Learning Based Parameterized Verification

Abstract. Parameterized verification is a complicated and undecidable problem. A widely-used method is "parameterized abstraction and guard strengthening", proposed by **C**hou, **M**annava and **P**ark (abbrv. **CMP**). However, this method is carried out manually, which may allow errors to creep in. The hardest part of automating CMP is finding invariants by computer, which highly relies on researchers' analysis in the previous works. To solve this problem, we propose an intuitive framework, Learning-Based CMP (abbrv. **LB-CMP**), which can automatically learn invariants from reachable set of protocols. By setting constrains to association rule learning, a set of straightforward invariants can be obtained and then used to strengthen the abstract protocols iteratively. It is noteworthy that our framework is not only an automation of CMP, but a new perspective to verify parameterized protocols. Our framework has been successfully applied to several typical protocols, especially FLASH, an industrial-scale protocol. To our best knowledge, we are the first work to apply the machine learning technique to parameterized verification. Our successfully attempt points out that it is possible and promising to apply machine learning, or more specific, data mining techniques to aid invariants-based formal verification. We hope our work may shed some light on the further research direction for this area.

1 Introduction

Parameterized concurrent systems exist in many practical areas, such as cache coherence, security, communication and network protocols [1]. Verifying such systems has attracted considerable interests from model checking and theorem proving communities due to its practical importance [2]. The challenge of parameterized verification is that one needs to check the correctness with arbitrary sizes of instances, which is proved to be an undecidable problem [3].

To address this problem, many approaches [4,5,6,7,8,9,10,11,12,13,14] have been proposed. Among them, "parameterized abstraction and guard strengthening" method, also known as **CMP** [7,8,9], has been widely applied to verify large-scaled and industrial-strength protocols like Intel's Chipset and FLASH protocols [15]. Although the main idea of CMP method is simple and effective (illustrated in Figure. 1), the drawback is apparently: it is carried out by hand, which may let in errors, especially when the protocol description is long [8]. To automate CMP, the crux lies in how to generate sufficient invariants automatically, because in the previous works, this step highly relies on researchers' understanding of protocols and analysis of counter examples given by the model checker. To solve this problem, we propose LB-CMP(Learning-Based CMP), which can automatically learn auxiliary invariants, abstract parameter

and strengthen abstracted protocols in a unified framework. Instead of understanding protocol description and analyzing the counter-example, we make use of the reachable sets and learn association rules from it. By setting constrains to the learning process, the straightforward and easy-to-understand association rules can be learned. After filtering local truths and spurious invariants from these association rules, the invariants can be obtained. Then, we automate the parameter abstraction and guard strengthening steps, which is tedious for manual work. Noticeably, CMP generates invariants by analyzing counter-example, which is a passive method carried out only after counter-example happens, while our work provides a more active perspective, which takes action before error happens. The results of LB-CMP can be found in the link [1].

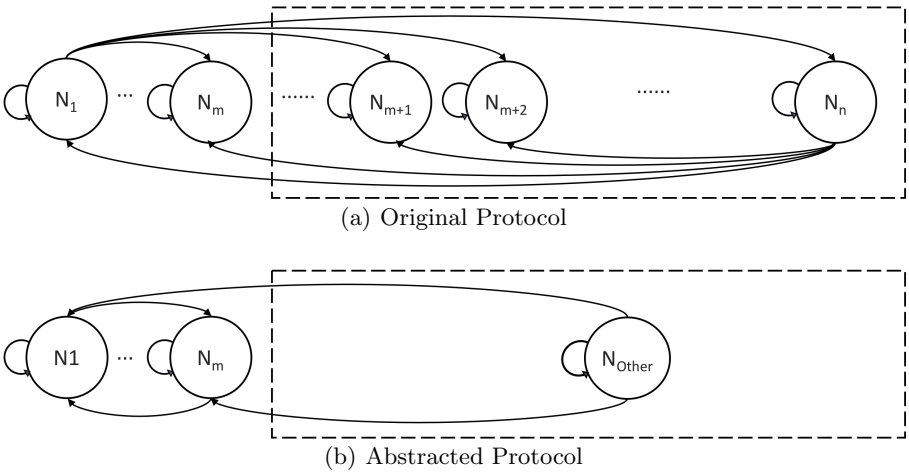

Fig. 1: **Illustration of CMP Method.** The figure illustrates the main idea of CMP. For a parameterized system $\mathcal{P}$ with n nodes, each circle denotes a node, indexed by 1, 2, ..., n. CMP retains a small number (assume m) of nodes, and abstracts the remaining nodes $(N_{m+1}, ..., N_n)$ as a single node N_{Other}. After refinement, the behavior of $\mathcal{P}$ can be simulated by $\mathcal{AP}$ and the desired properties can be verified in $\mathcal{AP}$.

Contributions

- We build the bridge from learning algorithms to parameterized verification. By applying association rule learning, we derive straightforward and easy-to-understand invariants to automatically strengthen the abstracted protocol.
- We formalize and automate parameterized abstraction and guard strengthening procedures in a new perspective. We strengthen guards of rules before counter-example happens, which avoids laborious analysis of counter-example and protocol description.

[1] https://github.com/ArabelaTso/Learning-Based-ParaVerifer

– We apply our framework to verify several typical benchmarks successfully. Especially, we verify the FLASH protocol, which is in industrial scale. To our best knowledge, we are the first work to automatically verify Flash with the method of parameterized abstraction and guard strengthen.

Related Work There have been a lot of studies in the field of parameterized verification [10,16,6,5,12,7,8,13,2,14,4]. These methods fall into two paradigms. Firstly, **manual approaches.** These methods require manual effort to formulate auxiliary invariants, with or without the aid of computer. Mcmillan *et. al* proposed "parameter abstraction and guard strengthening" technique to maintain the desired properties in the abstract system [7,8,9]. Later, Talupur *et. al* accelerated this technique using high-quality invariants which are derived from message flows [15]. Chen *et. al* employed a metacircular assume/guarantee technique to reduce the complexity of verifying finite instances of protocols [17,11]. Secondly, **automatic approaches.** These methods try to generate invariants automatically. Some attempts such as invisible invariants [6], indexed predicates [18], interpolant-based invariant generation [19], and split invariants [20] have been proposed in order to deduce invariants with the aid of computer. Arons *et. al* proposed the concept "invisible invariants", which are computed in a finite system instance to aid inductive invariant checking [6]. Conchon *et. al* came up with a BRAB algorithm which 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 [14]. Li *et. al* automatically generated auxiliary invariants from a small reference instance of protocols, and constructed a parameterized formal proof in the theorem prover [1].

2 Preliminaries

2.1 The Problem of Parameterized Verification

Let a **parameterized communication protocol** be $\mathcal{P}$. $\mathcal{P}$ is formalized as a triple (V, I, R), where V is a set of variables, $I(V)$ is a set of initial predicates over V, and R is a set of guarded transition rules. Variables in a protocol fall into two categories: local variables, usually denoted as elements in array, indexed with node identities, e.g., $arr[i]$, indicating the i-th element of the array arr; and global variables, refer to the non-indexed ones. Each guarded command $r \in R$ is in the form of $g \rhd A$, where the guard g is the guard predicate over V, and A is the transition action of the rule. We write $\mathsf{guard}(r) = g$, and $\mathsf{action}(r) = A$ for a rule r. Reachable states (abbrv. RS) of $\mathcal{P}$can be denoted as $\mathsf{RS}(\mathcal{P})$. A state s is in $\mathsf{RS}(\mathcal{P})$ if: there exists a formula $f \in I$ such that $s \models f$; or there exists a state s' and a guarded command $r \in R$ such that $s' \in \mathsf{RS}(\mathcal{P})$ and $s' \xrightarrow{r} s$. Assume ϕ is an invariant in $\mathcal{P}$. If $s \models \phi$ holds for all reachable states in $\mathsf{RS}(\mathcal{P})$, then we shall say ϕ is an invariant of $\mathcal{P}$, denoted as $\mathcal{P} \models \phi$

We now illustrate a simple example, *Mutual Exclustion* (written in Murphi language). As defined, there are 2 nodes in this system. Each process has four states, I, T, C, and E, transferring according to transition rules in the ruleset.

