L-CMP: An Automatic Learning-Based Parameterized Verification Tool

Jialun Cao
The State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences & University of Chinese Academy of Sciences
Beijing, China
caojl@ios.ac.cn

Yongjian Li*
The State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
lyj238@ios.ac.cn

Jun Pang
Faculty of Science, Technology and Communication & Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
Esch-sur-Alzette, Luxembourg
jun.pang@uni.lu

ABSTRACT

This demo introduces L-CMP, an automatic learning-based parameterized verification tool. It can verify parameterized protocols by combining machine learning and model checking techniques. Given a parameterized protocol, L-CMP learns a set of auxiliary invariants and implements verification of the protocol using the invariants automatically. In particular, the learned auxiliary invariants are straightforward and readable. The experimental results show that L-CMP can successfully verify a number of cache coherence protocols, including the industrial-scale *FLASH* protocol. The video presentation of L-CMP is available at https://youtu.be/6Dl2HiiiS4E, and the source code can be downloaded at https://github.com/ArabelaTso/Learning-Based-ParaVerifer.

CCS CONCEPTS

• **Theory of computation → Verification by model checking**;

KEYWORDS

Parameterized verification, model checking, association rule learning, machine learning

ACM Reference Format:

Jialun Cao, Yongjian Li, and Jun Pang. 2018. L-CMP: An Automatic Learning-Based Parameterized Verification Tool. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240487

1 INTRODUCTION

Parameterized concurrent systems, in particular, cache coherence protocols, exist in many practical applications [12]. Verifying such systems has attracted considerable academic interests due to its practical importance [13]. A parameterized protocol $\mathcal{P}(N)$ contains N homogeneous nodes (see sub-figure (a) in Figure 1), sometimes also contains a heterogeneous node (e.g., *FLASH* protocol contains a 'Home' node, see sub-figure (c) in Figure 1). The target is to verify that the properties are satisfied for arbitrary sizes of instances. Although the correctness of fixed number of instances can be proved, their correctness cannot imply the correctness of protocols of arbitrary sizes. This problem has been proved to be undecidable [2].

To address this problem, many approaches have been proposed [3, 5–7, 9, 14–17, 19]. Among them, "parameterized abstraction and guard strengthening" method, also known as CMP [6, 14], has been widely applied to verify large-scaled and industrial protocols, including Intel's Chipset and *FLASH* protocols [18]. The main idea of CMP is to construct an abstract protocol $\mathcal{AP}(m)$ (m is usually 2 or 3). $\mathcal{AP}$ contains m normal nodes plus an abstracted node *Other*, which simulates the behavior of the rest nodes. As we can see from Figure 1, in sub-figure (b), m is 2 because only homogeneous nodes in protocol (a), while in sub-figure sub-figure (d), m is set to be 3 because there is a heterogeneous node *Home* in sub-figure (c). In general, $\mathcal{AP}$ may fail to satisfy the desired properties because it is too 'permissive' [6], so researchers need to come up (manually) with a set of auxiliary invariant to restrict its behavior. However, to construct auxiliary invariants manually is often error-prone, especially when the protocol description is large and complex [6].

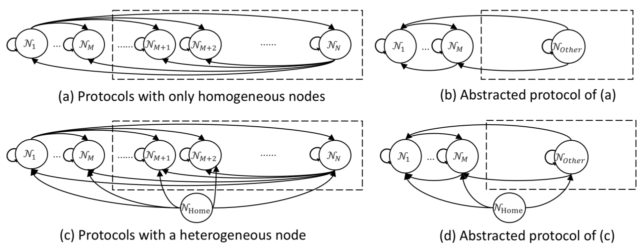

$\mathcal{N}_1$
$\mathcal{N}_M$
$\mathcal{N}_{M+1}$
$\mathcal{N}_{M+2}$
$\mathcal{N}_N$
$\mathcal{N}_{Other}$
$\mathcal{N}_{Home}$
(a) Protocols with only homogeneous nodes
(b) Abstracted protocol of (a)
(c) Protocols with a heterogeneous node
(d) Abstracted protocol of (c)

Figure 1: Different types of parameterized protocols and corresponding abstracted protocols after CMP

In this paper, we propose L-CMP, which can automatically learn auxiliary invariants, abstract and strengthen the parameterized protocols in a unified framework. With L-CMP, a set of auxiliary invariants can be learned from the reachable state sets of the instances

*Corresponding author

of protocols, and be applied to verify the protocol automatically. The soundness of L-CMP is guaranteed in [8].

