Case Studies on Extracting the Characteristics of the Reachable States of State Machines Formalizing Communication Protocols with Inductive Logic Programing

Dung Tuan Ho*, Min Zhang**, and Kazuhiro Ogata*

*Japan Advanced Institute of Science and Technology (JAIST)
{dung.ho,ogata}@jaist.ac.jp
**East China Normal University (ECNU)
zhmtechie@gmail.com

Abstract. A distributed system DS can be formalized as a state machine M and many desired properties of DS can be expressed as invariants of M. An invariant of M is a state predicate p of M such that p holds for all reachable states of M. To verify that DS enjoys a desired property, namely to prove that p is an invariant of M, we often need to find other invariants as lemmas, which is one of the most intellectual activities in Interactive Theorem Proving (ITP). For this end, our experiences on ITP tell us that it is useful to get better understandings of the reachable states R_M of M. We report on case studies in which Progol, an Inductive Logic Programming (ILP) system, has been used to extract the characteristics of the reachable states of state machines formalizing communication protocols. The case studies demonstrate that ILP has potential abilities to extract the characteristics of R_M.

Keywords: Inductive Logic Programming, Interactive Theorem Proving, Invariant, Lemma, Progol, State Machine, Reachable State

1 Introduction

A *state machine* M consists of a set S of states that includes the initial states I and a binary relation T over states. $(s, s') \in T$ is called a *transition*. *Reachable states* R_M of M are inductively[1] defined as follows: $I \subseteq R_M$, and if $s \in R_M$ and $(s, s') \in T$, then $s' \in R_M$. A distributed system DS can be formalized as M and many desired properties of DS can be expressed as invariants of M. An *invariant* of M is a state predicate p of M such that p holds for all $s \in R_M$.

To prove that p is an invariant of M, it suffices to find an inductive invariant q of M such that $q(s) \Rightarrow p(s)$ for each $s \in S$. An *inductive invariant* q of M is a state predicate of M such that $(\forall s_0 \in I)\, q(s_0)$ and $(\forall (s, s') \in T)\, (q(s) \Rightarrow q(s'))$. Note that an inductive invariant of M is an invariant of M but not vice versa.

[1] Note that "induction" is used to refer to two different meanings: one from machine learning and the other from mathematical induction.

Finding an inductive invariant q (or conjecturing a lemma q) is one of the most intellectual activities in ITP[2]. This activity requires human users to profoundly understand the system under verification or M formalizing the system to some extent. The users must rely on some reliable sources that let them get better understandings of the system and/or M to conduct the non-trivial task, namely *lemma conjecture*. For this end, our experiences on ITP tell us that it is useful to get better understandings of R_M. Some characteristics of R_M can be used to systematically construct a state predicate q_i that is a part of q.

$s \in S$ is characterized by some values that are called *observable values*. Based on our experiences on ITP, the characteristics of R_M are correlations among observable values of the elements of R_M. Generally, the number of the elements of R_M is unbounded and then a huge number of reachable states are generated from M. The task of extracting correlations among a huge number of data (reachable states in our case) is the role of Machine Learning (ML).

We have conducted two case studies on Alternating Bit Protocol (ABP, a simplified version of Sliding Window Protocol used in TCP) and Simple Communication Protocol (SCP, a simplified version of ABP) using a framework in which Progol, an ILP system, has been mainly used to extract some characteristics of the reachable states of their state machines formalizing the protocols. Before the two case studies reported in this paper, we (especially the third author) had conducted verification case studies in which it is proved that both protocols enjoy what is called the reliable communication property that whenever the current data to be delivered is i, the data upto i or $i-1$ have been successfully delivered to the receiver from the sender without any duplications nor drops. Through those verification case studies, we drew four possible reachable state patterns (see Fig. 5) for SCP and six possible reachable state patterns (see Fig. 6) for ABP. Those state patterns can be used as oracles for judging if learned hypotheses are reasonably good.

The rest of the paper is organized as follows. Sect. 2 introduces our motivation together with some background knowledge of systems verification with ITP and Sect. 3 briefly shows the verification process on the case studies. Sect. 4 describes our framework that is a combination of the tools used to construct an ILP input from a system specification that is suitable for a theorem prover. Sect. 5 summaries the results on some experiments for the case studies using our framework to extract the characteristics of R_M with ILP. Sect. 6 mentions some related work. Sect. 7 concludes the paper and mentions some future directions.

