Accelerated Verification of Parametric Protocols with Decision Trees

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

Taifeng Cao
School of Software Engineering
Beijing Jiaotong University
Beijing, China

David N. Jansen
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing, China

Jun Pang
Faculty of Science, Technology and Medicine
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
Esch-sur-Alzette, Luxembourg

Xiaotao Wei
School of Software Engineering
Beijing Jiaotong University
Beijing, China

***Abstract*—Within a framework for verifying parametric network protocols through induction, one needs to find invariants based on a protocol instance of a small number of nodes. In this paper, we propose a new approach to accelerate parameterized verification by adopting decision trees to represent the state space of a protocol instance. Such trees can be considered as a knowledge base that summarizes all behaviors of the protocol instance. With this knowledge base, we are able to efficiently construct an oracle to effectively assess candidates of invariants of the protocol, which are suggested by an invariant finder. With the discovered invariants, a formal proof for the correctness of the protocol can be derived in the framework after proper generalization. The effectiveness of our method is demonstrated by experiments with typical benchmarks.**

***Index Terms*—Formal methods, parameterized verification, machine learning, decision trees, cache coherence protocols**

I. Introduction

Parametric protocols are often verified by enumerating all reachable protocol states and checking the desired properties. This reachability analysis can be done through either explicit state enumeration [1], [2], [3] or using symbolic methods [4], [5]. Both approaches suffer from the state space explosion problem – the size of the reachable state space grows exponentially with respect to the protocol size, and eventually it exhausts all available computer memory for verification. However, according to earlier experiences in parameterized verification, explicit state enumeration has outperformed the symbolic approach, thanks to symmetry reduction techniques and more advanced compact hash tables [6].

Explicit state enumeration is suitable for model checking bounded protocol models, but it cannot produce useful information like invariants, in the form of predicate logic formulas. However, for formal verification of parametric protocols, an inductive proof framework (e.g., see [7]) usually requires an set of inductive invariants, for which it needs to find auxiliary invariants. Similarly, in the classic CMP method [6], auxiliary invariants are also needed to construct an abstract protocol model for a CEGAR-style refinement procedure.[1]

In order to bridge the gap between explicit-state model checking to symbolic invariant generation, we propose a decision-tree based machine learning approach. Within our approach, we first collect reachable states (abbrev. RS) of an instance of the parametric protocol under verification and record our knowledge about RS as a decision tree. Then we construct candidates of invariants by adopting an earlier algorithm as proposed in the literature [7], [8], [9], and filter true invariants according to the knowledge obtained in the first step. After proper generalization, the selected invariants can be used to construct a formal proof for parameterized verification.

Our contributions in this work are summarized as follows:

- We propose a decision-tree based learning technique to sum up symbolic formulas to approximate the behaviors of a protocol instance. In particular, we use explicit-state based model checking to get the reachable state set of a protocol instance with the help of symmetry reduction; the state space is stored in a table called a *data sheet*. We randomly select a variable with more than one value in the data sheet as target variable; we extend the classical decision-tree algorithm to classify the selected target variable based on other variables from the data sheet. The paths from the root to a leaf node cover the reachable state set. If an invariant candidate contradicts a formula corresponding to such a path, then it will be refuted; otherwise we regard it as a true invariant.
- We develop a learning paradigm to synthesize the set of auxiliary invariants to accelerate the verification of parametric cache coherence protocols. Invariant candidates will be conjectured by an invariant finder, and sent to a server, which constructs the aforementioned decision tree to decide whether such a candidate is indeed an invariant.

[1] CEGAR: counterexample guided abstraction refinement.

If so, then the invariant and proof dependency relation will be recorded and used to construct a formal proof for parameterized protocol verification.
- We implement the extended decision-tree based learning approach as a teacher to classify the data sheet generated from the RS of the protocol. We integrate the teacher into the framework of paraVerifier [7], [8], [9] to automatically find invariants and to prove parametric cache coherence protocols. We conduct experiments successfully on typical benchmarks.

Our work is the first one to apply decision trees to parameterized verification of cache coherence protocols. Experimental results show that our method can accelerate the finding of true invariants for parameterized verification with less memory and computation time.

Paper structure. The rest of this paper is organized as follows. In Section II, we present our research problem of parameterized verification, introduce some preliminaries used in this paper and discuss the main challenge of finding auxiliary invariants in parameterized verification. Our approach to accelerating parameterized verification by utilising decision-tree for learning invariant candidates is presented in Section III. With experimental results on typical benchmarks, we demonstrate that our new approach can speed up the verification of parametric cache coherence protocols in Section IV, in terms of computational time and memory cost. This is then followed by conclusions and future work in Section VI.

II. Parameterized Verification

A. Parametric communication protocols

In this section, we give a high-level description of the problem we want to tackle and introduce a number of basic notations. We work in the context of a *communicating network* of computers; we call them *nodes*. Every node runs the same protocol. We explicitly consider the number of nodes to be N; this makes the system *parametric*. We refer to [7] for the details of the specification formalism.