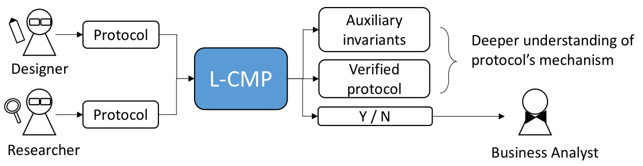


Figure 2: The envisioned users

A distinguished novelty of our tool lies in two aspects. Firstly, the combination of association rule learning and parameterized verification automates the verification process, and expands the application area of machine learning algorithms. It may also shed some light on the further combination of these two areas. Secondly, the invariants it obtains can not only help to automate the verification, but also provide deep insights for researchers to better understand the mechanism of protocols because the form of these invariants are straightforward and understandable. Therefore, our envisioned users are protocol designers and researchers who want to gain better understanding and insights of protocols, and business analysts who want to check the correctness of practical protocols (see Figure 2).

2 RELATED WORKS

There have been many studies who tried to address the problem of parameterized verification with the aid of computer. Arons *et al.* proposed the concept of "invisible invariants", which are computed in a finite system instance to aid inductive invariant checking [3]. Conchon *et al.* came up with the BRAB algorithm which is implemented in an SMT-based model checker. It computes over-approximations of backward reachable states that are checked to be unreachable in the parameterized system [7]. Li *et al.* proposed a method to automatically generate auxiliary invariants from a small reference instance of protocols and construct a parameterized formal proof in the theorem prover Isabelle [12]. Compared with these works, L-CMP provides a simpler way to verify parameterized protocols, and the auxiliary invariants it obtains are more understandable and straightforward.

3 AN OVERVIEW OF L-CMP

The architecture of L-CMP is shown in Figure 3. We can see that there are two main phases in the framework. The first phase is regarded as the *Learning* phase. It receives a parameterized protocol written in the input language of the model checker Murphi and returns a set of auxiliary invariants. This phase consists of three parts. Preprocessor utilizes Murphi to enumerate the reachable set of the instance of protocols and transforms it into a dataset consisting of numeric vectors. Then *Learner* learns association rules from the constructed dataset. Note that the learned association rules are not necessarily invariants, so *Selector* is executed with the help of Murphi to select invariants. After selection, what remained is regarded as auxiliary invariants and will be used in the second phase.

The second phase is an *advanced CMP* phase. Unlike CMP, whose strengthening process comes after abstraction, the advanced CMP carries out strengthening process before abstraction so that more predicates can be added into guard of rules. Phase two also consists of three main steps. *Strengthener* strengthens the guard of each rule using auxiliary invariants. It is noteworthy that not all auxiliary invariants will be used in this step, and once some of them are used, they will be recorded and verified in the final step. Next, *Abstractor* abstracts local variables in the rules. After this step, an abstracted protocol is generated and it will be subjected to Murphi together with the used auxiliary invariants. If it passes the model checking, then L-CMP comes outputs the result. Otherwise, parameters in *Learner* will be adapted and the next round of execution will start.