2 Preliminary

Systems verification is a research area aiming at rigorously checking if systems satisfy desired properties. ITP is a formal verification technique in which mathematical models are made for systems and desired properties are treated as theo-

[2] q may be in the form $q_1 \wedge \ldots \wedge q_n$. Each q_i may be called a lemma and is an invariant of M if q is an inductive invariant of M, although q_i may not be an inductive invariant of M.

rems of the mathematical models. State machines are used as such mathematical models. For example, it is possible to check if an e-commerce protocol satisfies the property that if an acquirer authorizes a payment, then both the buyer and seller concerned always agree on it [1]. Logics, theories, techniques and tools for theorem proving have been advanced a lot, e.g. logical decision procedures used in SMT [2]. However, some non-trivial interactions between human users and theorem provers are still needed to conduct proofs that non-trivial state machines enjoy non-trivial properties. One of the most intellectual activities in such interactions is to conjecture lemmas.

To formally verify that a system satisfies a desired property with ITP, the system is first formalized as a state machine M that is described in a formal specification language. A state predicate p is described in the same or a different specification language for the property. An interactive theorem prover is used to prove that p is an invariant of M. We use a proof score approach to systems verification called the OTS/CafeOBJ method[3], in which CafeOBJ, an algebraic specification language and system used as a specification language for M and p and also as an interactive theorem prover. There are three main activities in the OTS/CafeOBJ method to conduct ITP: application of simultaneous structural induction (SSI), case analysis (CA) and use of lemmas (including lemma conjecture).

Let us consider a mutual exclusion protocol called TAS as an example. TAS written in an Algol-like language is shown in Fig. 1 (a). TAS uses *lock* to control processes such that there is at most one process in Critical Section (or at cs). Initially, *lock* is false and each process is in Remainder Section (or at rs). test&set(b) atomically sets b true and returns false if b is false, and just returns true otherwise. TAS is formalized as a state machine M_{TAS} whose transitions are depicted in Fig. 1 (b) and (c). The arrow on which try_x and $[b = \mathrm{f}]$ are attached is interpreted as follows: if process x is at rs and b is false in a given state, then x moves to cs and b is set true. The arrow on which exit_x is attached is interpreted as follows: if process x is at cs, then x moves to rs and b is set false. Note that transitions are declared in terms of equations in the OTS/CafeOBJ method. One desired property TAS should enjoy is the mutual exclusion property. Let $\mathrm{mx}(s, x, y)$ be $(\mathrm{pc}(s, x) = \mathrm{cs} \wedge \mathrm{pc}(s, y) = \mathrm{cs} \Rightarrow x = y)$, where s is a state x, y are process identifiers and $\mathrm{pc}(s, x)$ is the location (rs or cs) where process x is in state s, and let $\mathrm{mx}(s)$ be $(\forall x, y \in \mathrm{Pid})\, \mathrm{mx}(s, x, y)$, where Pid is the set of all process identifiers. To verify that TAS enjoys the property, all we have to do is to prove that $\mathrm{mx}(s)$ is an invariant of M_{TAS}.

Fig. 2 shows a snip of a proof tree that $\mathrm{mx}(s)$ is an invariant of M_{TAS}, although proofs are written as texts in the OTS/CafeOBJ method. Given a state s and a process identifier k, $\mathrm{try}(s, k)$ is the state obtained by applying transition try_k in s, $\mathrm{exit}(s, k)$ is the state obtained by applying transition exit_k in s, and $\mathrm{lock}(s)$ is the Boolean value stored in variable *lock* in s. SSI on s is first used to split the initial goal into three sub-cases. What to do for the three sub-

[3] Due to the space limitation, we do not explain the OTS/CafeOBJ method in detail. Please refer to [3, 4] for the OTS/CafeOBJ method