Example 1 *We explain the formalism by a simple mutual exclusion protocol with N nodes. Global states are described using a finite set V of state variables, which range over a finite set $\mathbb{D}$. We use the global Boolean variable x to indicate that the critical resource is available or not. n is an array containing local variables: each $n[i]$ describes the state of node i, which is one of* $\mathsf{I}(\mathsf{dle})$, $\mathsf{T}(\mathsf{rying})$, $\mathsf{C}(\mathsf{ritical})$, *and* $\mathsf{E}(\mathsf{xiting})$.

Our simple mutual exclusion protocol has the following guarded commands (also called protocol rules):

$$\begin{aligned}
\mathsf{try}(j) &\equiv \quad n[j] = \mathsf{I} \;\rhd\; n[j] := \mathsf{T}\\
\mathsf{crit}(j) &\equiv x \wedge n[j] = \mathsf{T} \rhd \{n[j] := \mathsf{C}, x := \mathsf{false}\}\\
\mathsf{exit}(j) &\equiv \quad n[j] = \mathsf{C} \;\rhd\; n[j] := \mathsf{E}\\
\mathsf{idle}(j) &\equiv \quad n[j] = \mathsf{E} \rhd \{n[j] := \mathsf{I}, x := \mathsf{true}\}.
\end{aligned}$$

Each guarded command has the form $g \rhd A$, where the guard g is a predicate over V and A is a parallel assignment to variables $v_\ell := e_\ell$.

The protocol itself is defined by $mutexini_N$, the initial state of the protocol, and $mutexrules_N$, the set of protocol rules:

$$\begin{aligned}
mutexini_N &\equiv x \wedge \bigwedge_{i=1}^{N} n[i] = \mathsf{I}\\
mutexrules_N &= \{\mathsf{try}(j)\ ,\ \mathsf{crit}(j)\ ,\ \mathsf{exit}(j)\ ,\\
&\qquad \mathsf{idle}(j)\ \mid\ j = 1, \ldots, N\}\\
\mathcal{M}utual\mathcal{E}x_N &= (mutexini_N, mutexrules_N)
\end{aligned}$$

and it should satisfy the following invariant requirement:

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

The property $\mathsf{mutualInv}(i_1, i_2)$ *describes that $n[i_1]$ and $n[i_2]$ cannot be in a critical state at the same time. It should be understood that $i_1 \neq i_2$. Note that we normally use i to denote node identities in formulas and j to denote node identities in guarded commands.*

The research problem. In general, we assume given a network of N nodes; its (global) state space is described by the variables in V, and every node runs a communication protocol described by $\mathcal{P} = (I, R)$. We aim to prove some invariant property req of the protocol, which can be specified using a (parametric) formula without temporal modalities. We write it in the form $\neg \bigwedge_{n=1}^{k} f_n$, where each f_n is a literal $e_1 = e_2$ or $e_1 \neq e_2$. Thus, the question we want to answer is:

> *Does the invariant property req hold in every reachable state of the nodes running $\mathcal{P}$?*

B. An automatic proving approach

Recently, an automatic proving approach [7], [8], [9] is proposed by Li *et al.* to address the above research problem, and their central idea is to guide a theorem prover by providing *inductive invariant* candidates. In their approach, an inductive invariant is an invariant that is satisfied by the initial state and preserved by each guarded command of the protocol. The protocol satisfies the requirement req if there is an inductive invariant that implies req.

This approach requires to add auxiliary invariants to make the (conjunction of all) invariants inductive. The following steps are used to find auxiliary invariants candidates:

1) Concretize the requirement req into req^{c}. A single injective mapping from parameters to natural numbers is normally sufficient; as the node identities are symmetric.
2) Concretize a guarded command (from r to r^{c}) that describes a single step of the protocol. It selects a set of concretizations that covers all possible combinations of instantiations of parameters in req and r together, and concretize each pair (req, r) and record all the necessary actual parameter indices for each r separately.
3) Check whether req^{c} is an inductive invariant and, if necessary, find a strengthening aux that will be used as an auxiliary invariant. Here the concretizations enable this approach to exploit a BDD-based oracle to ensure

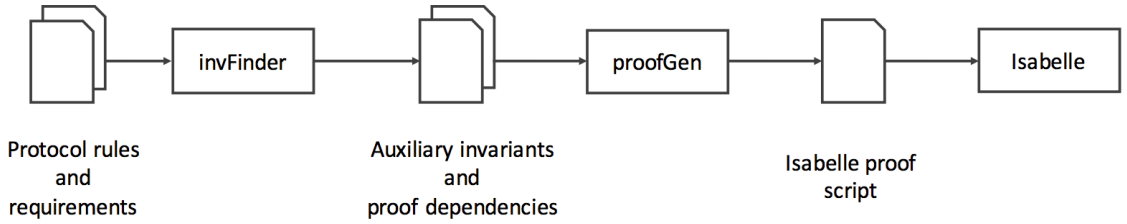

Fig. 1: The workflow of paraVerifier

that aux holds in a small instance of the protocol – this is achieved via the symbolic model checker NuSMV.[2]

4) Generalize aux into a parameterized form. This is possible as the approach carefully registers the indices of instances in the first two steps.

