in the instance. However, in a 3-node instance, this rule will be violated if *n[3].st* = E. In other words, this association rule is not a true invariant. Second, for large protocols like *FLASH* whose state space is huge, we adopt symmetry reduction technique to scale down the searching space. This technique saves verification time and space, but at the same time the pruned states may draw some association rules that are incorrect in the reachable state set of the instance. This type of association rules need be removed.

To remove above-mentioned association rules, we use a model checker to filter them. At first, we treat all of the

learned associate rules as invariants of the original protocol instance and submit them to the model checker. Once some of them are violated, they will be removed. Then, we increase the size of the protocol instance, and repeat the previous step. The iteration stops until no association rules can be removed or the size of the reachable state set reaches a limit, which is set to be 10G. The remaining association rules are considered as auxiliary invariants. Note that these auxiliary invariants are still not guaranteed to be invariants, and they need to be verified in the final abstracted protocol. Note some auxiliary invariants for the *Mutual Exclusion* protocol are show in Figure 3.

D. Guard strengthening

Unlike the counterexample-guided' strengthening in the original CMP method, we propose an invariant-guided' paradigm, which strengthens the guards of protocol rules in one step. In the former paradigm, from the original protocol instance the abstraction can make the abstracted protocol too "permissive" resulting in counterexamples. Researchers need to analyze the error traces of a counterexample and add constrains to the guard of protocol rules in order to avoid such a counterexample. After several attempts of the above operation, the reachable state set will be constrained in a safe region. On the contrary, our paradigm is more direct. It expands the reachable set to a safe boundary, avoiding repeated attempts. Besides, our paradigm allows us to automatically strengthen the abstracted protocol instead of relying on experts' analysis.

Recall that the guard of a protocol rule is a conjunction of predicates, and the auxiliary invariants are in form of implication, where the LHS is a set of predicates and the RHS is an atomic predicate. At first, for each rule, we select those auxiliary invariants whose LHS can be implied by the guards. Then, we add the RHS into the guard. We repeat the above two steps until no more predicates can be added into the guard.

Take the transition rules "Idle" and "Store" of the *Mutual Exclusion* protocol as examples. As shown in Figures 3(a) and 3(b), the guard of rule "Idle" *n[i].st = E* could imply the LHS of three auxiliary invariants as shown at the bottom of Figure 3(a). Then we add the RHS of these invariants to the guard of this rule (as shown in Figure 3(b) in blue). Note that the RHS of the first two auxiliary invariants include the parameter j, so they need to be generalized to all parameters; while the last one include the same parameter i as the ruleset definition, so it can be added directly. Rule "Store" can be strengthened in the same way (see Figures 3(d) and 3(e)).

E. Parameterized abstraction

The purpose of this step is to remove the predicates and statements related to the parameters. If a rule contains more than one parameter, then they will be abstracted one by one. Again, take rule "Store" as an example. In Figure 3(e), the parameter regarding abstract object (i.e., NODE) is i, so predicates *n[i].st = C* and *n[i].data = auxDATA* in its guard will be abstracted away, as well as the statement *n[i].data :=*

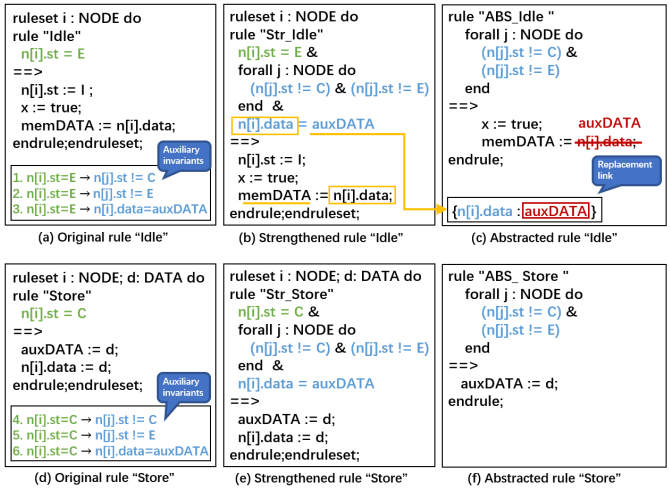


Fig. 3. Original, strengthened and abstracted protocol rules "Idle" and "Store".

d in action will also be removed, resulting in the abstracted rule "ABS_Store" in Figure 3(f).