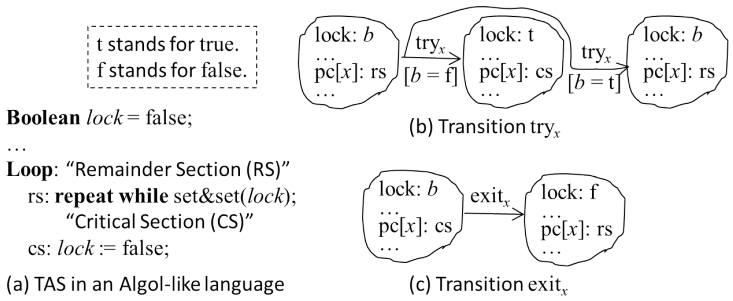

Fig. 1. TAS and a state machine M_{TAS} formalizing TAS

cases is to show $\mathrm{mx}(s_0, i, j)$, $\mathrm{mx}(s, i, j) \Rightarrow \mathrm{mx}(\mathrm{try}(s, k), i, j)$ and $\mathrm{mx}(s, i, j) \Rightarrow \mathrm{mx}(\mathrm{exit}(s, k), i, j)$, respectively, where s_0 is an arbitrary initial state, s is an arbitrary state, and i, j, k are arbitrary process identifiers. CA is then repeatedly used until what to show reduces either true or false. Any case in which what to show reduces true is discharged. For any case in which what to show reduces false, we need to conjecture lemmas[4]. Let us consider the case marked Case A in Fig. 2 in which $\mathrm{mx}(s, i, j) \Rightarrow \mathrm{mx}(\mathrm{try}(s, k), i, j)$ reduces false. Therefore, Case A needs a lemma. Let $\mathrm{lem1}(s, i)$ be such a lemma. We will soon describe how to conjecture the lemma. $\mathrm{lem1}(s, i) \Rightarrow (\mathrm{mx}(s, i, j) \Rightarrow \mathrm{mx}(\mathrm{try}(s, k), i, j)$ reduces true, discharging Case A, provided that we prove that $(\forall x \in \mathrm{Pid})\, \mathrm{lem1}(s, x)$ is an invariant of M_{TAS}. The proof needs $\mathrm{mx}(s, i, j)$ as a lemma. This is why we use simultaneous structural induction.

Let P and Q be the sets of states that correspond to predicates p and q, respectively. S, I, R_M, P and Q can be depicted as shown in Fig. 3. Proving that p is an invariant of M is the same as proving $R \subseteq P$. Let $(s, s') \in T$ be an arbitrary transition. In each induction case or a subcase of each induction case, all needed is basically to show $p(s) \Rightarrow p(s')$ so as to prove that p is an invariant of M. There are four possible situations: (1) s, $s' \notin P$, (2) $s \notin P$ and $s' \in P$, (3) $s, s' \in P$, and (4) $s \in P$ and $s' \notin P$. $p(s) \Rightarrow p(s')$ holds for (1), (2) and (3), but does not for (4). To complete the proof that p is an invariant of M, we need to know $s' \notin R_M$ for (4), namely that s' is not reachable for (4). To this end, we need to conjecture a lemma q such that q does not hold for s'. Case A in Fig. 2 is an instance of (4). Case A is characterized with $\mathrm{pc}(s, k) = \mathrm{cs}$, $\mathrm{lock}(s) = \mathrm{false}$, $i \neq k$, $j = k$, and $\mathrm{pc}(s, i) \neq \mathrm{cs}$ (that are attached to the path to Case A from the root), from which we can systematically conjecture the following lemma: $\neg(\mathrm{pc}(s, k) = \mathrm{rs} \wedge \neg\mathrm{lock}(s) \wedge i \neq k \wedge j = k \wedge \mathrm{pc}(s, i) = \mathrm{cs})$. This lemma could be used to discharge Case A, but lemmas should be shorter because we need to prove that lemmas are invariants of M. Any state predicate that implies the lemma could be a lemma, one of which is $\neg(\mathrm{pc}(s, i) = \mathrm{cs} \wedge \neg\mathrm{lock}(s))$ that is equivalent to $\mathrm{pc}(s, i) = \mathrm{cs} \Rightarrow \mathrm{lock}(s)$ that is $\mathrm{lem1}(s, i)$.

[4] It is possible and/or necessary to conjecture and use a lemma to discharge a case even though what to show in the case does not reduce to false.

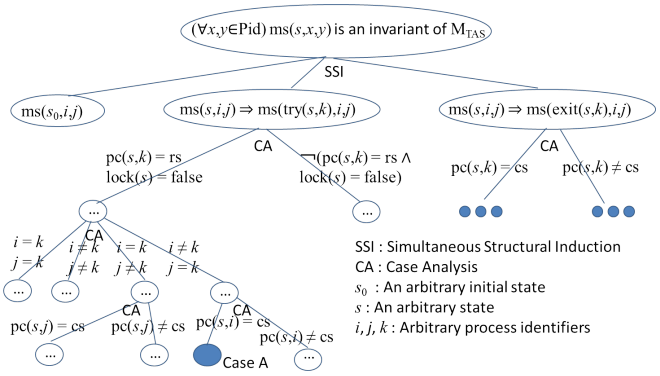

Fig. 2. A snip of a proof tree that mx(s) is an invariant of M_{TAS}

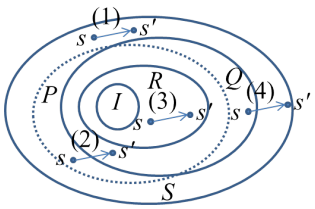

Fig. 3. Some possible situations when proving that p is an invariant of M

The systematic way to conjecture lemmas may not work for larger examples than TAS because case analysis may have to be repeated too many times until what to show reduces either true or false. Even if we reach the case in which what to show reduces false, a lemma conjectured could be so long that we may find it trouble to prove that the lemma is an invariant of M.