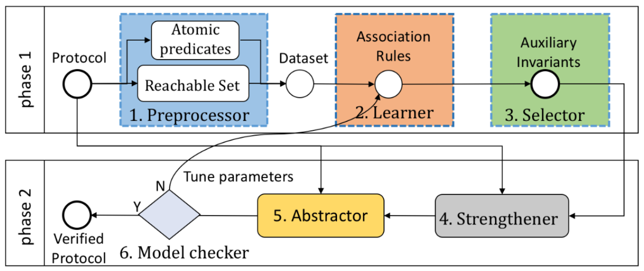


Figure 3: The architecture of L-CMP

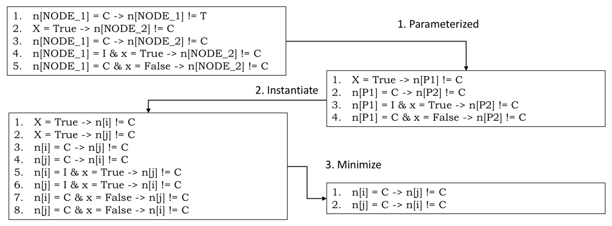
4 IMPLEMENTATION DETAILS

4.1 Phase One

Given a parameterized protocol in the Murphi language, the first step is to generate the reachable set of its instance. We use Murphi to compute the reachable set, which can write all reachable states at least once. In general, we assume that the protocol contains only homogeneous nodes so the size of instance is set to be 2. If there is a heterogeneous node in the protocol (e.g., the *FLASH* protocol), then the size of instance will be set to be 3, including two normal nodes and one *Home* node. Note that there are two possible forms to represent parameters: digits (i.e., 1, 2) or labeled type names (i.e., NODE_1, NODE_2, where 'NODE' is a user-defined data type). The first form is the ordinary expression, and the second form is used when the protocol applies symmetry reduction to reduce the searching space. L-CMP will record the form of parameters and use it as identifiers.

Then, the reachable set is transformed into a new dataset. We extract atomic predicates from the guard of rules and properties. This step is crucial because without it the following learning process may not be able to obtain sufficient auxiliary invariants. We also build a map to record all the possible values for each variable. According to the atomic predicates, the reachable set can be transformed into state vectors which contain only predicates. To be more specific, if the predicate holds at one state, then we can add the original predicate to the state. Otherwise, the negative form of the predicate will be added. Note that because of symmetry reduction, the value of some variables could be 'Undefined', meaning that the value of the variable is unknown. In this case, whether the predicate holds in that state is unknown, so the predicate will be deleted from the corresponding state. After this step, we acquire a new dataset consisting of records, where each record represents a reachable state in the reachable set, and the length of records are not necessarily the same.

The association rule learning is applied to the dataset in the third step. Association rules are strong relations between items in large databases. Assume that a database contains records, each of which includes several items, then the learning result will be the relations in the database, in the form of $\psi \longrightarrow \phi$ where ϕ and ψ are sets of items. In our case, the records in the dataset are already constructed, while items of association rule learning are correspondent to the predicates in the records. To implement association rule learning, there are several algorithms such as Eclat and FP-growth. In this paper, we apply the Apriori [1] algorithm because it is more convenient to constrain the size of frequent set. The frequent set is a set of items that frequently appears in the dataset. We set the size of the frequent set to be at most K, where K is a parameter that is initially set to be 3. It may be fixed according to the subsequent steps. Note that the minimum support and minimum confidence in Apriori are also parameters, which are set to be 0.0 and 1.0, respectively.



```
1.  n[NODE_1] = C -> n[NODE_1] != T
2.  X = True -> n[NODE_2] != C
3.  n[NODE_1] = C -> n[NODE_2] != C
4.  n[NODE_1] = I & x = True -> n[NODE_2] != C
5.  n[NODE_1] = C & x = False -> n[NODE_2] != C
```

```
1.  X = True -> n[P1] != C
2.  n[P1] = C -> n[P2] != C
3.  n[P1] = I & x = True -> n[P2] != C
4.  n[P1] = C & x = False -> n[P2] != C
```

```
1.  X = True -> n[i] != C
2.  X = True -> n[j] != C
3.  n[i] = C -> n[j] != C
4.  n[j] = C -> n[i] != C
5.  n[i] = I & x = True -> n[j] != C
6.  n[j] = I & x = True -> n[i] != C
7.  n[i] = C & x = False -> n[j] != C
8.  n[j] = C & x = False -> n[i] != C
```

```
1.  n[i] = C -> n[j] != C
2.  n[j] = C -> n[i] != C
```