Starting from the start state, the state of each node is assigned to be I. Then Murphi model checker finds all rules whose guards are satisfied in the current state, and randomly chooses one to execute its action. This find-execute process will be iterated infinitely until a property is violated or no new state can be enumerated.

```
CONST
    NODE_NUM : 2;
TYPE
    state : enum{I, T, C, E};
    NODE: scalarset(NODE_NUM);
VAR
    n : array [NODE] of state;
    x : boolean;
startstate
    for i: NODE do
        n[i] := I;
    endfor;
    x := true;
endstartstate;
```

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

rule "Crit"
    n[i] = T & x = true ==>
    n[i] := C; x := false;
endrule;
rule "Exit"
    n[i] = C ==>
    n[i] := E;
endrule;
```

```
rule "Idle"
    n[i] = E ==>
    n[i] := I;
    x := true;
endrule;
endruleset;

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

2.2 CMP Method

For a parameterized protocol $\mathcal{P}$ with n processes, CMP method retains a small number (usually two or three) of processes, and replaces the remaining processes with a single process which can be regarded as an abstraction of all the other nodes, illustrated as N_{Other} in Figure. 1. Then formulating a set of protocol invariants which constrain the behavior of N_{Other}, such that the desired properties can be hold in the abstracted protocol.

The workflow of CMP method can be roughly divided into two phases:

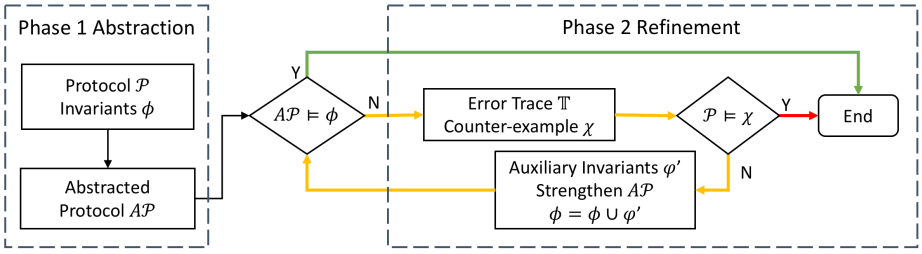

Fig. 2: Workflow of CMP

– **Abstraction.** The first phase is constructing the abstraction model by parameter abstraction: for a protocol $\mathcal{P}$, remains m nodes, adds the *Other* node, and abstracts rules for node *Other*. The abstracted protocol is denoted as $\mathcal{AP}$.
– **Refinement.** The second phase follows the iterative "counterexample-guided abstraction refinement" paradigm [8] (shown as yellow lines). At first, checking the invariants Φ in $\mathcal{AP}$. If it pass the model checker (shown as green line), then CMP finished, although this situation is rare; if there is a counter-example, then the following steps will be iteratively implemented: (1) analyzing the error trace $\mathbb{T}$ and counter-example χ, locating a rule r' that caused the error. (2) if the counter-example also happens in $\mathcal{P}$, which means Φ cannot be hold in the original

protocol, then stop verification (shown as red line); else, the counter-example is caused by the relaxation of guard in phase 1, which means the guard of r' needs to be strengthened. (3) formulating an auxiliary invariant ϕ' which, if true, strengthen the guard of r' such that the counter-example χ becomes unreachable, and adding ϕ' into the set of invariants Φ. The procedure is completed when $\mathcal{AP}$ satisfies ϕ and all the newly-found auxiliary invariants ϕ'.

However, there are two serious obstacles in automating guard strengthening. Firstly, the condition in if-statements also matter. There are if-statements in the action of transition rules, and the condition of these if-statements also accounts for the guard strengthening. The detailed example is given as follow. Thus, it is necessary to take them into consideration when guard strengthening. Secondly, assignments consist of local and global variables need the help of auxiliary variables. This situation happens in parameterized abstraction process, and usually happens when dealing with data-related assignments. For an assignment whose left side is a global variable while the right side is a local variable which needs to be abstracted, the abstraction of it needs to be carried out with some replacement.

Now we take the rule RecvInvAck in German protocol as an example:

```
ruleset i : NODE do rule "RecvInvAck"
    Chan3[i].Cmd = InvAck & CurCmd != Empty
==>
    Chan3[i].Cmd := Empty; ShrSet[i] := false;
    if (ExGntd = true) then
        ExGntd := false; MemData := Chan3[i].Data;
    endif;
endrule; endruleset;
```

Problem 1. The condition in if-statements. Strictly speaking, conditions in if-statement are also special guards which needs strengthening and abstraction by applying invariants. E.g., we need an invariant Chan3[i].Cmd = InvAck & ExGntd = true $\rightarrow$ Chan3[i].Data = AuxData to do CMP to the rule. Notice that the conjunctions of the antecedents are from both the guard and the condition of if-statement. In the previous work, people manually combine parts of the guard and the condition in the if-statement to figure out an invariant, and do CMP on the rule. We need to deal with the condition in the if-statement in a similar way as we deal with the guard.

Problem 2. The assignment consists of local and global variables. We can see that there is an assignment MemData := Chan3[i].Data whose left side of assignment symbol is a global variable, while right side is a local variable. When parameterized abstraction, the right side needs to be abstracted, leaving left side undecidable. Thus, We need identify such assignments and corresponding invariants to replace the local variable.

To solve these two problems, we adopt the following solutions:

Solution 1. Protocol normalization. To normalize rules with if-statement in action, we develop a tool as an expansion of Murphi. It automatically deal with

rules and split a rule into two sub-rules, then add condition and its negation to the guards respectively. Here is the normalization of rule RecvInAck.

```
ruleset i : NODE do rule "RecvInvAck-1"
  Chan3[i].Cmd = InvAck & CurCmd != Empty & ExGntd = true
==>
  Chan3[i].Cmd := Empty; ShrSet[i] := false; ExGntd := false; MemData := Chan3[i].Data;
endrule; endruleset;

ruleset i : NODE do rule "RecvInvAck-2"
    Chan3[i].Cmd = InvAck & CurCmd != Empty & ¬ExGntd = true
==>
    Chan3[i].Cmd := Empty; ShrSet[i] := false;
endrule; endruleset;
```

Solution 2. Local variable replacement. To conquer this obstacle, we firstly strengthen the guard iteratively until no new guards can be added; then establish a mapping function which maps a local variable into an expression according to the equations in the strengthened guard, and replace the local variable with its mapped value in the dictionary. For instance, we apply the invariant Chan3[i].Cmd = InvAck & ExGntd = true → Chan3[i].Data = AuxData to strengthened the guard with a conclusion Chan3[i].Data = AuxData; then map Chan3[i].Data to AuxData according to the mapping function; finally replace Chan3[i].Data with AuxData in the action of rule. The mapping and replacement process is realized automatically.

At last, we recite the result in [9] to establish theoretical foundation of CMP.

Lemma 1. *Let* $M < N$, $M < \mathsf{Other} \leq M$, $P =< I_N, R_N >$ *be a protocol, and* $Q =< I_M, R_M \cup \{\alpha(r(\mathsf{Other}).r \in R)\} >$ *is an abstracted protocol by applying CMP for some abstraction functions* α *to an node* Other *s.t.*

1. $I_M \longrightarrow I_N$
2. $\mathsf{guard}(r(\mathsf{Other})) \wedge inv \longrightarrow \mathsf{guard}(\alpha(r(\mathsf{Other})))$
3. $\forall\ a\ .e\ .i \leq M.\ r \in R.\ as \in \mathsf{action}(\alpha(r(\mathsf{Other}))).as = (a[i] := e(i)) \longrightarrow (i \leq M \wedge as \in \mathsf{action}(r(\mathsf{Other})))$
4. $\forall\ u\ e\ r \in R.\ as \in \mathsf{action}(\alpha(r(\mathsf{Other}))).as = (u := e) \longrightarrow (\exists e'.u := e' \in \mathsf{action}(r(\mathsf{Other})) \wedge (\mathsf{guard}(r(\mathsf{Other})) \wedge inv \longrightarrow e = e'))$
5. $\forall\ a\ .e\ .i \leq M.\ r \in R.\ as \in \mathsf{action}((r(\mathsf{Other})).as = (a[i] := e(i)) \wedge i \neq \mathsf{Other} \longrightarrow (\exists e.a[i] := e(i) \in \mathsf{action}(\alpha(r(\mathsf{Other})))$
6. $\forall\ u\ e\ r \in R.\ as \in \mathsf{action}((r(\mathsf{Other}))).as = (u := e) \longrightarrow (\exists e.u := e' \in \mathsf{action}(\alpha(r(\mathsf{Other}))) \wedge (\mathsf{guard}(r(\mathsf{Other})) \wedge inv \longrightarrow e = e'))$
7. $Q \models inv$,

then $P \models inv$.