However, if we directly abstract rule "Str_Idle" in Figure 3(b), the safety properties will be violated since the behavior of other nodes will be changed due to the abstraction. For example, assume we directly remove the predicates and statements regarding i in rule "Str_Idle", resulting in rule "ABS_Idle" (Figure 3(b) without the part *memDATA := n[i].data*). Now assume the system works according to the abstracted rule "ABS_Idle" and "ABS_Store" as well as the original transition rules, when $memData$ is 1 and the node $Other$ is in the critical region, it changes the $auxData$ from 1 to 2 according to rule "ABS_Store" then exits the critical region according to the rule "ABS_Idle". Then, another normal node enters the critical region and reads $memData$, which is 1. At this moment, the property "DataProp" will be violated because the node $Other$ has already changed $auxData$ to 2, while the changed data has not been synchronized to $memData$. Such mistakes introduced by direct abstraction needs to be prevented in order to make sure normal nodes hold the original properties.

We thus introduce a replacement link to deal with the statements in the action part. Links are built between a local variable and global variable only if they are the RHS of auxiliary invariants used in guard strengthening and linked by the equal sign. For instance, for rule "Str_Idle" in Figure 3(b), among the RHS of three used auxiliary invariants, only *n[i].data= auxDATA* satisfies the requirement, so the replacement link could be built as shown at the bottom of Figure 3(c). Next, we need to check the right side of each statement if it contains local variables and a replacement link exists for the local variables, and then we replace them with the corresponding global variables. For example, as shown in Figure 3(b), the right side of the last statement is *n[i].data*, which is a local variable and appears in the replacement link, so in Figure 3(c), *n[i].data* is replaced by *auxDATA* according to the replacement link. Finally, parameterized abstraction is carried out on all the protocol rules. Note that if there are more

TABLE II
EXPERIMENTAL RESULTS

Protocol	Instance	# of RS	# of assoRules	# of auxInvs	# of used auxInvs	# of RS(abs)	Time (min)	Result
MOESI	2	6	736	10	5	7	0.03	✓
MESI	2	5	144	8	5	5	0.03	✓
MutualEx	2	12	656	12	3	18	0.12	✓
MutData	2	88	540	12	6	768	0.12	✓
GERMAN	2	852	21,202	224	37	1,314	0.83	✓
FLASH	3	1,350,226	234,578	1,638	331	26,962,920	384.89	✓

than two parameters (e.g., i and j) in a rule, then there will be three situations: 1) i : $Other$, j: normal node; 2) i: normal node, j : $Other$; and 3) i: $Other$, j : $Other$. The first two situations can be treated symmetrically. When parameter i is instantiated by node $Other$, then the predicates and statements on it will be abstracted as in case 2), while those on j are retained. For the last situation, the predicates and statements on both i and j need to be abstracted.

F. Verification

In this final step, the abstracted protocol needs to be model checked. The size of instance remains the same as the one from which the reachable state set is generated. The used auxiliary invariants need to be verified as invariants in this step as well.

VI. EVALUATION

We have implemented our framework [**?**] and applied it to several typical parameterized cache coherence protocols. Among these protocols, *Mutual Exclusion, MOESI, MESI* are small-scale protocols, while *German* and *FLASH* protocols are relatively more complicated and large-scale protocols.[2] *German* is a cache coherence protocol devised by Steven German in 2000 [40], which is now a common example in research papers on parameterized verification [8], [11], [25], [41], [42], while *FLASH* is much more complex and realistic than *German*. It has more than 67 million states when instantiated with four processes. As Chou *et al.* pointed out in [11], "if the method works on *FLASH* protocol, then there is a good chance that it will also work on many real-world cache coherence protocols", this marks *FLASH* as an industrial benchmark for parameterized verification.

Our experimental results are summarized in Table II. The configuration indicates the size of benchmarks. We can see from the table that when the parameter is set to be 2, the first four protocols are quite small, with less than 100 reachable states. *German* is a bit larger, with more than 800 states. While for 3-node *FLASH*, the number of reachable states is more than 1,500 times that of *German*. How the number of nodes is set is mainly due to the architecture of the protocols. If the protocol only contains homogeneous nodes, then the parameter is set to be 2, mainly thanks to the symmetry in the protocols. Sometimes, a protocol also contains a heterogeneous node (e.g., *FLASH* protocol contains a 'Home' node), then the parameter is set to be 3 – there are 2 normal nodes and one 'Home' node.