The above four steps have been implemented as a tool invFinder within the framework of paraVerifier [7] and its extension L-CMP [10], whose main workflow is depicted in Figure 1. paraVerifier also contains proofGen with the support of the theorem prover Isabelle [11], and it is supposed to be *fully automatic* – in contrast to other existing approaches, it automatically finds auxiliary candidate invariants, and automatically tags their proof dependencies. The tool proofGen automatically transforms them into Isabelle proof scripts, when will be in turn fed into Isabelle to prove the requirement req.

C. Finding auxiliary invariants

In the above step 3, one needs to check whether a (concretized) requirement req^c is an inductive invariant for a (concretized) guarded command r^c. The set $invs$ records all auxiliary invariants found so far. invFinder first computes the weakest precondition $wp^c \equiv \mathsf{WP}(\mathsf{action}(r^c), req^c)$ and then decides how to proceed as follows:[3]

1) If $wp^c \equiv req^c$, the statement $\mathsf{action}(r^c)$ does not change req^c. This typically holds for guarded commands that don't use variables appearing in req^c. invFinder registers *"r preserves req"* as a hint for the theorem prover.
2) If $\mathsf{guard}(r^c) \rightarrow wp^c$ is found to be a tautology, then the command r^c is only enabled if req^c holds afterwards. Thus, invFinder records *"r establishes req"* as a hint for the theorem prover.
3) If neither of the above two cases holds, a new auxiliary invariant aux will be constructed. invFinder registers *"r ensures req with the auxiliary invariant aux."*

To find a simple formula aux, invFinder first constructs $\neg \bigwedge_{n=1}^{k} f_n^c \iff \neg\mathsf{guard}(r^c) \vee wp^c$ and then considers all its subformulas starting with the smallest ones, such as $\neg f_1$, $\neg f_2$, $\ldots$; $\neg(f_1 \wedge f_2)$, $\neg(f_1 \wedge f_3)$, $\ldots$; $\neg(f_1 \wedge f_2 \wedge f_3)$, $\ldots$. Each subformula will be checked whether it is indeed an auxiliary invariant. In this paper, this will be implemented by sending the formula to an oracle running DT-Decide based on a data sheet collected from the RS of a protocol instance (see details in Section III). After finding an auxiliary invariant, it will be generalized by changing its concrete process identifiers to parameter variables $i_1, i_2, \ldots$. After this step, this auxiliary invariant is added to the set of requirements that need to be eventually checked with proofGen.

This main step will be repeated until proof dependencies for all pairs (req^c, r^c) have been checked or a pair (req^c, r^c) is found for which no proof dependency applies. The former case terminates the search for auxiliary invariants successfully, while the latter indicates that the requirement req^c cannot be handled by paraVerifier, most likely because req is not actually an invariant. paraVerifier often has the set of invariants converge quickly because of two reasons: (1) the chosen invariant candidate is a correct invariant in a small protocol instance; (2) the size of the chosen candidate is as small as possible because paraVerifier starts the search for auxiliary invariants with the smallest candidates.

TABLE I: A fragment of the output of invFinder

concrete invariant	rule	proof hint
$\mathsf{mutualInv}(1,2)$	$\mathsf{crit}(1)$	based on $\mathsf{invOnXC}(2)$
	$\mathsf{crit}(2)$	based on $\mathsf{invOnXC}(1)$
	$\mathsf{crit}(3)$	preserves $\mathsf{mutualInv}(1,2)$
$\mathsf{invOnXC}(1)$	$\mathsf{crit}(1)$	establishes $\mathsf{invOnXC}(1)$
	$\mathsf{crit}(2)$	establishes $\mathsf{invOnXC}(1)$

Example 2 *We continue with Example 1. Assume given the concrete requirement* $\mathsf{mutualInv}(1,2)$*, combined with the three concrete guarded commands* $\mathsf{crit}(1)$*,* $\mathsf{crit}(2)$*, and* $\mathsf{crit}(3)$*. The resulting outputs of the cases described above is shown in Table I. In the table, each line records a concrete invariant, a concrete guarded command and the hint for the theorem prover. The first line introduces an auxiliary invariant* $\mathsf{invOnXC}(1)$*. Its generalized form,* $\mathsf{invOnXC}(i)$*, is added to the set of candidate invariants, and it is then again concretized to check whether it is an inductive invariant. These checks are documented in the last two lines of Table I.*

III. Learning Invariants with Decision Trees

In this section, we present in detail our decision-tree based method to learn invariant candidates, in order to speed up the formal verification of parametric protocols.

[2]In this paper, we will use a decision tree that describes the reachable states as such an oracle, instead of using the BDD-based oracle.

[3]$\mathsf{action}(r)$ is used to return the parallel assignment part of the guarded command r.