Our experiences on ITP tell us that better understandings of M and/or how M behaves let us conjecture useful lemmas to complete the proof concerned. Moreover, the properties we are interested in are invariants in this paper. Therefore, it suffices to get better understandings of R_M. In general, R_M contains an infinite number of states, and the task of extracting knowledge from such a huge database is the role of ML. However, classical machine-learning techniques only work for a database whose elements are expressed in propositional form, while our database consists of system states expressed in first-order form. There is the ML technique that can deal with first-order forms: Inductive Logic Programming (ILP). This is why we use ILP.

3 Verification of Communication Protocol

Communication Protocol is a class of algorithms designed to manage the data transmitted between senders and receivers through unreliable channels such that

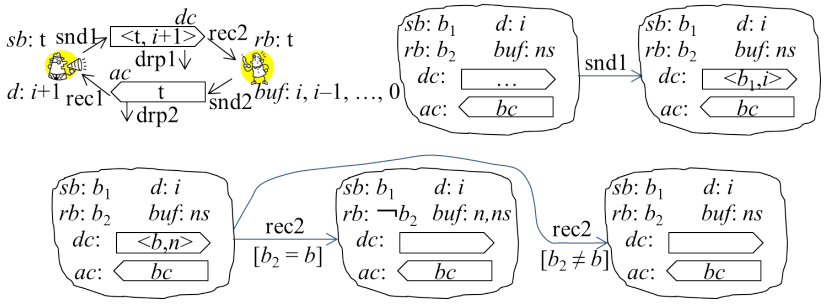

Fig. 4. SCP and part of a state machine M_{SCP} formalizing SCP

packets may be duplicated and dropped. There are many such algorithms proposed so far. In this paper, we take into account two simplified versions of TCP: SCP and ABP, where SCP is a simplified version of ABP. Fig. 4 shows a snapshot (a state) of a state machine M_{SCP} formalizing SCP and depicts two classes of transitions out of six in M_{SCP}. Let Bool, Nat, List, BNPair, PCell, and BCell be the sets (or the types) of Boolean values, natural numbers, lists of natural numbers, Bool-Nat pairs, cells of BNPair, and cells of Bool, respectively. Let t and f denote true and false, respectively. Each state of M_{SCP} is characterized by six values: $sb \in$ Bool, $d \in$ Nat, $rb \in$ Bool, $buf \in$ List, $dc \in$ PCell, and $ac \in$ BCell. Those initial values are t, 0, t, nil, empty, and empty, respectively. ac is used to deliver $\langle sb, d\rangle$ to Receiver from Sender, and bc is used to deliver rb (used as an ack of the data received and stored as the top of buf by Receiver) to Sender from Receiver. M_{SCP} has six classes of transitions: snd1, rec1, snd2, rec2, drp1, and drp2. snd1 puts $\langle sb, d\rangle$ into dc. If ac has $b \in$ Bool, then rec1 gets b from ac, and complements (or negates) sb and increments d if $b \neq sb$. snd2 puts rb into ac. If dc has $\langle b, n\rangle \in$ BNPair, then rec2 gets $\langle b, n\rangle$ from dc, and complements rb and stores n in buf at top if $b = rb$. If dc has $\langle b, n\rangle \in$ BNPair, then drp1 just drops $\langle b, n\rangle$ from dc. If ac has $b \in$ Bool, then drp2 just drops b from ac.

Let PQueue and BQueue be the sets of queues of BNPair and queues of Bool, respectively. The difference between ABP and SCP is as follows. $dc \in$ PQueue and $ac \in$ BQueue are used in ABP. snd1, rec1, snd2, rec2, drp1 and drp1 in ABP are quite similar to those in SCP, although the top element of each queue is taken into account in ABP. ABP has two more classes of transitions dup1 and dup2. If dc and ac are not empty, then dup1 and dup2 duplicates the top element of dc and ac, respectively, in ABP. Let M_{ABP} be a state machine formalizing ABP.