Figure 4: The process of minimizing association rules

The number of learned association rules is generally large, so we need a step to minimize it (see Figure 4). There are two possible forms of parameters as mentioned above, so we need to replace them with a unified form. We replace parameters with $P1$, $P2$ in the order of appearance. This is why the 'NODE_2' of second statement in top left of Figure 4 will be replaced by $P1$. Moreover, those symmetrical association rules will be unified as one. Note that some of association rules are axiomatic but useless (see the first association rule in the top of Figure 4), so they will be removed. Then the parameterized association rules will be instantiated by i and j, respectively. Finally, we build a dictionary to record all the left sides of association rules which have same right side. For each right side, we build an index to represent all the left sides. Note that each left side is a set. If one left side is the subset of another, then the index of that larger set will be removed. For example, if association rules $p \longrightarrow r$ and $p \lor q \longrightarrow r$ are obtained, then after this step, only $p \longrightarrow r$ will remain.

Selector, as its name implies, selects auxiliary invariants from the association rules. This step needs the help of Murphi. Although it is impossible to verify all the association rules for protocols with arbitrary sizes, we can check them in several small instances. To realize it, we construct a translator which translates the association rules into the Murphi language, so that we check them in Murphi. Then, these translated association rules together with the original instance from which the reachable set calculated are subjected to

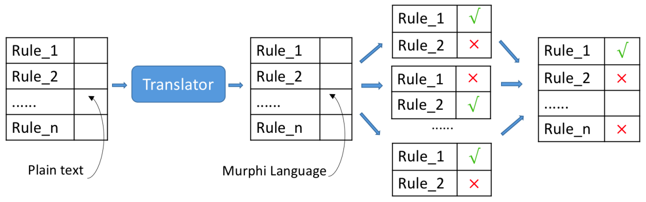


Figure 5: The process of selecting auxiliary invariants

Murphi. Whenever an association rule fails to pass the model checking step, L-CMP removes it and continues to check the rest. When all the remaining association rules pass the verification, L-CMP will increase the size of instance and repeat the above process. This selection step proceeds until no association rules can be removed or the searching space is getting too large. Note that we apply multiprocessing to accelerate this step (as shown in Figure 5), each process checks a part of association rules, and returns whether they pass the verification. Then the main processor collects all the result and removes the failed ones from the association rule set. Finally, the remaining association rules are regarded as auxiliary invariants.

4.2 Phase Two

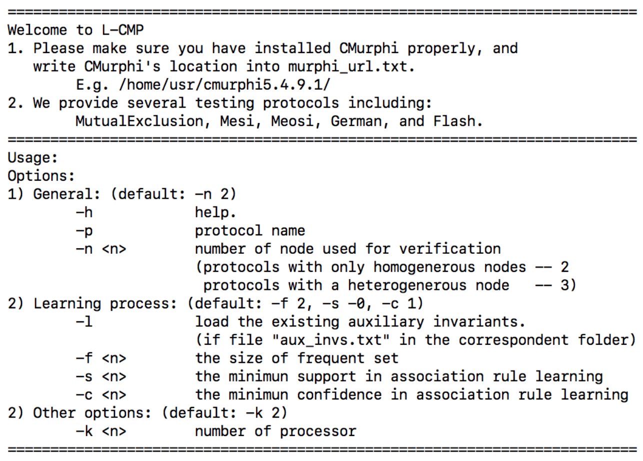
Before going into details, we need to elaborate the structure of protocol rules. A rule consists of two main parts: **guard** and **action**. If the predicates in guard are satisfied, the statements in action can be executed. As we can see, the guard is a conjunction of predicates, while the auxiliary invariants are in form of $\phi \longrightarrow \psi$, where ϕ, ψ are sets of predicates. Thus, it is straightforward to add the conclusion (i.e., ψ) of auxiliary invariants to the guard of rules if its antecedent (i.e., ϕ) is a subset of the guard. This strengthening step iterates until no more predicates can be added into the guard.

```
ruleset {parameters definition} do      -- for example, i : NODE
rule "rule_name"
     guard part      -- conjunction of predicates
==>
     action part      -- a set of statements
endrule; endruleset;
```