A. Decision trees

A decision tree (DT) is a flowchart-like structure in which each internal node represents a test on an attribute (e. g. whether a coin flip comes up heads or tails), each branch represents one outcome of the test, and each leaf node represents a decision taken after testing all attributes. A decision tree is a compact representation to classify examples. The paths from root to leaf represent classification rules, which allow to predict the value of a target variable based on values of input variables. Data comes in records of the form $(\mathbf{x}, Y) = (x_1, x_2, x_3, ..., x_k, Y)$. The dependent variable Y is the target variable that we are trying to classify or generalize. The vector x is composed of the input variables x_1, x_2, x_3, etc. Algorithms for constructing decision trees usually choose a variable x_i at each step that best splits the set of items. Typical algorithms include ID3 (Iterative Dichotomiser 3) [12], C4.5 (a successor of ID3) [13], and CART (Classification And Regression Tree) [14].

One of the well-known metrics to pick the best variable is based on a notion of information gain in line with Shannon's entropy function. It calculates the entropy of every attribute x of the data set, then partitions the set into subsets using the attribute for which the resulting entropy after splitting is minimized (equivalently information gain is maximized). The inductive bias in these learning algorithms is roughly to compute the smallest decision tree that is consistent with the sample – a bias that again suits our setting well, as we would like to construct a small decision tree to save both time and memory for parameterized verification.

However, in a classical decision tree algorithm like ID3, when there is no more predicting attribute, then a single-node tree is returned with a label which is the most common value of the target attribute in the examples. This is not surprising because the idea of classical decision tree is based on statistics, that is, the classification (or decision) is based on the most frequent value of the target variable – *it underapproximates the data*. This ideal of predicting the most likely value is not suitable for our verification purpose. Intuitively, in our context, our data sheet corresponds to the reachable state set. We pick a target variable (attribute) and create a decision tree to classify the target variable based on the data sheet. Our goal is to create a model that predicts all possible values of the target variable based on the other variables – *we overapproximate the state space*. A path from root to a leaf is regarded as a classification rule, a formula f; or more exactly, a conjunction of atomic formulas, each of which restricts the value of one variable. These formulas f form a partition of the state space. We don't care about the frequencies of the different values for the target variable, but we do care to record all possible values. Except for the change just described, we can use any decision tree construction algorithm as-is.

Now we can give the algorithm shown in Algorithm 1 to make a decision tree $tree$ from a data sheet $data$ if a target attribute and other atrributes $attribs$ are given. Line 2 deals with the case that decision on the value of $targ\_attrib$ can be made at once because all the samples agree the same value a on it, thus the leaf node is returned with a label $targ\_attrib = a$; line 4 deals with the case no more attribute can be used to classify $targ\_attrib$, thus a leaf node with $targ\_attrib = a_1 \vee ... \vee targ\_attrib = a_m$. Here we need record all the possible values of $targ\_attrib$ instead of the most possible value of $targ\_attrib$. This change has been made for the use to classify state space completely. Line 6 begin to make branches, line 6 picks the "best" attribute A to classify samples. In order to make the classification best, we need calculate the entropy of every attribute a of the data set, then partition ("split") the set into subsets using the attribute for which the resulting entropy

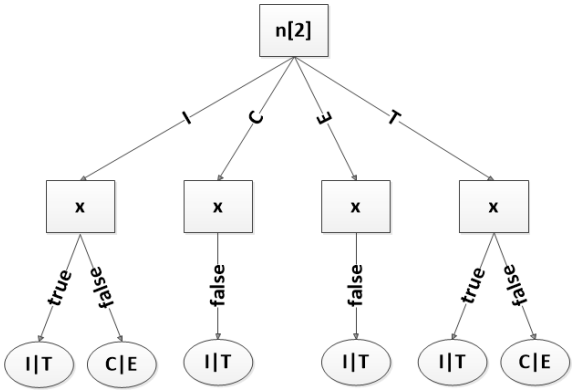

Fig. 2: Extended decision tree (target variable: $n[1]$)

Algorithm 1: makeDt (data, targ_attrib,attribs)

1 An extended ID3 decision tree making algorithm is shown in Algorithm 1.

Input: A data sheet $data$, target attribute $targ\_attrib$, atrributes $attribs$

Output: A decision tree $tree$ to classify $targ\_attrib$

1: **if** all the samples agree on the same value a on the $targ\_attrib$ **then**
2: return a single node labelled with $targ\_attrib = a$
3: **else if** $atrributes = \{targ\_attrib\}$ **then**
4: return a single node labelled with $targ\_attrib = a_1 \vee ... \vee targ\_attrib = a_m$, where a_1,...,a_n are all possible values of $targ\_attrib$ in $data$
5: **else**
6: $A \leftarrow$ the Attribute that best classifies examples.
7: create a node Root labelled with A.
8: **for** each possible value v_i of A **do**
9: Add a new tree branch $branch$ labelled with v_i below Root, corresponding to the test $A = v_i$
10: $Examples(v_i) \leftarrow$ be the subset of examples that have the value v_i for A
11: $branch \leftarrow$ makeDt$(Examples(v_i), targ_attrib, attribs - \{A\})$
12: **end for**
13: **end if**
14: return Root

after splitting is minimized; or, equivalently, information gain is maximum. From line 11, the algorithm makeDt is recurred to make a decision trees by using a subset of the samples that have the value v_i for A.

B. Decision-tree based invariant learning

In our method, we first collect the reachable state set from an explicit state-based model checker, i.e., Murphi,[4] which can enumerate all reachable states of a protocol instance. We transform the state set into a data sheet, similar to Table II.