In Lemma 1, 1 means that the initial predicate I_M implies I_N; usually I_M is a projection of I_N on nodes $i \leq M$; 2 demonstrates the original guard of some rule $r(other)$ should be stronger than that of the abstracted one; On one side, 3 states that one of the remaining assignment to a local variable $a[i]$ in the abstracted statement, and $\mathsf{action}(\alpha(r(\mathsf{Other})))$ should also be an assignment in

the original statement $\mathsf{action}(r(\mathsf{Other}))$, where i should be an remaining node i s.t. $i \leq M$. Usually such an assignment is a part of an forall statement; 4 means if one of the remaining assignment to a global variable u with an expression e in the abstracted statement $\mathsf{action}(\alpha(r(\mathsf{Other})))$ exists, then there is an assignment $u := e'$ in the original statement $\mathsf{action}(r(\mathsf{Other}))$ s.t. $e = e'$ under the assumption $\mathsf{guard}(r(\mathsf{Other}))$; On the other side, 5 says that the assignment $a[i] := e(i)$ in the original statement $\mathsf{action}((r(\mathsf{Other}))$ s.t. $i \leq M$ should also remained in the abstracted one; 6 means that there is an assignment $u := e$ in the original statement, then there should be $u := e'$ in the abstracted statement s.t. $e = e'$ under the assumption $\mathsf{guard}(r(\mathsf{Other}))$; 7 states that the abstracted protocol Q satisfies the invariant inv. If all the above conditions satisfies, then $P \models inv$.

2.3 Association Rule Mining

Association rule mining is one of the important techniques in data mining area [21]. It is usually used to find frequent patterns, correlations, or casual structures from databases [21]. In this paper, we apply association rule mining to explore auxiliary invariants from $\mathsf{RS}(\mathcal{R})$ (after symmetry reduction). The basic concepts and measurements of association rule mining can be seen in [22,23]. In this paper, we utilize two important constrains, support and confidence, to generate frequent sets and to learn association rules.

The core problem lies in how to set this two constrains. We first elaborate our goals: (1) We hope to let in as many as possible items, but the size of frequent sets is better to be as small as possible, due to extremely huge time consumption of learning association rules. (2) We hope to maximize the certainty of association rules, so that they are more possible to become auxiliary invariants. For these purposes, we analyze the distribution of features in the reachable set we collect, and found that the percentage of some features are fairly small (e.g., Sta.Proc[j].CacheState = CACHE_E only takes up 0.0026 in Flash protocol). Thus, we set minimum support $\theta = 0$, which equals to generate all possible combinations upon the itemset. And for confidence, we set minimum confidence $\delta = 1$ so that we can at least make sure that the learned association rules hold in $\mathrm{RS}(\mathcal{R})$ (although symmetry reduction technique may cause spurious invariants, which will be discussed later). It is noticeable that the size of frequent set is also constrained. Because if without the constrain of the frequent set size, the amount of association rules will be extremely huge, which is a burden for the following verification. Thus, we carry out a series of experiments to find the minimum size of frequent set that can generate sufficient association rules. To our surprise, the answer is fairly small, we only need at most 3-item frequent sets to calculate association rules.

3 An Overview of Our Method

Our work can be divided into two phases (shown as Figure. 3).

Learning Invariants. In this phase, we first collect $mathsfRS$ from small instance of protocols as dataset. And a remarkable difference from other feature engineering works is that we directly extract features from protocol description,

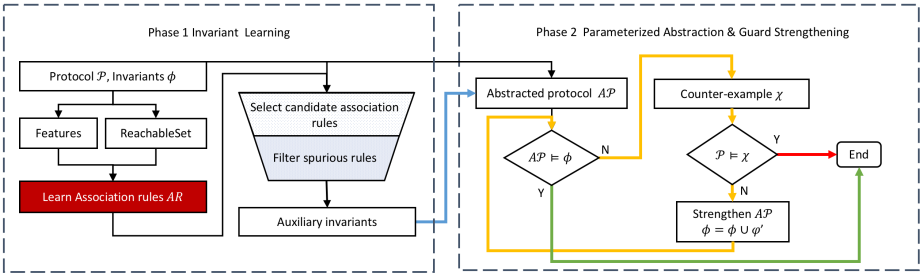

Fig. 3: **Framework of Learning Based Parameter Abstraction**Our method can be divided into two parts.

without laborious attempts for finding effective features. Then feed dataset and extracted features into association rule learning algorithm, which is modified and set constrains to adapt to guard strengthening. The resulting association rules are in a large amount, thus we need to select the useful ones from them. Because the purpose is to strengthening guard of rules, we select association rules whose antecedent can be implied by guards of rules. This step help us to shrink candidate rules. Next step is carried out in the aid of Murphi. Iteratively adding these candidate rules into the original protocols, and eliminating the spurious ones. In fact, this step helps us to select invariants in limited sizes, so they are actually local truths. Thus, once these local truths are used to strengthen protocols, they also need to be checked in the refined protocols.

Parameterized Abstraction and Guard Strengthening. This phase works similarly with CMP. The color lines in red and green work as the same. Noticeable that the auxiliary invariants are applied to strength guards before counter-example happens. And this guard strengthen is actually an iterative process until no more guards can be refined. Then checked the strengthened abstracted protocol. If it holds for all original and used invariants in model checker, then end the verification. If counter-example happens, then we come back to learning phase to enlarge the size of frequent sets for association rule learning.

4 Detailed Account

4.1 Learning invariants

Protocol normalization. As mentioned in Section 2.2, the condition of an if-statement also matters when refine the abstracted protocol. Thus, we develop a program t normalize the protocol automatically in the following way: (1) A rule contains if-statements will be divided into two sub-rules; normalization will be carry out infinitely until no if-statement occurs at the top level in the action of rules; (2) A rule whose guards contains disjunction (e.g., $\phi \vee \psi$) will be divided into two sub-rules with guards ϕ and ψ separately; (3) A rule whose guards contains an implication (e.g., $\phi \rightarrow \psi$) will be equally transferred to two sub-rules with $\neg\phi$ and ψ separately. After normalization, a guard of a normalized rule is in cubic form, namely, conjunctions of atom predicates or forall-formulas,

and no if-statement exists at the top-level in the action of rules. Semantically the normalized protocol maintains the same behavior as the original one.

Dataset Collection. We collect the dataset from the output of Murphi. Detailed statistics are listed in Figure. 5. Noticed that Murphi applies symmetry and multiset reduction techniques to reduce the memory requirements [24], which brings about both pros and cons to our work. It benefits us with a quite smaller RS, decreasing time consumption of association rule learning largely. For example, the usage of 'undefine' narrows the scale of 3-node Flash from billions down to millions level. This dramatical reduction avoids the state explosion and saves a large amount of time for association rule learning. Yet, it may also cause spurious invariants. To eliminate these spurious association rules, we need to select invariants in the following step.

Data Analysis. We analyze the RS to acquire a comprehensive view of the distribution of features. We found that some features only account for a very small percentage. In the Figure. 4, we illustrate the percentage of every predicates in German and Flash protocols. We can see that the lowest rate is 0.0211 in German , and 0.0026 in Flash . While these features are important in our experiments, because when they are ruled out from features, the association rules cannot effectively strengthen the guard. This observation suggests us when setting constrains for association rule learning, the minimum support needs to be as small as possible so that each feature has opportunity to be considered.

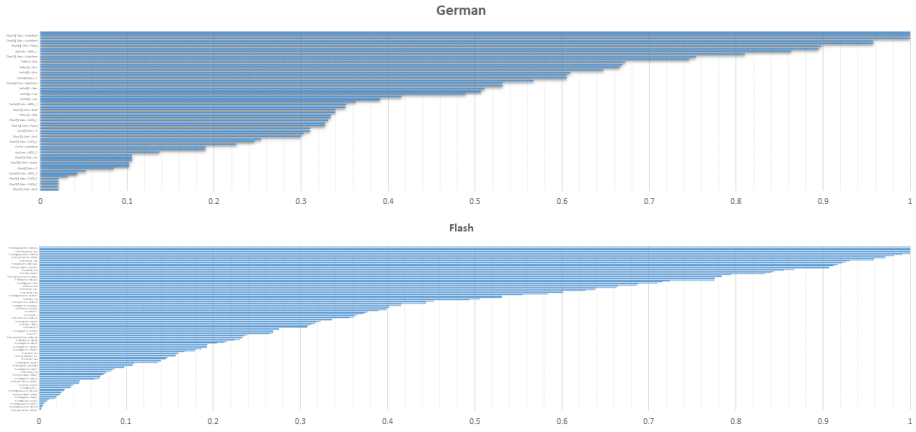

Fig. 4: **Proportions of predicates for German and Flash** The datasets are collected from 2-node instance of German protocol (with 852 states), and 3-node instance of Flash protocol (with 1350226 states) respectively.