One property SCP and ABP should enjoy is what is called the reliable communication property. The property can be described as follows: all data up to the current one d or the previous one $d-1$ have been successfully delivered to Receiver without any duplicates nor any drops. To verify that SCP and ABP enjoy the property, all we have to do is to prove that the following state predicate is an invariant of M_{SCP} and M_{ABP}, respectively: $((sb = rb) \Rightarrow (buf =$

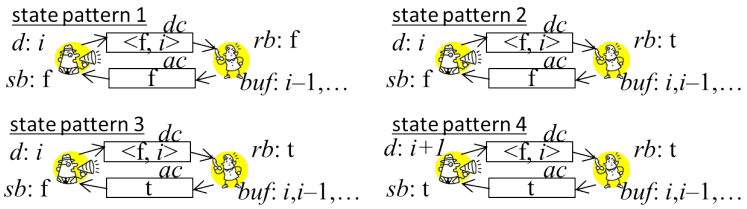

Fig. 5. Four state patterns of M_{SCP}

$d-1,\ldots,0)) \wedge ((sb \neq rb) \Rightarrow (buf = d, d-1,\ldots,0))$. Let rcp be the state predicate. We may explicitly write rcp(s), where s is a state. Conducting the formal verification that rcp is an invariant of M_{SCP} and gradually getting better understandings of SCP, we have realized that the reachable states of M_{SCP} can be classified into the four state patterns shown in Fig. 5, and lemmas can be conjectured from the four state patterns. To prove that rcp(s) is an invariant of M_{SCP}, we first apply SSI to s, generating seven sub-cases (or sub-goals). One sub-case is the induction case in which rec2 is taken into account. Let us consider the induction case. The case is first split into two sub-cases based on the condition of rec2: (1) dc is empty and (2) dc is not empty. Case (1) is discharged. For case (2), let dc contain $\langle b, n\rangle$. Case (2) is further split into two sub-cases based on whether rb equals b: (2-1) $rb \neq b$ and (2-2) $rb = b$. Case (2-1) is discharged. Case (2-2) is further split into two sub-cases based on whether sb equals b: (2-2-1) $sb \neq b$ and (2-2-2) $sb = b$. Case (2-2-1) is not discharged without use of any lemmas. The four state patterns shown in Fig. 5 let us realize that state pattern 1 is one and only one such that rb equals b, from which we can conjecture the lemma: $dc = \mathrm{c}(\langle b, n\rangle) \wedge rb = b \Rightarrow \langle sb, d\rangle = \langle b, n\rangle$, where c is the constructor of PCell for non-empty cells. The lemma is used to discharge case (2-2-1). Case (2-2-2) is further split into two sub-cases based on whether d equals n: (2-2-2-1) $d \neq n$ and (2-2-2-2) $d = n$. Case (2-2-2-1) is discharged with another lemma. Case (2-2-2-2) is discharged with a simple lemma of Boolean values. Then, the induction case is discharged.

Conducting the formal verification that rcp is an invariant of M_{ABP} and gradually getting better understandings of ABP, we have realized that the reachable states of M_{ABP} can be classified into the six state patterns shown in Fig. 6, and lemmas can be conjectured from the six state patterns.

4 Framework

ILP is a machine learning technique for constructing concept definitions (logic programs) from examples and a logical domain theory (background knowledge)[5]. In general setting, an ILP learning task is defined as follows. Given a background knowledge B and examples E consisting of positive examples E^+ and negative examples E^-, such that $B \not\models E^+$ and $B \wedge E^- \not\models \square$, the aim is then to find