TABLE II: Data sheet representing RS of $\mathcal{M}utual\mathcal{E}x_N$ with $N = 2$

No.	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}
$n[1]$	I	T	I	C	T	I	E	C	T	I	E	T
$n[2]$	I	I	T	I	T	C	I	T	C	E	T	E
x	t	t	t	f	t	f	f	f	f	f	f	f

t and f abbreviate true and false, respectively.

In order to verify that formula in the form of $\neg f$ is an invariant of the protocol instance, we call Algorithm 2, where $data$ is the data sheet extracted from the protocol instance. To save time and memory, we construct a decision tree $tree$ to decide f, as shown in Figure 2. We only select a sub-table $data'$ which contains just columns covering $\mathsf{vars}(f)$ and use it to create the decision tree, see in Line 3. We further store the newly created tree into a store of trees $stored$ for further deciding a formula which has the same variable set as $\mathsf{vars}(f)$. After having the sub-data sample $data'$, Line 5 selects a variable v form $\mathsf{vars}(f)$, which has more than one value in the data sheet S, if such a variable exists, and creates a decision tree $tree$ that classifies variable $target$. Otherwise, there is only one data sample in $data$ and Line 8 creates a simple tree by randomly selecting a variable as a leaf node and creating a single branch leading to it. In order to avoid false invariants caused by symmetry reduction (see next subsection), all the formulas f' which are symmetric to f are added to F, and the satisfiability of each formula in F, conjoined with a classification rule in $tree$, is checked. The call $path2Form(p)$ in Line 16 computes a formula f_p describing the classification rule implied by the path, and $\mathsf{elimX}(f_p)$ eliminates references to "don't care"-values. A SMT solver like Z3 is used for satisfiability checking. If $f' \wedge \mathsf{elimX}(f_p)$ is satisfiable, $\neg f$ is not an invariant. If this formula is unsatisfiable for all $p \in paths(tree)$, then $\neg f$ is indeed an invariant.

C. Symmetry reduction

In this and the following subsections, we present two techniques to make the data sheet as compact as possible. The number of states in the reachable state set normally affects the speed of learning. Symmetry reduction is the removal of states that are symmetric to some other state; in our case, symmetry is defined through permuting the node identities. This is able to prune the state space dramatically, but it can lead to some conclusions that are incorrect in the original reachable state set. Figure. 3 illustrates the symmetry of the state space for

[4]http://formalverification.cs.utah.edu/Murphi/index.html

Algorithm 2: DT-decide ($\neg f$)

Input: negative formula $\neg f$, a data sheet $data$
Output: Boolean value showing whether $\neg f$ is an invariant

1: Look up whether there is a $tree \in stored$ for $\mathsf{vars}(f)$
2: **if** $tree$ has not been found **then**
3: $data' \leftarrow$ the projection of $data$ on $\mathsf{vars}(f)$
4: **if** $data'$ is not trivial **then**
5: $target \leftarrow$ a suitable target variable in $data'$
6: Create a decision tree $tree$ from $data'$ for $target$
7: **else**
8: Create a single-path tree for $data'$
9: **end if**
10: Append $tree$ to $stored$
11: **end if**
12: $F \leftarrow \{f' | f'$ is symmetric to $f\}$
13: $isInv \leftarrow true$
14: **for** $p \in paths(tree)$ **do**
15: **for** $f' \in F$ **do**
16: **if** $sat(f' \wedge (\mathsf{elimX}(path2Form(p))))$ **then**
17: **return** "$\neg f$ is not an invariant"
18: **end if**
19: **end for**
20: **end for**
21: **return** "$\neg f$ is an invariant"

Example 1. The state space is partitioned into two parts, each of which has seven states. $s_1 \sim s_2$, $s_3 \sim s_5$, ... According to symmetry reduction, only states $s_0, s_1, s_3, s_4, s_6, s_7, s_{10}$ are explored and will be extracted as a data sheet of reachable states, as shown in Table III. so we can reduce the data sheet to Table III.

TABLE III: Reduced data sheet representing RS of $\mathcal{M}utual\mathcal{E}x_N$ with $N = 2$

No.	s_0	s_1	s_3	s_4	s_6	s_7	s_{10}
$n[1]$	I	T	C	T	E	C	E
$n[2]$	I	I	I	T	I	T	T
x	t	t	f	t	f	f	f

It is clear that the number of states has been reduced from 12 to 7. However, from the reduced Table III, we could derive a spurious invariant $n[2] = \mathsf{C} \longrightarrow n[1] = \mathsf{C}$, because it doesn't find any sample which satisfies the antecedent $n[2] = \mathsf{C}$. In order to avoid this type of errors, we will check additional symmetric formulas. The principle is to check all formulas f' which are symmetric to f in the reduced space. For instance, to assert $n[2] = \mathsf{C} \longrightarrow n[1] = \mathsf{C}$, we also need to check $n[1] = \mathsf{C} \longrightarrow n[2] = \mathsf{C}$ in the reduced state space.