Feature Finding. In intuitive thinking, a variable corresponds to a feature, or item in association rule learning. However, we found that it is insufficient and ineffective to only take variables as features. Let us consider two situations in German protocol:

1. Variables such as Cache[i].Data = DATA_1 are not useful features, because the data assignment can be replaced by any other value. While predicates like Chan3[i].Data = AuxData do matters, no matter which specific value is assigned to Chan3[i].Data and AuxData. Thus, we need to formulate sufficient predicates (serve as features) that can be inferred from protocol, and are meaningful for later association rule learning.
2. Predicates such as ExGntd = True mean that there is a process i whose variable Cache[i].State is set to be E. On the contrary, if ExGntd is set to be False, Cache[i].State has two optional assignments (I and S). It is unsure that which exact assignment is for the process i, but we can ensure its state can not be E. Thus it is also necessary to take the negative pattern for each meaningful predicates.

Therefore, we need to find more effective features for the following learning steps. Then we realize that we already have a comprehensive view of how the protocol works, since the mechanism of protocol has already been written, which means there is no need to do blindly attempts for finding useful features. Feature engineering is trying to explore how the unknown world works because they have no idea about what the regulations run behind, while different from these works, the situation we confront is an defined protocol. It is fairly natural and intuitive to extract useful features directly from the protocol itself. Thus, we proposed an algorithm to search closures of atomized predicates from guards of rules and properties of the protocol instances under the so-called weakest precondition operator WP, as shown below:

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

where $\mathsf{WP}(A, f)$ substitutes each occurrence of a variable v_ℓ by the corresponding e_ℓ in formula f.

Initially our algorithm computes $A_0 = \bigcup\{\mathsf{atoms}(f).f \in (\{\mathsf{guard}(r).r \in R\} \cup invs)\}$, where $\mathsf{atoms}(f)$ returns atomic predicate of f, R and $invs$ are the set of rules and properties under verification; then it continues to compute $A_{i+1} = \{\mathsf{WP}(\mathsf{action}(r), f).f \in A_i, r \in R\}$ until $A_{i+1} = A_i$. At the termination of the returned A_n is the set of atomic formulas, which will be regarded as meaning features for machine learning.

Dataset Preprocessing After finding meaningful features, it is trivial to reconstruct dataset according to these features. For predicates in form of $v_\ell := e_\ell$, is it easy to assign True or False by comparing value of v_ℓ in each state with e_ℓ. While for those in form of $v_{\ell_1} := v_{\ell_2}$, there are two situations: (1) at least one of v_{ℓ_1} and v_{ℓ_2} is undefined; (2) none of them is undefined. For the first situation, the predicate $v_{\ell_1} := v_{\ell_2}$ is undecidable, thus set the value to be 'Undefined'. For the second situation, e_{ℓ_1} for v_{ℓ_1} and e_{ℓ_2} for v_{ℓ_2} can be compared, results in True or False for this predicate. After preprocessing, the dataset will be consist of binary values.

Association Rule Learning As we mentioned before, the informative predicates take up low proportion, so the minimum support θ needs to be set as

small as possible. We set $\theta = 0$ so that every predicates has possibility to form association rules. As for minimum confidence δ, we set $\delta = 1$ because we hope to acquire rules with 100% certainty in the dataset.

For finding frequent item sets and learning association rules, there are constrains are made to adopt to our learning framework.

- **Frequent sets finding** Generating candidate frequent set takes a large amount of time, especially when the dataset is at the million level and the minimum support is set to be extremely small. Thus, we carry out learning algorithm by adding the size of frequent set one by one. To our surprise, association rules learned from up to 3-item frequent sets are adequate to strength the abstracted protocol, which not only saves large amount of time, and provides a simply and straightforward format to easily strength abstracted guard commands.
- **Association rules learning** In the learning process, we make a little different about the association rules. If the consequent(also known as right-hand-side) of an association rules has more than one item, then we divided them into two or more sub-rules, so that the association rules are in the from of $\bigwedge_{i=1}^{M} \phi_i \longrightarrow \psi$ s.t. $M < 3$ (generated from k-item frequent sets where $k < 4$). In our experience of parameterized verification of cache coherence protocols, M, the upper bound of the number of antecedent of an association rule is 3. This restriction is important because we still will meet the problem of explosion if we enlarge this upper bound. It is a trivial modification so that we can acquire association rules in the unified format. Moreover, more association rules acquires, more invariants we need to verified, which takes an extremely extra time.

Candidate Invariant Selection The result association rules may be extremely large amount, which is a burden for the later verification step. Thus we select association rules whose condition(also known as left-hand-side) is the subset of guards of any transition rules in the protocol. This step helps us to save considerable time.

Invariant Selection Candidates mined from RS are not necessarily the invariant. There are two kinds of reasons may bring about :

- **Local truth.** Although we have set $\theta = 1$, the necessity of these candidates can only be guaranteed within the RS of small instances, which means they are local truths in the parameterized system. For example, in a 2-node instance of Mutual Exclusion protocol, when two nodes are in state of I (represents invalid), then the global variable x can be implied to be True. While this inference fails when more nodes are added.
- **Spurious invariant.** Spurious invariants may be cause by symmetry reduction technique of model checker Murphi. For example, in the RS of 2-node German instance, association rule Proc[j].CacheState = CACHE_E $\rightarrow$ Sta.Pending = True can be calculated, which means if a state satisfies the predicate Sta.Proc[j].CacheState = CACHE_E, then it should also satisfies

Sta.Pending = True. However, this association rule can not be true according to the protocol description.

After eliminating the local truths and spurious invariants, which usually spends the most of time in invariant learning process, we now acquire a set of auxiliary invariants. In fact, these auxiliary invariants only checked in small instance of protocol, which means they are actually local truths. Thus, once these auxiliary invariants are used to strengthen protocols, they also need to be checked in the refined protocols.

Invariant Analysis We analyze the form of invariants, and classify them into three paradigms:

1. Invariants only contain global variables (e.g. Inv_58). This kind of invariants escapes from the parameterized abstraction because the abstraction process only has effect on local variables.
2. Invariants contains local variables with one kind of index (e.g. Inv_24, 139). For Inv_24, we can see that a local variable can imply a global variable, which means the global variable play its part no matter whether the local variable abstracted or not. While for Inv_139, index appears in both sides, which means the the right side of the invariants may be abstracted and has no effect on strengthening.
3. Invariants contains two kinds of indexes (e.g. Inv_305, 36). This kind of invariants is unique because they indicate the interaction between two processes on the concurrent system, where one process can directly affects the other (Inv_36), or two processes collective effect global variables (Inv_305).

```
Auxiliary invariants used for guard strengthening of Flash protocol:

Inv_58: Sta.Dir.Pending = false → Sta.FwdCmd = UNI_None
Inv_24: Sta.InvMsg[i].Cmd = INV_InvAck → Sta.Dir.Pending = true
Inv_139: Sta.InvMsg[i].Cmd = INV_InvAck → Sta.Proc[i].CacheState != CACHE_S
Inv_305: Sta.UniMsg[i].Cmd = UNI_Get & Sta.UniMsg[i].Proc = j → Sta.FwdCmd = UNI_Get
Inv_36: Sta.Proc[i].CacheState = CACHE_E → Sta.Dir.ShrSet[j] = false
```

4.2 Parameterized Abstraction and Guard Strengthening

In this subsection, we demonstrate the detailed implementation of automating parameterized abstraction and guard strengthening using the above learned auxiliary invariants. Our work distinguish CMP from that we propose a more aggressive method to strengthen guards before counter-example happens. We also conquer the obstacles mentioned in section. 2.2, which optimizes the abstraction and strengthening processes. Now, let us introduce some notations for the further discussion.

- $\mathsf{filter}((\lambda\ x.P\ x), xs)$ selects the elements in list xs, which satisfy property P.
- $\mathsf{map}((\lambda\ x.f\ x), xs)$ takes a list and a function f for transforming elements of that list xs, and returns a new list with the transformed elements

- $\mathsf{replace}(dict, key, data)$ add a data $data$, which will be retrieved by key key if the pair $(key, data)$ does not exists in the dictionary $dict$; or replace some old data $data'$ with $data$ if $data'$ is already retrieved by key key
- $\mathsf{lookUp}(dict, key)$ returns some data retrieved by key key; or None if there is no data mapped from key.
- $\mathsf{generalize}(f, c)$ returns a formula which replace a concrete parameter c with a variable i_c. E.g., $\mathsf{generalize}(n[1] = \mathsf{I})$ is $n[i_1] = \mathsf{I}$.
- $\mathsf{occur}(i, f) \equiv i \in \mathsf{paramsOf}(f)$, where $\mathsf{paramsOf}(f)$ is the array indexes occurring in f.
  $\mathsf{occur}(c, f)$ states that a index i occurs free in a formula f. For instance, 1 occurs in formula $n[1] = I$.
- $\mathsf{relate}(inv, r) \equiv \mathsf{guard}(r) \longrightarrow \mathsf{ant}(inv)$ The guard of rule r implies the antecedent of an invariant formula inv.
- $\mathsf{asgn2Lv}(as, i)) \equiv \exists a\ e.as = (a[i] := e)$. The assignment statement as assigns some value of expression e to an array-element variable $a[i]$.
-

$$\mathsf{drawCon}(f, c) \equiv \begin{cases} f & \text{if } \mathsf{occur}(c, f) \\ \forall i_c.\mathsf{generalize}(f, c) & \text{otherwise} \end{cases} \tag{1}$$

- $\mathsf{drawCons}(fs, i) \equiv \{\mathsf{drawCon}(f, i) | f \in fs\}$.