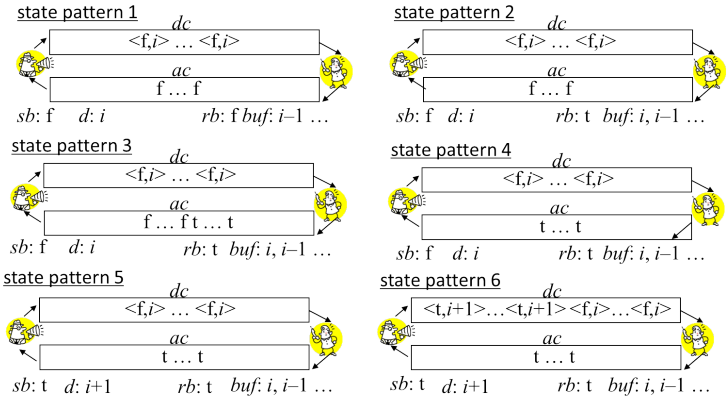

Fig. 6. Six state patterns of M_{ABP}

a hypothesis H such that $B \wedge H \models E^+$ (completeness) and $B \wedge H \wedge E^- \not\models \square$ (consistency). To fully describe a learning task in ILP, we need to clearly define B, H, E together with E^+ and E^-. Since our learning task is to characterize R_M, E is a set of system states consisting of reachable states (E^+) and unreachable states (E^-). Then, H is a logic program that is an approximate definition of R_M such that it is expected to be able to judge if a given state is reachable for M. Note that it is in general impossible to construct a computable predicate that can always judge if a given state is reachable for M from the undecidability of the reachability problem. Finally, B is a set of clauses that define data structures, types, predicates, etc. used in E and to construct H.

We have designed a framework for our purpose. The architecture of the framework is shown in Fig. 7. An input to the framework is an equational system specification of a state machine of which we would like to extract the characteristics of the reachable states. An equational specification is written in CafeOBJ and suited for ITP. An equational specification is first translated into a rewrite theory specification written in Maude (a sibling language of CafeOBJ) with an automatic translator YAST [6]. A rewrite theory specification is suited for model checking. The Maude search command, a bounded model checker for invariants, is then used to generate reachable states that are positive examples in our learning task. Possible unreachable states that are negative examples in our learning task are generated as follows. Given a state predicate that is likely to be an invariant of a state machine concerned, we randomly generate states and then produce each of the states that does not satisfy the state predicate as an unreachable state. Those states generated as E are expressed in Maude, which should be converted into unit Horn clauses in Prolog.

Types and data structures used in an equational specification should be converted into Horn clauses in Prolog so that they can be used as B. For example, natural numbers specified in CafeOBJ is as follows:

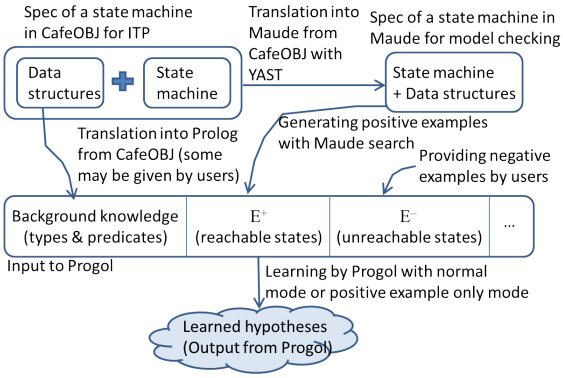

Fig. 7. Architecture of proposed method

[Nat]
op 0 : → Nat {**constr**}
op s : Nat → Nat {**constr**}

which is converted into the following Horn clauses:

pnat(0).
pnat(s(X)) :- pnat(X).

Moreover, user defined functions in equations should be converted into Horn clauses. For example, let us consider the following function:

op mk : Nat → List
eq mk(0) = 0 .
eq mk(s(X)) = s(X) | mk(X) .

where List is the type (sort) for lists of natural numbers, and nil and _|_ are the constructors of List[5]. Given a natural number n, mk(n) makes the list $n, n-1, \ldots, 0$. The function is converted into the Horn clauses:

mk(0, [0]) :- ! .
mk(s(N), [s(N) | $L1$]) :- pnat(N), mk(N, $L1$) .

In addition to E and B, what are fed into Progol also contains mode declarations. The mode declarations used in our experiments are shown in Appendix A. The predicates used as B are also shown in Appendix A with brief explanation. We have conducted several experiments on ABP and SCP with the two learning modes implemented in Progol: Normal learning mode and Learning from positive examples only. The both results are not very different. In the next section, we will show the results (learned hypotheses) of the experiments, compare them with the state patterns, and describe how to conjecture lemmas based on the learned hypotheses.

[5] In the paper, $a \mid b \mid c \mid$ nil is expressed as a, b, c.

5 Experiments

We have experimented with normal learning mode (with both positive and negative examples) and with positive example only mode (with positive examples only) to learn hypotheses from various collections of system states (from 100 to 5000 for SCP case and from 500 to 10000 for ABP case). The results with both learning modes were almost the same.

Therefore, in the rest of the section, we describe some sets of clauses extracted from various collections of reachable states with learning from positive only mode [7]. Moreover, the clauses are compared with the state patterns shown in Fig. 5 and Fig. 6 to explain which characteristics are extracted. We also describe how to conjecture lemmas from the learned hypotheses.