Example 3 *Now we illustrate the above technique by a simple mutual exclusion protocol with data manipulations with N nodes. a is extended to be an array containing local variables consisting of two fields: $a[i].st$ describes the state of node i, while $a[i].d$ the data stored locally. Two more global data*

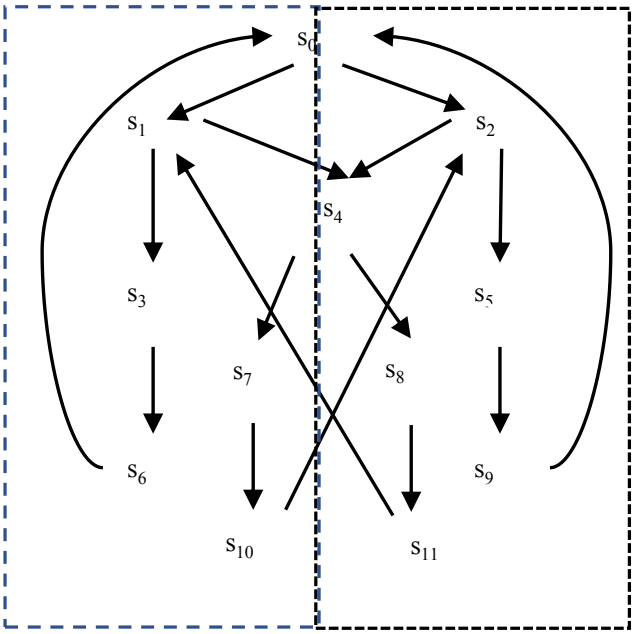

Fig. 3: Symmetric state space illustration.

variables $memD$ and $auxD$ are used to indicate copies of data stored in the main memory and for the most fresh data.

This mutual exclusion protocol has the following rules:

$$\begin{array}{lll}
\mathsf{try}(j) & \equiv & a[j].st = \mathsf{I} \ \rhd \ a[j].st := \mathsf{T} \\
\mathsf{crit}(j) & \equiv x \wedge a[j].st = \mathsf{T} \rhd \{a[j].st := \mathsf{C}, x := \mathsf{false}, \\
 & & \qquad a[j].d := memD\} \\
\mathsf{exit}(j) & \equiv & a[j].st = \mathsf{C} \ \rhd \ a[j].st := \mathsf{E} \\
\mathsf{idle}(j) & \equiv & a[j].st = \mathsf{E} \rhd \{a[j].st := \mathsf{I}, x := \mathsf{true}, \\
 & & \qquad memD := auxD\} \\
\mathsf{store}(j,d) & \equiv & a[j].st = \mathsf{C} \rhd \{a[j].d := d, auxD := d\}
\end{array}$$

The protocol itself is defined by $mutexini_N$, the predicate specifying the initial state of the protocol, and $mutexrules_{N,D}$, the set of protocol rules or guarded commands:

$$\begin{array}{ll}
mutexini_N & \equiv x \wedge memD = auxD \wedge \bigwedge_{i=1}^{N} a[i] = \mathsf{I} \\
mutexrules_{N,D} & = \{\mathsf{try}(j)\ ,\ \mathsf{crit}(j)\ ,\ \mathsf{exit}(j)\ ,\ \mathsf{idle}(j)\ , \\
 & \qquad \mathsf{store}(j,d) \mid j = 1,...,N; d = 1,...,D\} \\
\mathcal{M}utual\mathcal{E}x_{N,D} & = (mutexini_N, mutexrules_{N,D})
\end{array}$$

and it should satisfy the invariant requirement:

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

$$\mathsf{dataInv}(i_1) \equiv \neg(a[i_1].st = \mathsf{C} \wedge a[i_2].d \neq auxD)$$

The property $\mathsf{mutualInvData}(i_1, i_2)$ *is very similar to* $\mathsf{mutualInv}(i_1, i_2)$ *in Example 1, while* $\mathsf{dataInv}(i_1)$ *specifies that $a[i_1].d$ always stores the most fresh copy of data $auxD$ when $a[i].st$ is* C*. For protocol $\mathcal{M}utual\mathcal{E}x_{N,D}$ when $N = 2$ and $D = 2$ the number of states in its* RS *without symmetry reduction is 88, while it is only 23 after symmetry reduction.*

D. "Don't care"-values

In addition to symmetry reduction discussed in the previous subsection, we also explore the use of "don't care"-values. Such a value is an abstraction of all possible concrete values. The technique of "don't care"-values is frequently used in STE or Murphi. Usually, it is combined with symmetry reduction to reduce the explored state space further. For instance, for the simple mutual exclusion protocol with data $\mathcal{M}utual\mathcal{E}x_{N,D}$ when $N = 2$ and $D = 2$, we can reduce the number of explored RS to 13 if we use both symmetry reduction and "don't care"-values. As an example, if the data value in the cache is irrelevant, we set it to "don't care": if $n[1].st = \mathsf{I} \wedge n[2].st = \mathsf{I} \wedge x$, all data variables $n[1].d$, $n[2].d$, $auxD$ and $memD$ are irrelevant, so we can compress $D^4 = 16$ entries in the data sheet into one.