[2]The versions of *German* and *FLASH* protocols we use are the same as those used in Chou *et al.*'s work [11].

The statistics includes four types of data, which are the number of learned association rules, the number of auxiliary invariants, the number of used invariants, and the size of the reachable state set of the corresponding abstracted protocols. We can see that even from the reachable state set of small-scale protocols, a considerable amount of association rules can be learned. Yet after model checking these rules, only small proportion of them (approximately 1%) will remain, regarded as auxiliary invariants. Then, less than half of them can next be used to actually strengthen the protocol rules. After strengthening and abstraction, the abstracted systems can be converged within a reasonable and acceptable scale. Regarding the total amount of time consumption, we can see that the small-scale protocols can be verified with few seconds, followed by *German* taking less than a minute. Whereas, *FLASH* takes about 12 hours to complete the verification. Overall most time is consumed by selecting invariants and verifying abstracted systems in our framework, except for *FLASH*, where it takes more than half of the total amount of time to learn association rules – mainly due to its enormous reachable state set. Although the entire verification of *FLASH* takes a considerable amount of time, we believe that this can be accelerated by allocating more processors in parallel.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a learning-based framework to conduct parameterized verification automatically with the support of an association rule learning algorithm. The auxiliary invariants we learned are in a rather simple but powerful form (i.e., implications). Besides, we proposed a new invariant-guided strengthening paradigm, which is more effective than the counterexample-guided paradigm in the literature. Our framework has been successfully applied to several parameterized verification benchmarks including the FLASH protocol. The novelty of our work lies in the two aspects: (1) the combination of learning algorithms and parameterized verification; and (2) the invariant-guided strengthening paradigm, which is an effective strengthening strategy and provides a new perspective to the original CMP method.

In future, we plan to perform more case studies to further evaluate our framework. We also want to develop techniques or heuristics to improve the quality of learned association rules within our framework. Next, we want to extend the ability of our framework to prove not only safety properties but also liveness properties. Furthermore, we will also investigate other learning algorithms, and explore more possibilities in combining them with parameterized verification.

Acknowledgments Yongjian Li is supported by grant 61672503 from National Natural Science Foundation and grant 2017YFB0801900 from the National Key Research and Development Program in China.

REFERENCES

[1] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi, "Invariants for finite instances and beyond," in *Formal Methods in Computer-Aided Design (FMCAD'13)*. IEEE CS, 2013, pp. 61–68.

[2] P. A. Abdulla, A. P. Sistla, and M. Talupur, "Model checking parameterized systems," in *Handbook of Model Checking*. Springer, 2018, pp. 685–725.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[4] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proc. 12th ACM Symposium on Principles of Programming Languages (POPL'85)*. ACM Press, 1985, pp. 97–107.

[5] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Proc. 5th International Symposium on Programming*, ser. Lecture Notes in Computer Science, vol. 137. Springer, 1982, pp. 337–351.

[6] K. R. Apt and D. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Information Processing Letters*, vol. 22, no. 6, pp. 307–309, 1986.

[7] Y. Lv, H. Lin, and H. Pan, "Computing invariants for parameter abstraction," in *Proc. 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'07)*. IEEE CS, 2007, pp. 29–38.

[8] A. Pnueli, S. Ruah, and L. D. Zuck, "Automatic deductive verification with invisible invariants," in *Proc. 7th International Conference Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'01)*, ser. Lecture Notes in Computer Science, vol. 2031. Springer, 2001, pp. 82–97.

[9] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck, "Parameterized verification with automatically computed inductive assertions," in *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, ser. Lecture Notes in Computer Science, vol. 2102. Springer, 2001, pp. 221–234.

[10] K. L. McMillan, "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)*, ser. Lecture Notes in Computer Science, vol. 2144. Springer, 2001, pp. 179–195.

[11] C. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proc. 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, ser. Lecture Notes in Computer Science, vol. 3312. Springer, 2004, pp. 382–398.

[12] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," *Proc. 4th Workshop on Automated Verification of Infinite State Systems (AVIS'05)*, 2005.

[13] X. Chen and G. Gopalakrishnan, "A general compositional approach to verifying hierarchical cache coherence protocols," Technical Report, School of Computing, University of Utah, Tech. Rep., 2006.