Now we begin to introduce how to automate CMP using auxiliary invariants learned in subsection 4.1. Function gStrEn recursively strengthens the guard of r by deriving conclusions of F. Statement 1 selects invariants whose antecedents is implied by the guard of rule r. If there is no such invariants, r is directly returned; else statement 5 draws the consequents of invariant formulas. Statement 6 fetches the guard and statement parts of of rule r into g and act respectively. Statement 7 strengthens g with the conjunction of all conclusion obtained in step 5, and statement make the strengthened one to be the guard of the new rule. Notice that all the conclusions added are implied by the conjunction of the guard of the rule and F, therefore the new guard g' is implied by the conjunction of g and F.

Function gStrEn$(r, F, other)$

Input: a rule r, a list of formulas F, an parameter $other$
Output: a new rule

1 $F' \leftarrow \mathsf{filter}((\lambda\ f.\ \mathsf{relate}(f, r)), F)$
2 **if** $F' = \emptyset$ **then**
3 $\mathsf{return}(r)$
4 **else**
5 $CF \leftarrow \mathsf{drawCons}(F', other)$
6 $(\bigwedge G) \rhd act \leftarrow r$
7 $g' \leftarrow \bigwedge(CF \cup G)$
8 $\mathsf{gStrEn}(g' \rhd act, F - F', other)$

Function makedictItem$(dict, eq)$

Input: a dictionary r, an atomic formula eqf
Output: a new dictionary

1 **if** $\exists\ a\ j\ u.(eqf = (a[j] = u) \vee eqf = (u = a[j])) \wedge i = j$ **then**
2 $dict \leftarrow \mathsf{replace}(dict, a[i], u)$
3 $\mathsf{return}(dict)$
4 **else**
5 $\mathsf{return}(dict)$

Based on the aforementioned dictionary, we replace the local variable $a[Other]$ with a global variable v, where $a[Other]$ occurs in the right side of assignment $u := a[Other]$ to a global variable u. According to the dictionary, $a[Other] = v$ is implied by the conjunction of $\mathsf{guard}(r)$ and $invs$.

Function makedictItem($dict, eq$)

Input: a dictionary r, an atomic formula eqf

Output: a new dictionary

1 **if** $\exists\ a\ j\ u.(eqf = (a[j] = u) \vee eqf = (u = a[j])) \wedge i = j$ **then**
2 $dict \leftarrow \mathsf{replace}(dict, a[i], u)$
3 $\mathsf{return}(dict)$
4 **else**
5 $\mathsf{return}(dict)$

Function cmp($r, invs, i, other$)

Input: a parameterized rule r with a symbolic parameter i, a parameter $other$

Output: a new rule of the abstracted node indexed by $other$

1 $r^c \leftarrow r[i \mapsto other]$
2 $civs \leftarrow \mathsf{instByPolicy}(invs, policy)$
3 $r' \leftarrow \mathsf{guardStrengthen}(r^c, cinvs, other)$
4 $\mathsf{return}(\mathsf{abstract}(r', other))$

Function refineAct($dict, as$)

Input: a dictionary r, an assignment statement as

Output: a new statement

1 **if** $\exists\ a\ u.as = (u := a[other])$ **then**
2 $result \leftarrow \mathsf{lookUp}(dict, a[other])$
3 **if** $(result = \mathsf{None})$ **then**
4 $\mathsf{error}(\text{“}can't\ refine\text{”})$
5 **else**
6 $\mathsf{Some}(v) \leftarrow result$
7 $\mathsf{return}(u := v)$
8 **else**
9 $\mathsf{return}(as)$

Function abstract($r, other$)

Input: a rule r, an index $other$

Output: a new rule

1 $g \rhd act \leftarrow r$
2 $\mathsf{Parallel}(acts) \leftarrow act$
3 $\bigwedge G \leftarrow g$
4 $A \leftarrow \mathsf{filter}((\lambda\ a.\neg\mathsf{asgn2Lv}(a, other)), acts)$
5 **if** $A = \emptyset$ **then**
6 $\mathsf{return}(\mathsf{None})$
7 **else**
8 $dict \leftarrow \mathsf{mkDict}(G, other)$
9 $acts' \leftarrow \mathsf{map}((\lambda\ as.\mathsf{refineAct}(dict, as) \neq \mathsf{None}), A)$
10 $G' \leftarrow \mathsf{filter}((\lambda\ g.\ \mathsf{occur}(other, g), G)$
11 $act' \leftarrow \mathsf{Parallel}(acts')$
12 **if** $\mathsf{occur}(i, act')$ **then**
13 error
14 **else**
15 $\mathsf{return}(\mathsf{Some}(\bigwedge G' \rhd act'))$

In order to deal with the update of the right side of an assignment aforementioned, we first make a dictionary which maps a local variable $a[Other]$ into a global variable v if $a[other] = v$ or $v = a[other]$ if the above equations can be

implied. According to the set of all the equations like those which are found in the guard g, we make the dictionary. By the the definition of the dictionary, we have $g \wedge invs \longrightarrow a[other] = v$.

Recall that the local variables of the Other is unobservable, thus the assignments for Other will be abstracted, remaining those non-Other nodes. Statement 4 select the assignments to non-Other nodes. If the set of such assignments is empty, then None is returned, which means the abstraction of the rule will be skipped; else a dictionary will be created according to the guard of the rule and will be used to substitute all the local variables of the *other* node occurring in the right side of an assignment. At the same time, the parts on the Other node occurring in G will be removed (or abstracted) because the local state variables of the Other node is not observable. A safety check will be done to guarantee that local variables of the Other node does not occur in the right side of an assignment. If this condition is not satisfiable, an error occurs, else an option denoting the abstracted rule $g' \rhd act'$ is returned.

Function cmp$(r, invs, i, other)$ firstly computes a set of concrete invariants by instantiating the parameterized invariants into concrete invariants $cinvs$ and instantiate the parameterized rule into a concrete rule r^c with an actual parameter c with some parameter instantiation policy, calls guardStrengthen to strengthen the guards of the rule r^c by using the concrete invariants $cinvs$, then calls abstract to abstract the rule.

4.3 Experiment Result

We apply LB-CMP to some classical protocols. The running environment is Linux 4-core, 16G. Detailed statistics are listed as follow:

Protocol name	# Node	# States	# Association rules	# Candidate Rules	# Invariants	# Invariants used for strengthening	# States of abstracted protocol	Time for association rules learning	Time for selecting invariants
Moesi	2	6	147	19	2	0	7	0.004 s	5.176 s
Mesi	2	5	80	25	4	1	5	0.003 s	4.676 s
MutualEx	2	12	450	27	16	5	18	0.009 s	8.042 s
German	2	852	21202	369	239	30	1314	1.433 s	88.714 s
Flash-no-data	3	784637	224946	2805	268	158	33544184	8930.652 s	631.460 s
Flash	3	1350226	381528	3263	651	333	26962920	21909.092 s	696.654 s

Fig. 5: **Experiment results**

The final abstracted protocol of *Mutual Exclusion* is as follow. We give a concisely explanation of the entire process: The parameter abstraction is done by following the instruction in [8], and the automation implement is trivial, so we do not go into detail in this paper. We now demonstrate how to use learned and filtered invariants to refine guarded commands. For Mutual Exclusion protocol, we obtain five rules after the above-mentioned steps. For rule "Try" and "Exit", their actions will be abstracted, so we do not need to refine them. For rule "Crit", its original guard is "n[i] = T& x = true". Look up auxiliary invariants and find out the ones whose precondition can be implied by these guards. We can see

that rule_1 and 2 are conditioned by "x = true", so we add the consequence "n[i] != E" and "n[i] != C" in the guard of "ABS_Crit". The same process will be done to rule "Idle". It is noteworthy that if the index of i or j exists in the consequence, then the consequence needs to be generalized to all NODE if the same index does not exist in the condition.

```
TYPE                          rule "ABS_Crit"
union {NODE, enum{Other}}     x = true & forall j : NODE do
                                  (n[j] != C) & (n[j] != E)   rule_1: x = true -> n[j] != E
                              end ==>                         rule_2: x = true -> n[j] != C
startstate                        x := false; end;
for i: NODE do
    n[i] := I;                rule "ABS_Idle"
endfor;                       x = false & forall j : NODE do  rule_3: n[i] = E -> n[j] != E
    x := true;                    (n[j] != C) & (n[j] != E)   rule_4: n[i] = E -> n[j] != C
endstartstate;                end ==>                         rule_5: n[i] = E -> x = false
                                  x := true; end;
```

5 Conclusion and Future Work

In this paper, we propose an automatic framework which can automatically learn auxiliary invariants, abstract parameter and strengthen abstracted protocols in a unified framework. The originality of our work lies in the following aspects:

1. Instead of analyzing counter-example to formulate invariants manually, we derive straightforward invariants from reachable set of protocols;
2. Instead of strengthening protocols after counter-example happens, we refine the guard of rules before error happens, which is a more aggressive method to strengthen protocols.