For the *FLASH* protocol in our experiments, the number of reachable states is 107,866,864, while with symmetry reduction and "don't care"-values it is only 857,453. The latter is about 0.8 % of the former.

E. Extending paraVerifier with DT-decide

We extend the paraVerifier framework with this new invariant learning approach based on decision trees (see Figure 4). Essentially, we use Murphi to get the reachable state set of a protocol instance, and transform the state set into a data sheet. With the algorithm DT-decide, we explore the data sheet and construct decision trees as necessary. The trees act as an oracle to check invariant candidates received from invFinder.

IV. Experiments

We have applied our new framework (Figure 4) to six benchmarks. Among these protocols, *Mutual Exclusion, MOESI, MESI* are small-scale protocols, while *German* [15] and *FLASH* protocols[5] are more complicated. The *FLASH* protocol is much more complex and realistic than *German*. For a 3-node *FLASH* instance, the number of reachable states is around 1500 times that of 3-node *German*.

Table IV gives a comparison of verification time and memory consumption for the benchmarks between the BDD-based oracle (from the original paraVerifier framework) and the decision-tree-based oracle (from our new framework).[6] At first sight, for small protocols (*Mutual Exclusion, MOESI, MESI*) the decision-tree-based oracle is slightly slower than the BDD-based oracle, but requires less memory. However, for large examples the decision-tree-based oracle performs much better: for *FLASH_nodata* it takes 20 minutes with 511 MB memory while paraVerifier needs 244.67 minutes with 2.4 GB memory; for *FLASH_data* it takes 166.67 minutes (3.2 GB memory) which is about 50% (67%) of paraVerifier. The main reason is that running NuSMV to obtain a BDD-based oracle for large examples takes much more time than the decision-tree based approach.

The experimental results that show our decision-tree based invariant learning framework can save memory and speed up parameterized verification for large protocols. Our results further confirm the following statement in [6]:

[5]We verify the protocol with and without data modeling.

[6]The verification time reported in Table III [7] did not include the time of computing the BDD reachable state spaces of the protocol instances.

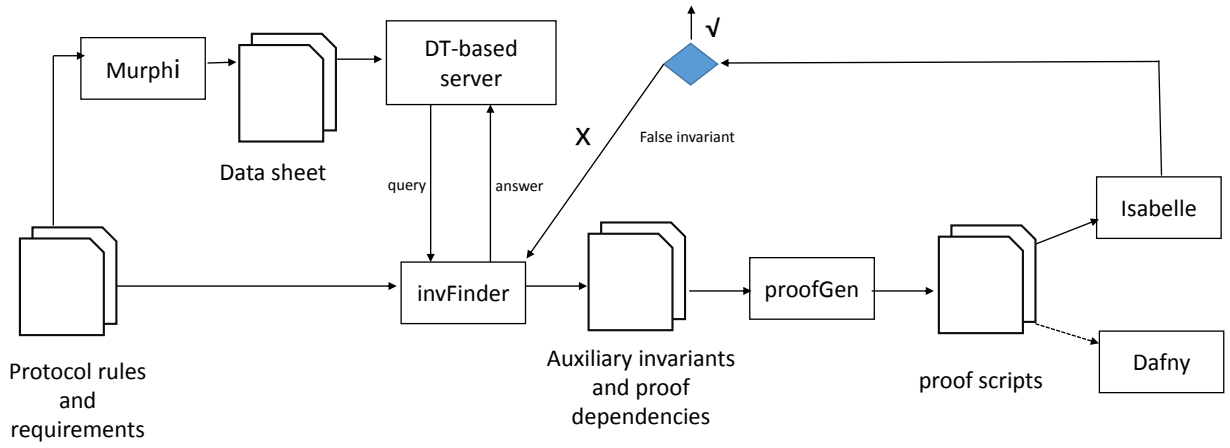

Fig. 4: Our new framework for parameterized verification: extending paraVerifier with decision-tree based invariant learning

TABLE IV: Comparison of verification time and memory consumption: BDD-based oracle vs. DT-based oracle

	time		memory	
	DT	BDD	DT	BDD
mutualEx	6.0s	3.5s	25M	7.3M
MESI	4.8s	4.6s	25M	84M
MOESI	5.0s	4.8s	25M	85M
Germanish	3.0s	2.5s	25M	90M
German	60.0s	38.6s	42M	135M
FLASH_nodata	20m	244.67m	511M	2.4G
FLASH_data	166.67m	248.33m	3.2G	4.8G

> *... as experience shows that for cache coherence protocols explicit-state model checkers are often superior to symbolic model checkers ....*

In our work, the use of an explicit state-based model checker Murphi, combined with symmetry reduction and decision-tree based invariant learning, is the key to successfully prove the correctness of large cache coherence protocols.