5.1 Simple Communication Protocol

With respect to the background knowledge describing some data types, such as Boolean values, natural numbers in Peano style and lists of natural numbers together with some auxiliary functions described in Mode Declarations for constructing clauses (the descriptions of the functions are shown in Appendix A), the experiments on SCP produced various sets of clauses from various collections of reachable states. Based on the quality of what have been obtained from each set of clauses, we have realized that the experiments on around 1000 states are good enough for our learning task.

Some constraints have been used to generate reachable states used as positive examples so as to make computational burden not too big and make learned hypotheses reasonably good. Among the constraints used are as follows. Each natural number n used in each reachable state was to satisfy the condition $1 \leq n \leq 10$. This is because otherwise natural numbers are inductively defined, forcing Progol to be loaded with too big computational burden. Each collection has been generated from a different state that is reachable from an initial state.

We have conducted multiple experiments and obtained multiple sets of learned axioms. Among them, we have selected one (called Set 1) that is most likely to be closer to the four state patterns. Set 1 has been learned by Progol with Mode declaration in Appendix A and is as follows:

$\mathrm{state}(A, B, C, D, \mathrm{c}(\mathrm{p}(A, B)), \mathrm{c}(E))$:- $\mathrm{mk}(B, D)$.
$\mathrm{state}(A, B, A, C, \mathrm{c}(\mathrm{p}(D, E)), \mathrm{c}(A))$:- $\mathrm{neg}(A, D), \mathrm{mk}(B, [B \mid C])$.
$\mathrm{state}(A, B, A, C, \mathrm{c}(\mathrm{p}(A, B)), \mathrm{c}(A))$:- $\mathrm{mk}(B, [B \mid C])$.

where $\mathrm{p}(b, n)$ denotes $\langle p, n \rangle$ and c is used to construct non-empty cells.
The clauses define the predicate that takes six arguments whose types are declared in *Mode declaration 1*. The six arguments correspond to sb, rb, d, buf, dc, and ac, respectively. The first clause in Set 1 says that if buf is the list that consists of $d, d-1, \ldots, 0$ in this order, then the head is reachable. The head also says that dc consists of the pair of sb and p. Therefore, the clause extracts the characteristics shared by *State Pattern 2* and *State Pattern 3*. The second clause

in Set 1 says that if sb is different from the first element b in the pair $\langle b, n\rangle$ of dc and buf is the list that consists of $d-1, \ldots, 0$ in this order, then the head is reachable. The head also says that rb and the Boolean value in ac are the same as sb. Therefore, the clause extracts almost all characteristics of *State Pattern 4*. The clause does not mention the second element n in the pair $\langle b, n\rangle$ of dc. The third clause in Set 1 says that if buf is the list that consists of $d-1, \ldots, 0$ in this order, then the head is reachable. The head also says that sb, rb, the first element b in the pair $\langle b, n\rangle$ of dc, and the Boolean value in ac are the same. The clause perfectly extracts the characteristics of *State Pattern 1*.

If the set $\{\mathrm{state}(\overrightarrow{x}) \text{ :- } cond_i(\overrightarrow{x}) \mid i = 1, \ldots, n\}$ of clauses perfectly defines the reachable states of a state machine concerned, $\bigvee_{i=0}^{n} cond_i(\overrightarrow{x})$ must be the strongest inductive invariant of the state machine, where $\overrightarrow{x}$ is a sequence of variables. Note that we assume that term patterns are written as part of conditions. For example, $\mathrm{state}(A, B, A, C, \mathrm{c}(\mathrm{p}(A, B)), \mathrm{c}(A)) \text{ :- } \mathrm{mk}(B, [B|C])$ is written as $\mathrm{state}(A, B, A2, C, D, E) \text{ :- } \mathrm{mk}(B, [B|C]), A2 = A, D = \mathrm{c}(\mathrm{p}(A, B)), E = C(A)$. Since such a perfect set of clauses cannot be learned in general due to the undecidability of the reachability problem, however, this is not the way we can use to conjecture lemmas from learned hypotheses. Moreover, the formula constructed by $\bigvee_{i=0}^{n} cond_i(\overrightarrow{x})$ must be too long to be used effectively, even if it is the strongest inductive invariant.