Compared with other automatic techniques such as [14,1], the invariants derived from our framework is in the most simple and straightforward format. We only need the form of $\bigwedge_{i=1}^{N} f_i \longrightarrow f'$ where $N \leq 2$. While in [14,1], they used invariant in form of $\neg \bigwedge_{i=1}^{N} f_i$ to give a formally inductive proof where $N = 4$.

For the further research, we plan to extend our work from the following directions. First, completing the mechanical proof of the simulation relation between the abstracted and the original protocol in theorem provers, so that the ultimate correctness of our method can be formally argued. Second, exploring the measurement which can tell the importance of invariants, because some invariants can dramatically scale up the reachable set or guarantee the invariants, while others only have little influence. If we can identify these influential invariants, the process of LB-CMP can be optimized. Third, expanding the ability of LB-CMP to prove general safety and liveness properties. We also want to use the LB-CMP to learn invariants in loops to prove properties of loop programs.

As we demonstrate in this paper, combining machine learning algorithms, or more specific, data mining techniques, with parameterized verification is a possible and promising direction, which not only works as an aid of parameterized verification, but shows the possibility of combination between these two fields.

References

1. Y. Li, K. Duan, Y. Lv, J. Pang, and S. Cai, "A novel approach to parameterized verification of cache coherence protocols," in *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE, 2016, pp. 560–567.
2. Y. Lv, H. Lin, and H. Pan, "Computing invariants for parameter abstraction," in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 2007, pp. 29–38.
3. K. R. Apt and D. C. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Information Processing Letters*, vol. 22, no. 6, pp. 307–309, 1986.
4. S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi, "Invariants for finite instances and beyond," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 2013, pp. 61–68.
5. A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," in *TACAS*, vol. 1. Springer, 2001, pp. 82–97.
6. T. Arons, A. Pnueli, S. Ruah, Y. Xu, and L. Zuck, "Parameterized verification with automatically computed inductive assertions?" in *International Conference on Computer Aided Verification*. Springer, 2001, pp. 221–234.
7. K. L. McMillan, "Parameterized verification of the flash cache coherence protocol by compositional model checking," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 2001, pp. 179–195.
8. C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *FMCAD*, vol. 4. Springer, 2004, pp. 382–398.
9. S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," *Automated verification of infinite state systems*, 2005.
10. A. Pnueli and E. Shahar, "A platform for combining deductive with algorithmic verification," in *Computer Aided Verification*. Springer, 1996, pp. 184–195.
11. X. Chen and G. Gopalakrishnan, "A general compositional approach to verifying hierarchical cache coherence protocols," Technical Report, UUCS-06-014, School of Computing, University of Utah, Salt Lake City, UT 84112 USA. Noverber 26, Tech. Rep., 2006.
12. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 113–127, 2001.
13. S. Pandav, K. Slind, and G. Gopalakrishnan, "Counterexample guided invariant discovery for parameterized cache coherence verification," in *CHARME*. Springer, 2005, pp. 317–331.
14. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaıdi, "Cubicle: A parallel smt-based model checker for parameterized systems," in *CAV*. Springer, 2012, pp. 718–724.
15. M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2008, p. 10.
16. 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.
17. X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou, "Reducing verification complexity of a multicore coherence protocol using assume/guarantee," in *Formal Methods in Computer Aided Design, 2006. FMCAD'06*. IEEE, 2006, pp. 81–88.

18. S. K. Lahiri, R. E. Bryant, and A. E. Bryant, "Constructing quantified invariants via predicate abstraction," in *VMCAI*, vol. 2937. Springer, 2004, pp. 267–281.
19. K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," *Lecture Notes in Computer Science*, vol. 4963, pp. 413–427, 2008.
20. 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.
21. C. Kaur, "Association rule mining using apriori algorithm: A survey," *International Journal of Advanced Research in Computer Engineering & Technology*, vol. 2, no. 6, pp. 2081–2084, 2013.
22. R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Acm sigmod record*, vol. 22, no. 2. ACM, 1993, pp. 207–216.
23. R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.
24. 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.

6 Appendix

Due to the space limitation, we can not depict our automated LB-CMP detaily in the main text. Instead, we use more complex case studies of German and Flash to illustrate how our LB-CMP works. Let us review the normalized German protocol[8] as follows:

6.1 German protocol

```
const ---- Configuration parameters ----

NODE_NUM : 4;
DATA_NUM : 2;

type ---- Type declarations ----

NODE : scalarset(NODE_NUM);
DATA : scalarset(DATA_NUM);

ABS_NODE : union NODE, enumOther;

CACHE_STATE : enum I, S, E;
CACHE : record State : CACHE_STATE; Data :
DATA; end;

MSG_CMD : enum Empty, ReqS, ReqE, Inv, InvAck,
GntS, GntE;
MSG : record Cmd : MSG_CMD; Data : DATA; end;

var ---- State variables ----

Cache : array [NODE] of CACHE; -- Caches
Chan1 : array [NODE] of MSG; -- Channels for
Req*
Chan2 : array [NODE] of MSG; -- Channels for
Gnt* and Inv

Chan3 : array [NODE] of MSG; -- Channels for
InvAck
InvSet : array [NODE] of boolean; -- Set of
nodes to be invalidated
ShrSet : array [NODE] of boolean; -- Set of
nodes having S or E copies
ExGntd : boolean; -- E copy has been granted
CurCmd : MSG_CMD; -- Current request command
CurPtr : ABS_NODE; -- Current request node
MemData : DATA; -- Memory data
AuxData : DATA; -- Auxiliary variable for
latest data

---- Initial states ----

ruleset d : DATA do startstate "Init"
for i : NODE do
Chan1[i].Cmd := Empty; Chan2[i].Cmd := Empty;
Chan3[i].Cmd := Empty;
Cache[i].State := I; InvSet[i] := false;
ShrSet[i] := false;
end;
ExGntd := false; CurCmd := Empty; MemData :=
d; AuxData := d;
end end;


-- Invariant properties --

invariant "CntrlProp"
forall i : NODE do forall j : NODE do
i != j -> (Cache[i].State = E ->
Cache[j].State = I) &
(Cache[i].State = S -> Cache[j].State = I |
Cache[j].State = S)
end end;

invariant "DataProp"
( ExGntd = false -> MemData = AuxData ) &
forall i : NODE do Cache[i].State != I ->
Cache[i].Data = AuxData end;
```

```
---- State transitions ----

ruleset i : NODE do
rule "RecvGntE1"
Chan2[i].Cmd = GntE
==>
Cache[i].State := E;
Cache[i].Data := Chan2[i].Data;
Chan2[i].Cmd := Empty;
undefine Chan2[i].Data;
endrule; endruleset;


ruleset i : NODE do
rule "RecvGntS2"
Chan2[i].Cmd = GntS
==>
Cache[i].State := S;
Cache[i].Data := Chan2[i].Data;
Chan2[i].Cmd := Empty;
undefine Chan2[i].Data;
endrule; endruleset;

ruleset i : NODE do
rule "SendGntE3"
CurCmd = ReqE &
CurPtr = i &
Chan2[i].Cmd = Empty &
ExGntd = false &
forall j : NODE do
ShrSet[j] = false
end
==>
Chan2[i].Cmd := GntE;
Chan2[i].Data := MemData;
ShrSet[i] := true;
ExGntd := true;
CurCmd := Empty;
undefine CurPtr;
endrule; endruleset;


ruleset i : NODE do
rule "SendGntS4"
CurCmd = ReqS &
CurPtr = i &
Chan2[i].Cmd = Empty &
ExGntd = false
==>
Chan2[i].Cmd := GntS;
Chan2[i].Data := MemData;
ShrSet[i] := true;
CurCmd := Empty;
undefine CurPtr;
endrule; endruleset;

ruleset i : NODE do
rule "RecvInvAck5"
Chan3[i].Cmd = InvAck &
ExGntd = true
==>
Chan3[i].Cmd := Empty;
ShrSet[i] := false;
ExGntd := false;
MemData := Chan3[i].Data;
undefine Chan3[i].Data;
endrule; endruleset;

ruleset i : NODE do
rule "RecvInvAck6"
Chan3[i].Cmd = InvAck &
ExGntd = false
==>
Chan3[i].Cmd := Empty;
ShrSet[i] := false;
endrule; endruleset;

ruleset i : NODE do
rule "SendInvAck7"
Chan2[i].Cmd = Inv &
Chan3[i].Cmd = Empty &
Cache[i].State = E
==>
Chan2[i].Cmd := Empty;
Chan3[i].Cmd := InvAck;
Chan3[i].Data := Cache[i].Data;
Cache[i].State := I;
undefine Cache[i].Data;
endrule; endruleset;

ruleset i : NODE do
rule "SendInvAck8"
Chan2[i].Cmd = Inv &
Chan3[i].Cmd = Empty &
Cache[i].State != E
==>
Chan2[i].Cmd := Empty;
Chan3[i].Cmd := InvAck;
Cache[i].State := I;
undefine Cache[i].Data;
endrule; endruleset;

ruleset i : NODE do
rule "SendInv9"
Chan2[i].Cmd = Empty &
InvSet[i] = true &
CurCmd = ReqE
==>
Chan2[i].Cmd := Inv;
InvSet[i] := false;
endrule; endruleset;

ruleset i : NODE do
rule "SendInv10"
Chan2[i].Cmd = Empty &
InvSet[i] = true &
CurCmd = ReqS &
ExGntd = true
==>
Chan2[i].Cmd := Inv;
InvSet[i] := false;
endrule; endruleset;


ruleset i : NODE do
rule "RecvReqE11"
CurCmd = Empty &
Chan1[i].Cmd = ReqE
==>
CurCmd := ReqE;
CurPtr := i;
Chan1[i].Cmd := Empty;
for j : NODE do
InvSet[j] := ShrSet[j]; end;
endrule; endruleset;

ruleset i : NODE do
rule "RecvReqS12"
CurCmd = Empty &
Chan1[i].Cmd = ReqS
==>
CurCmd := ReqS;
CurPtr := i;
Chan1[i].Cmd := Empty;
for j : NODE do
InvSet[j] := ShrSet[j];
end;
endrule; endruleset;


ruleset i : NODE do
rule "SendReqE13"
Chan1[i].Cmd = Empty &
Cache[i].State = I
==>
Chan1[i].Cmd := ReqE;
endrule; endruleset;

ruleset i : NODE do
rule "SendReqE14"
Chan1[i].Cmd = Empty &
Cache[i].State = S
==>
Chan1[i].Cmd := ReqE;
endrule; endruleset;

ruleset i : NODE do
rule "SendReqS15"
Chan1[i].Cmd = Empty &
Cache[i].State = I
==>
Chan1[i].Cmd := ReqS;
endrule; endruleset;

ruleset i : NODE;
data : DATA do
rule "Store"
Cache[i].State = E
==>
Cache[i].Data := data;
AuxData := data;
endrule;endruleset;

ruleset d : DATA do
startstate
for i : NODE do
Chan1[i].Cmd := Empty;
Chan2[i].Cmd := Empty;
Chan3[i].Cmd := Empty;
Cache[i].State := I;
InvSet[i] := false;
ShrSet[i] := false;
end;
ExGntd := false;
CurCmd := Empty;
MemData := d;
AuxData := d;
endstartstate;
endruleset;
```