Phase 2 starts at partitioning the rules into four parts: definition, rule name, guard and action. Then we parse the guard into predicates and execute the above-mentioned strengthening process. The auxiliary invariants which are used to strengthen the guard will be recorded. Besides, a dictionary is exploited to map the antecedents of used auxiliary invariants to the corresponding conclusions. Next, the abstraction step is implemented by removing the predicates and statements relating to parameters in the definition part. For example, if a rule has two parameters i and j, at first, we remove those related to parameter i and generate the first abstracted rule, and do the same for parameter j. Further, those related to either i or j are all removed and a third new rule is generated. Hence, if a rule has one parameter, then after strengthening and abstraction steps, it will generate at most one new rule. While if it has two parameters, the answer will be three. It is noteworthy that if the right side of a statement is about to be abstracted, then the dictionary we build previously will be used to check whether the right part is a key. If it is a key, then it will be replaced by its corresponding value. This process prevents the loss of data during abstraction.

5 TOOL USAGE

Here is an example showing how L-CMP works. After downloading L-CMP and installing Murphi and Python, it is important to inlcude the path to Murphi into the file 'murphi_url.txt'. We provide several protocols in L-CMP. The help document is illustrated in Figure 6. To verify *FLASH*, for example, we can simply type 'python3 main.py -p flash -n 3' and the verification will be executed automatically. Because *FLASH* is fairly large, so it takes some time to verify. After it finishes, the result will be printed out and several important files such as verified protocols, association rules, auxiliary invariants and other backup files will be saved in the corresponding folder.



```
=========================================================================
Welcome to L-CMP
1. Please make sure you have installed CMurphi properly, and
   write CMurphi's location into murphi_url.txt.
        E.g. /home/usr/cmurphi5.4.9.1/
2. We provide several testing protocols including:
        MutualExclusion, Mesi, Meosi, German, and Flash.
=========================================================================
Usage:
Options:
1) General: (default: -n 2)
        -h              help.
        -p              protocol name
        -n <n>          number of node used for verification
                        (protocols with only homogenerous nodes -- 2
                         protocols with a heterogenerous node   -- 3)
2) Learning process: (default: -f 2, -s -0, -c 1)
        -l              load the existing auxiliary invariants.
                        (if file "aux_invs.txt" in the correspondent folder)
        -f <n>          the size of frequent set
        -s <n>          the minimun support in association rule learning
        -c <n>          the minimun confidence in association rule learning
2) Other options: (default: -k 2)
        -k <n>          number of processor
=========================================================================
```

Figure 6: The help document of L-CMP

6 EXPERIMENTAL EVALUATION

We applied L-CMP to several typical parameterized protocols. Among them, *MOESI, MESI, Mutual Exclusion* (abbrv. MutualEx), *Mutual Exclusion with data* (abbrv. Mutdata) are small-scale protocols, while *German* and *FLASH* protocols are more complicated[1]. The size of instance for most protocols is set to be 2, while for *FLASH*, it is set to be 3.

Table 1: Experiment results

protName	states	# assoRules	# auxInvs	# usedF	Time (s)	result
MOESI	23	736	20	5	24.744	✓
MESI	8	144	16	5	24.412	✓
MutualEx	12	656	12	3	89.869	✓
Mutdata	88	540	12	6	19.703	✓
German	852	21202	448	8	85.418	✓
Flash	1350226	358710	1636	327	41371.023	✓

The experimental results for each of these protocols are summarized and presented in Table 1. Each record includes protocol name, size of reachable sets, number of association rules, number of auxiliary invariants, number of used auxiliary invariants, running time, and result. We can see from the table that reachable sets of the first four protocols are rather small, followed by *German*, while the reachable set of *FLASH* is enormous comparing with the previous ones. As for running time, almost the first five protocols can be verified within about one minute, whereas *FLASH* takes about 11 hours. Most of the time is spent on selecting auxiliary invariants from association rules. This process can be accelerated by allocating more processors to L-CMP. If we allocate 16 processors to L-CMP, the total running time for verifying *FLASH* is approximate 6 hours.