[14] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, ser. Lecture Notes in Computer Science, vol. 2031. Springer, 2001, pp. 113–127.

[15] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi, "Cubicle: A parallel smt-based model checker for parameterized systems - tool paper," in *Proc. 24th International Conference on Computer Aided Verification (CAV'12)*, ser. Lecture Notes in Computer Science, vol. 7358. Springer, 2012, pp. 718–724.

[16] K. R. Apt and D. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Information Processing Letters*, vol. 22, no. 6, pp. 307–309, 1986.

[17] M. C. Browne, E. M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," *Information and Computation*, vol. 81, no. 1, pp. 13–31, 1989.

[18] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *Journal of the ACM*, vol. 39, no. 3, pp. 675–735, 1992.

[19] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy, "The stanford FLASH multiprocessor," in *Proc. International Symposia on Computer Architecture (ISCA'98)*. ACM Press, 1998, pp. 485–496.

[20] P. A. Abdulla, F. Haziza, and L. Holík, "All for the price of few," in *Proc. 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, ser. Lecture Notes in Computer Science, vol. 7737. Springer, 2013, pp. 476–495.

[21] A. Kaiser, D. Kroening, and T. Wahl, "Dynamic cutoff detection in parameterized concurrent programs," in *Proc. 22nd International Conference on Computer Aided Verification (CAV'10)*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 645–659.

[22] Y. Li, K. Duan, D. N. Jansen, J. Pang, L. Zhang, Y. Lv, and S. Cai, "An automatic proving approach to parameterized verification," *ACM Transactions on Computational Logic*, vol. 19, no. 4, pp. 27:1–27:25, 2018.

[23] E. M. Clarke, D. E. Long, and K. L. McMillan, "Compositional model checking," in *Proc. 4th Annual Symposium on Logic in Computer Science (LICS'89)*. IEEE CS, 1989, pp. 353–362.

[24] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proc. Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE CS, 2008, pp. 1–8.

[25] S. K. Lahiri and R. E. Bryant, "Constructing quantified invariants via predicate abstraction," in *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, ser. Lecture Notes in Computer Science, vol. 2937. Springer, 2004, pp. 267–281.

[26] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parameterized verification," in *Proc. 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, ser. Lecture Notes in Computer Science, vol. 3855. Springer, 2006, pp. 126–141.

[27] A. Cohen and K. S. Namjoshi, "Local proofs for global safety properties," *Formal Methods in System Design*, vol. 34, no. 2, pp. 104–125, 2009.

[28] O. Grinchtein, M. Leucker, and N. Piterman, "Inferring network invariants automatically," in *Proc. 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, ser. Lecture Notes in Computer Science, vol. 4130. Springer, 2006, pp. 483–497.

[29] K. L. McMillan, "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)*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 413–427.

[30] Y. Li, K. Duan, Y. Lv, J. Pang, and S. Cai, "A novel approach to parameterized verification of cache coherence protocols," in *Proc. 34th IEEE International Conference on Computer Design (ICCD'16)*. IEEE CS, 2016, pp. 560–567.

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

[32] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[33] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou, "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 CS, 2006, pp. 81–88.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.

[35] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[36] S. Conchon, D. Declerck, and F. Zaïdi, "Cubicle- *W*: Parameterized model checking on weak memory," in *Proc. 9th International Joint Conference on Automated Reasoning (IJCAR'18)*, ser. Lecture Notes in Computer Science, vol. 10900. Springer, 2018, pp. 152–160.

[37] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM International*

Conference on Special Interest Group on Management of Data (SIGMOD'93), vol. 22, no. 2. ACM Press, 1993, pp. 207–216.

[38] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 41–75, 1996.

[39] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th International Conference on Very Large Data Bases (VLDB'94)*, vol. 1215. Morgan Kaufmann, 1994, pp. 487–499.

[40] S. M. German, "Personal communications," 2000.

[41] K. Baukus, Y. Lakhnech, and K. Stahl, "Parameterized verification of a cache coherence protocol: Safety and liveness," in *Proc. 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, ser. Lecture Notes in Computer Science, vol. 2294. Springer, 2002, pp. 317–330.

[42] E. A. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *Proc. 12th Advanced Research Working Conference Correct Hardware Design and Verification Methods (CHARME'03)*, ser. Lecture Notes in Computer Science, vol. 2860. Springer, 2003, pp. 247–262.