Next, we list the ABS_German model, which is the result returned by our LB-CMP.

```
ABS_NODE: unionNODE, enumOther;
CurPtr: ABS_NODE;

rule "ABS_SendGntE3"
CurCmd = ReqE &
forall j : NODE do
ShrSet[j] = false end &
CurPtr = Other &
MemData = AuxData &
ExGntd = false &
forall j : NODE do
(Cache[j].State != E) &
(Chan2[j].Cmd != GntE) end
==>
ExGntd := true;
CurCmd := Empty;
undefine CurPtr;
end;

rule "ABS_SendGntS4"
CurPtr = Other &
MemData = AuxData &
CurCmd = ReqS &
ExGntd = false &
forall j : NODE do
(Chan2[j].Cmd != Inv) &
(Cache[j].State != E) &
(Chan3[j].Cmd != InvAck) &
(Chan3[j].Cmd = Empty) &
(Chan2[j].Cmd != GntE) end
==>
CurCmd := Empty;
undefine CurPtr;
end;

rule "ABS_RecvInvAck5"
ExGntd = true &
CurCmd != Empty &
forall j : NODE do
(Chan2[j].Cmd = Empty) &
(Cache[j].State = I) &
(Chan2[j].Cmd != Inv) &
(Cache[j].State != S) &
(Cache[j].State != E) &
(Chan2[j].Cmd != GntS) &
(ShrSet[j] = false) &
(Chan3[j].Cmd != InvAck) &
(InvSet[j] = false) &
(Chan3[j].Cmd = Empty) &
(Chan2[j].Cmd != GntE)
end
==>
ExGntd := false;
MemData := AuxData;
end;

rule "ABS_RecvReqE11"
CurCmd = Empty &
forall j : NODE do
(Chan3[j].Cmd != InvAck) &
(Chan2[j].Cmd != Inv) &
(Chan3[j].Cmd = Empty)
end
==>
CurCmd := ReqE;
CurPtr := Other;
for j : NODE do
InvSet[j] := ShrSet[j];
end; end;

rule "ABS_RecvReqS12"
CurCmd = Empty &
forall j : NODE do
(Chan3[j].Cmd != InvAck) &
(Chan2[j].Cmd != Inv) &
(Chan3[j].Cmd = Empty)
end
==>
CurCmd := ReqS;
CurPtr := Other;
for j : NODE do
InvSet[j] := ShrSet[j];
end; end;

ruleset data: DATA do
rule "ABS_Store"
ExGntd = true &
forall j : NODE do
(Chan2[j].Cmd = Empty) &
(Cache[j].State = I) &
(Chan2[j].Cmd != Inv) &
(Cache[j].State != S) &
(Cache[j].State != E) &
(Chan2[j].Cmd != GntS) &
(ShrSet[j] = false) &
(Chan3[j].Cmd != InvAck) &
(InvSet[j] = false) &
(Chan3[j].Cmd = Empty) &
(Chan2[j].Cmd != GntE)
end
==>
AuxData := data;
end;end;
```

Here we elaborate CMP process of the rule RecvInvAck5 for the abstraction node $Other = 3$. According to our CMP function in Section 2.2, it will choose a proper parameter instantiation policy to instantiate the set of parameterized invariants. A simple but robust instantiation policy is to instantiate formal parameters with all possible actual parameters $n \leq M$, shown as below:

- For an invariant form with one parameter such as $((ExGntd = true) - > (!(Cache[i].State = S))$, we instantiate the parameter i with $[1], [2], [3]$ resepctively.
- For an invariant form with one parameters such as $(((Chan3[i].Cmd = InvAck) \& (ExGntd = true)) - > (!(Chan2[j].Cmd = Inv)))$, we instantiate i and j with two distinct constants c_i and c_j in the rang3 1..3.