V. Related Work

There have been a lot of studies applying machine learning techniques in the field of formal verification. We discuss papers that are mostly relevant to our work. The most successful work in applying decision-tree based learning techniques in invariant synthesis is [16], [17], they proposed the classification concept of *implication*, in addition with positive and negative samples. Then decision-tree learning algorithms are extended to classify implication samples, and they develop several scalable methods to construct small decision trees using statistical measures. An extensive study of various natural measures for learning decision trees is also performed in [16], [17]. Some learners and an appropriate teacher are implemented for invariant synthesis, The experimental results show that their new algorithms are efficient and convergent for a large suite of programs. In the context of parameterized verification, positive samples are just the states in the research state set RS; however, it is difficult to compute suitable negative examples, which are not contained in RS. The number of negative states is still much larger than that of RS. It is quite difficult to propose good measures to sample these negative states, and the quality of invariants depends heavily on these sample of negative samples. Thus, we adapt a different strategy, by extending the classical decision tree learning algorithm mainly for partitioning reachable states, i.e., positive states. we select a variable which has more than one value in the data sheet and use other variables to classify the value of the target variable. A single case corresponds to a path from the root node to a leaf node, where all possible values of the target variable are enumerated. The disjunction of all the cases thus overapproximates the state space modeled by the state set.

In the last decade, how to find sufficient and appropriate invariants automatically has gradually been an active research area [18], [19], [20], [21], [22], [8]. The concept of "invisible invariants" has been proposed in 2001 [23]. It is an automatic technique for parameterized verification. In this method, auxiliary invariants are computed in a finite system instance to aid inductive invariant checking. Combining this idea with parameterized abstraction, Lv *et al.* used a small protocol instance to compute candidate invariants [24]. However, the invariants found by these works are "raw" boolean predicates, which are hard to understand. Later, a SMT-based model checker Cubicle has been proposed [22] and developed [25]. In their works, the BRAB algorithm has been introduced, which can automatically infer invariants and generate Why3-proof certificates for SMT-solvers. It was the first tool that proves automatically the safety properties of *FLASH*. Although these works can automatically verify parametric protocols, the invariants they found are not easily readable and understandable.

VI. Discussions and Future Work

To the best of our knowledge, this work is the first one to use decision trees to learn invariants from a data sheet describing the reachable states of a protocol and apply the learned invariants to accelerate parameterized verification. A standard decision tree learning algorithm has been extended to represent an over-approximation of the reachable states of

a protocol, and formulas can be constructed corresponding to each path to form classification rules. In turn, we can use these formulas as a compact knowledge base to decide whether a formula is satisfied. Symmetry reduction and "don't care"-values are two essential techniques to obtain a small reachable state set, which help achieve much smaller data sheets. We also adapt the checking of invariant candidates to ensure that no false invariants are learnt because symmetry reduction skips some states. We have integrated the decision-tree based learning technique into paraVerifier and used it as an oracle to learn invariants for parameterized verification. From our experiments, we observed that such an oracle is quite effective. In most cases, more than one third of memory can be saved in our experiments by using the decision-tree based oracle instead of a NuSMV BDD-based oracle.

Our decision-tree based invariant learning approach can be considered as a grey-box technique. We need to know which candidate formulas are to be decided, these candidates are output from invFinder by analyzing the preconditions and guards. However, the decision-tree based oracle is completely agnostic of the program and the specification once it has received the formula, and it only needs to decide this formula according to the data sheet and the learnt decision tree.

We believe our approach can be useful for checking invariants of protocols whose state space are difficult to be enumerated by symbolic approaches. We can simulate a protocol, collect reachable states from these simulations and transform this information into a decision tree, from which we can learn invariants and try to construct an inductive invariant set. In future, we will extend our work to deal with general safety properties (not only invariants) and liveness properties. Another direction is to extend our work to learn loop invariants for program verification.

Acknowledgement. This work was partially supported by grant 2017YFB0801900 from the National Key R&D Program and grant 61672503 from National Natural Science Foundation in China with the same strength, in no particular order.

REFERENCES

[1] D. L. Dill, "The Murφ verification system," in *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, ser. Lecture Notes in Computer Science, vol. 1102. Springer, 1996, pp. 390–393.

[2] C. N. Ip and D. L. Dill, "Efficient verification of symmetric concurrent systems," in *Proc. 11th International Conference on Computer Design (ICCD'93)*. IEEE, 1993, pp. 230–234.

[3] S. C. C. Blom, J. R. Calame, B. Lisser, S. M. Orzan, J. Pang, J. C. van de Pol, M. Torabi Dashti, and A. J. Wijs, "Distributed analysis with μCRL: a compendium of case studies," in *Proc. 13th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, ser. Lecture Notes in Computer Science, vol. 4424. Springer, 2007, pp. 683–689.

[4] K. L. McMillan, *Symbolic Model Checking*. Kluwer, 1993.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. 5th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, ser. Lecture Notes in Computer Science, vol. 1579. Springer, 1999, pp. 193–207.

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

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

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

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

[10] J. Cao, Y. Li, and J. Pang, "L-CMP: An automatic learning-based parameterized verification tool," in *Proc. 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. ACM Press, 2018, pp. 892–895.

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

[12] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[13] ——, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[14] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Chapman & Hall/CRC, 1984.

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

[16] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *Proc. 26th International Conference on Computer Aided Verification (CAV'14)*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 69–87.

[17] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proc. 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, 2016, pp. 499–512.

[18] 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.

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

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

[21] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 413–427.

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

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

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

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