[1]The version of *German* and *FLASH* protocols we use are the same as those used in Chou *et al.*'s work [6]

7 CONCLUDING REMARKS

In this paper, we present an easy-to-use tool, L-CMP, which can verify parameterized protocols automatically. It provides straightforward and understandable invariants. Besides, it also has solid theoretical background. The verification results of typical benchmarks are reported in detail. Experiments demonstrate both feasibility and effectiveness of L-CMP. In the future, we plan to extend the ability of L-CMP to verify more general safety and liveness properties.

ACKNOWLEDGEMENTS

The authors are partially supported by grant 61672503 from National Natural Science Foundation and grant 2017YFB0801900 from the National Key Research and Development Program in China.

REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami. 1993. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, Vol. 22, 207–216.
[2] K. R. Apt and D. C. Kozen. 1986. Limits for automatic verification of finite-state concurrent systems. *Information Processsing Letters* 22, (6), 307–309.
[3] A. Pnueli, S. Ruah, Y. Xu, and L. Zuck. 2001. Parameterized verification with automatically computed inductive assertions?. In *CAV*. Springer, LNCS 2102, 221–234.
[4] K. Baukus *et al.* 2002. Parameterized verification of a cache coherence protocol: Safety and liveness. In *VMCAI*. Springer, LNCS 2294, 317–330.
[5] X. Chen and G. Gopalakrishnan. 2006. *A General Compositional Approach to Verifying Hierarchical Cache Coherence Protocols.* Technical Report. UUCS-06-014, School of Computing, University of Utah.
[6] C.-T. Chou, P. K Mannava, and S. Park. 2004. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*. Springer, LNCS 3312, 382–398.
[7] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaıdi. 2012. Cubicle: A parallel SMT-based model checker for parameterized systems. In *CAV*. Springer, LNCS 7358, 718–724.
[8] S. Krstic. 2005. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite State Systems*.
[9] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. 2013. Invariants for finite instances and beyond. In *FMCAD, 2013*. IEEE, 61–68.
[10] E. A. Emerson and V. Kahlon. 2003. Exact and efficient verification of parameterized cache coherence protocols. In *CHDVM*. Springer, LNCS 2860, 247–262.
[11] S. K. Lahiri and R. E. Bryant. 2004. Constructing quantified invariants via predicate abstraction. In *VMCAI*. Springer, LNCS 2937, 267–281.
[12] Y. Li, K. Duan, Y. Lv, J. Pang, and S. Cai. 2016. A novel approach to parameterized verification of cache coherence protocols. In *ICCD*. IEEE, 560–567.
[13] Y. Lv, H. Lin, and H. Pan. 2007. Computing invariants for parameter abstraction. In *MEMOCODE*. IEEE, 29–38.
[14] K. L. McMillan. 2001. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME*. Springer, LNCS 2144, 179–195.
[15] S. Pandav, K. Slind, and G. Gopalakrishnan. 2005. Counterexample guided invariant discovery for parameterized cache coherence verification. In *CHARME*. Springer, LNCS 3725, 317–331.
[16] A. Pnueli, S. Ruah, and L. Zuck. 2001. Automatic deductive verification with invisible invariants. In *TACAS*. Springer, LNCS 2013, 82–97.
[17] A. Pnueli and E. Shahar. 1996. A platform for combining deductive with algorithmic verification. In *CAV*. Springer, LNCS 1102, 184–195.
[18] M. Talupur and M. R. Tuttle. 2008. Going with the flow: Parameterized verification using message flows. In *FMCAD*. IEEE, 1-8.
[19] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. 2001. A technique for invariant generation. In *TACAS*. Springer, LNCS 2013, 113–127.