In order to do **CMP** on the rule RecvInvAck5, LB-CMP instantiates all the learned invariants into some concrete formulas and call gStrEn to find the invariants to strengthen the guard of the rule, and those listed as follows are related with the rule $RecvInvAck5[3]$.

```
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan2[1].Cmds != Inv
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan2[2].Cmds != Inv
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan3[1].Cmds != InvAck
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan3[2].Cmds != InvAck
Chan3[3].Cmd = InvAck & ExGntd = true -> Cache[1].State = I
Chan3[3].Cmd = InvAck & ExGntd = true -> Cache[2].State = I
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan2[1].Cmd = Empty
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan2[2].Cmd = Empty
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan3[1].Cmd = Empty
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan3[2].Cmd = Empty
Chan3[3].Cmd = InvAck & ExGntd = true -> Chan3[3].Data = AuxData
Chan3[3].Cmd = InvAck & ExGntd = true -> InvSet[1] = false
Chan3[3].Cmd = InvAck & ExGntd = true -> InvSet[2] = false
Chan3[3].Cmd = InvAck & ExGntd = true -> ShrSet[1] = false
Chan3[3].Cmd = InvAck & ExGntd = true -> ShrSet[2] = false
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan2[1].Cmds != Inv
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan2[2].Cmds != Inv
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan3[1].Cmds != InvAck
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan3[2].Cmds != InvAck
ExGntd = true & Chan3[3].Cmd = InvAck -> Cache[1].State = I
ExGntd = true & Chan3[3].Cmd = InvAck -> Cache[2].State = I
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan2[1].Cmd = Empty
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan2[2].Cmd = Empty
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan3[1].Cmd = Empty
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan3[2].Cmd = Empty
ExGntd = true & Chan3[3].Cmd = InvAck -> Chan3[3].Data = AuxData
ExGntd = true & Chan3[3].Cmd = InvAck -> InvSet[1] = false
ExGntd = true & Chan3[3].Cmd = InvAck -> InvSet[2] = false
ExGntd = true & Chan3[3].Cmd = InvAck -> ShrSet[1] = false
ExGntd = true & Chan3[3].Cmd = InvAck -> ShrSet[2] = false
Chan3[3].Cmd = InvAck -> Cache[1].States != E
Chan3[3].Cmd = InvAck -> Cache[2].States != E
Chan3[3].Cmd = InvAck -> Cache[3].States != E
Chan3[3].Cmd = InvAck -> Cache[3].States != S
Chan3[3].Cmd = InvAck -> Chan2[1].Cmds != GntE
Chan3[3].Cmd = InvAck -> Chan2[2].Cmds != GntE
Chan3[3].Cmd = InvAck -> Chan2[3].Cmds != GntE
Chan3[3].Cmd = InvAck -> Chan2[3].Cmds != GntS
Chan3[3].Cmd = InvAck -> Chan2[3].Cmds != Inv
Chan3[3].Cmd = InvAck -> CurCmd != Empty
Chan3[3].Cmd = InvAck -> Cache[3].State = I
Chan3[3].Cmd = InvAck -> Chan2[3].Cmd = Empty
Chan3[3].Cmd = InvAck -> InvSet[3] = false
Chan3[3].Cmd = InvAck -> ShrSet[3] = true
ExGntd = true -> Cache[1].States != S
ExGntd = true -> Cache[2].States != S
ExGntd = true -> Cache[3].States != S
ExGntd = true -> Chan2[1].Cmds != GntS
ExGntd = true -> Chan2[2].Cmds != GntS
ExGntd = true -> Chan2[3].Cmds != GntS
```

In fact, the above concrete invariants are the instance of the following parameterized invariants by instantiating with actual parameters.

```
Chan3[i].Cmd = InvAck & ExGntd = true -> Chan2[j].Cmd != Inv
Chan3[i].Cmd = InvAck & ExGntd = true -> Chan3[j].Cmd != InvAck
Chan3[i].Cmd = InvAck & ExGntd = true -> Chan2[j].Cmd = Empty
Chan3[i].Cmd = InvAck & ExGntd = true -> Chan3[j].Cmd = Empty
Chan3[i].Cmd = InvAck & ExGntd = true -> Chan3[i].Data = AuxData
Chan3[i].Cmd = InvAck & ExGntd = true -> InvSet[j] = false
Chan3[i].Cmd = InvAck & ExGntd = true -> ShrSet[j] = false
ExGntd = true & Chan3[i].Cmd = InvAck -> Chan2[j].Cmd != Inv
ExGntd = true & Chan3[i].Cmd = InvAck -> Chan3[j].Cmd != InvAck
Chan3[i].Cmd = InvAck -> Cache[j].State != E
Chan3[i].Cmd = InvAck -> Cache[i].State != S
Chan3[i].Cmd = InvAck -> Cache[i].State = I
Chan3[i].Cmd = InvAck -> Chan2[i].Cmd != GntE
Chan3[i].Cmd = InvAck -> Chan2[i].Cmd != GntS
Chan3[i].Cmd = InvAck -> Chan2[i].Cmd != Inv
Chan3[i].Cmd = InvAck -> CurCmd != Empty
Chan3[i].Cmd = InvAck -> Chan2[i].Cmd = Empty
Chan3[i].Cmd = InvAck -> InvSet[i] = false
Chan3[i].Cmd = InvAck -> ShrSet[i] = true
ExGntd = true -> Cache[i].State != S
ExGntd = true -> Chan2[j].Cmd != GntS
```

Using the drawCons function, LB-CMP draws the following conclusions:

```
forall i : NODE do Chan2[i].Cmd != Inv endforall
forall i : NODE do Chan3[i].Cmd != InvAck endforall
forall i : NODE do Cache[i].State = I endforall
forall i : NODE do Chan2[i].Cmd = Empty endforall
forall i : NODE do Chan3[i].Cmd = Empty endforall
Chan3[3].Data = AuxData
forall i : NODE do InvSet[i] = false endforall
forall i : NODE do ShrSet[i] = false endforall
forall j : NODE do Chan2[j].Cmd != Inv endforall
forall j : NODE do Chan3[j].Cmd != InvAck endforall
forall j : NODE do Cache[j].State = I endforall
forall j : NODE do Chan2[j].Cmd = Empty endforall
forall j : NODE do Chan3[j].Cmd = Empty endforall
forall j : NODE do Cache[j].State != E endforall
Cache[3].States!= E
Cache[3].States!= S
forall j : NODE do Chan2[j].Cmd != GntE endforall
Chan2[3].Cmds!= GntE
Chan2[3].Cmds!= GntS
Chan2[3].Cmds!= Inv
CurCmd != Empty
Cache[3].State = I
Chan2[3].Cmd = Empty
InvSet[3] = false
ShrSet[3] = true
forall j : NODE do Cache[j].State != S endforall
forall i : NODE do Chan2[i].Cmd != GntS endforall
```

For instance, from the invariant $ExGntd = true$ & $Chan3[3].Cmd = InvAck \rightarrow Chan2[1].Cmd! = Inv$, by the definition of drawCon, because Other does not occur in $Chan2[1].Cmd! = Inv$, so we generalize it into $forall i : NODE do Chan2[i].Cmd! = Inv endforall. Chan3[3].Data = AuxData$ is also added, and is used for the local variable assignment.

After the first round of guard strengthening, the function gStrEn is recursively called by removing those invariants which have been used, to find new invariants to strengthen the guard. In this case for rule $RecvInvAck5[3]$, gStrEn is called more than twice.

After guard strengthening, the abstract is called to refine and abstract the rule. Firstly, assignments to non-Other nodes such as $ExGntd := false$ are found, then a dictionary is set up according to the atomic predicates in the cubic of the guard, where a key $Chan3[3].Data$ maps $AuxData$; then the $refineAct$ is called by keeping $ExGntd := false$, and do the local variable replacement for the local variable $Chan3[3].Data$ with $AuxData$. At last, all the atomic predicates on Other such as $InvSet[3] = false$ and $Chan3[3].Cmd = InvAck$ are removed.

```
rule "ABS_RecvReqE11"
CurCmd = Empty &
forall i : NODE do
Chan3[i].Cmd = Empty & Chan3[i].Cmd != InvAck endforall & Chan2[i].Cmd != Inv
end ==>
begin
CurCmd := ReqE; CurPtr := Other;
for j : NODE do
InvSet[j] := ShrSet[j];
endfor;
endrule;
```

In the ABS_German protocol, $M = 2$, and $Other = 3$. The invariant inv we want to verify after doing **CMP** is for all nodes, the conjunction of the set of all the invariants F used in the **CMP** process including the original property. Namely $inv = \bigwedge F$. Next, let us explain how the ABS_German- protocol which is returned by our cmp function, satisfies the antecedents defined in Lemma 1.

- The initial statement (startstate) in ABS_German protocol is almost similar with that in original German statement except that the former only assign I to a node $i \leq M$ but the latter to a node $j \leq N$. Obviously the condition 1 is satisfied;
- For a rule r like , it only changes its own local state. The rule will be simply abstracted as $true \Rightarrow SKIP$. Such a rule will do nothing to change protocol's state. Conditions 2-6 will be satisfied.
- For a rule like ABS_RecvInvAck5, each item in its guard cubic is either from the guard of RecvInvAck5(Other) or from a conclusion implied by the conjunction of its guard and inv, thus guard of ABS_RecvInvAck5 is implied by RecvInvAck5(Other), thus 2 is satisfied. In the action of ABS_RecvInvAck5, only assignments to global variables such as $ExGntd := false$; and $MemData := AuxData$, the former is the same with the one in RecvInvAck5, the latter is generated by local variable replacement. From the procedure of local variable replacement, we have $(Chan3[3].Data = AuxData)$, which is also implied by RecvInvAck5(Other). Thus, conditions 4 and 5 are also satisfied.
- For a rule like ABS_RecvReqE11, there are two kinds of actions: assignments to global variables, and assignments to all the non-Other local variables,

which is indexed by an index $i \leq M$ ($i : NODE$). They are all copied from the original assignment in RecvReqE11. Thus conditions 2-6 are satisfied.

– After we use Murphi to do model checking for the ABS_German protocol, all the auxiliary invariants hold. Thus 7 holds.

From the above discussion, by Lemma 1, we can say the original German protocol satisfies *inv*, which implies the mutual exclusion property and data-consistency property.