Let $cond_i(\overrightarrow{x})$ be $pre_i(\overrightarrow{x}), con_i(\overrightarrow{x}), oth_i(\overrightarrow{x})$, where $oth_i(\overrightarrow{x})$ may be void. We suppose that if $pre_k(\overrightarrow{x})$ holds, then each $cond_i(\overrightarrow{x})$ for $i \in \{1, \ldots, n\} - \{k\}$ does not hold. Then, $pre_k(\overrightarrow{x}) \Rightarrow con_k(\overrightarrow{x})$ is one possible candidate of lemma. If there exist more than one such k, say $k_1, \ldots, k_m$, then $\bigwedge_{j=1}^{m}(pre_{k_j}(\overrightarrow{x}) \Rightarrow con_{k_j}(\overrightarrow{x}))$ is one possible candidate of lemma. This is basically how we conjecture lemmas from learned axioms (or hypotheses) or state patterns, such as the four state patterns for SCP and the six state patterns for ABP.

The third axiom of the learned ones for SCP contains $rb = b, dc = \mathrm{c}(\langle b, n\rangle)$, $\langle sb, d\rangle = \langle b, n\rangle$ as part of the condition. If $rb = b$, the condition of the second axiom does not hold. The first axiom has $\langle sb, d\rangle = \langle b, n\rangle$ as part of the condition. Therefore, according to the way to conjecture lemma, we can conjecture $dc = \mathrm{c}(\langle b, n\rangle) \wedge rb = b \Rightarrow \langle sb, d\rangle = \langle b, n\rangle$ as one lemma. This lemmas is the same as the one conjectured from the four state patterns in Sect. 3.

5.2 Alternating Bit Protocol

ABP is a modified version of SCP such that its channels are unbounded. Therefore, we need to define some more complex data structures: queues of pairs of Boolean values and natural numbers - used for dc, and queues of Boolean values - used for ac. Moreover, some auxiliary functions for these data structures are also defined as shown in Appendix B. From the experiments, it was sufficient to use about 1500 reachable states for our learning task. In addition to the same constraints used to generate reachable states in the experiments for SCP, some more constraints were used. For example, each queue used in each reachable state contains at least two elements. Since more recursively defined data structures are used in ABP than in SCP, Progol spends more resources

(both computational and spatial resources) to search and compute candidates of learned hypotheses and often reaches the limitation of resources. Hence, the learned hypotheses in the experiments for ABP were not as good as those for SCP. Each set of learned hypotheses does not extract many characteristics of the reachable states of ABP. For example, let us consider the following set of clauses learned with *Mode Declaration 2* in Appendix A.

$$\begin{aligned}
&\mathrm{state}(A,B,C,D,[\mathrm{p}(A,B)\,|\,E],[A\,|\,F]) \text{ :- } \mathrm{neg}(A,C)\ .\\
&\mathrm{state}(A,B,C,D,[\mathrm{p}(C,E)\,|\,F],[A\,|\,G]) \text{ :- } \mathrm{neg}(C,H),\mathrm{succ}(E,B),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{mk}(B,[B\,|\,D]),\mathrm{memberb}(H,G)\ .\\
&\mathrm{state}(A,B,C,D,[\mathrm{p}(C,E)\,|\,F],[A\,|\,G]) \text{ :- } \mathrm{mk}(E,D),\mathrm{memberp}(\mathrm{p}(A,B),F)\ .\\
&\mathrm{state}(A,B,C,D,[\mathrm{p}(A,B)\,|\,E],[F\,|\,G]) \text{ :- } \mathrm{mk}(B,D)\ .\\
&\mathrm{state}(A,B,A,C,[\mathrm{p}(D,E)\,|\,F],[A\,|\,F]) \text{ :- } \mathrm{mk}(E,C)\ .\\
&\mathrm{state}(A,B,C,D,[\mathrm{p}(A,B)\,|\,E],[A\,|\,F]) \text{ :- } \mathrm{mk}(B,[B\,|\,D])\ .\\
&\mathrm{state}(A,B,C,D,[\mathrm{p}(E,F)\,|\,G],[A\,|\,H]) \text{ :- } \mathrm{neg}(A,E),\mathrm{succ}(F,B),\mathrm{mk}(B,[B\,|\,D])\ .
\end{aligned}$$

We asked Progol to learn the definition of predicate *state* for ABP as we did for SCP. The first clause partially extracts the characteristics of *State Pattern 1* shown in Fig. 6. The second clause partially extracts the characteristics of *State Pattern 6*. The third clause partially extracts the characteristics of *State Pattern 2* and *State Pattern 3*. The fourth clause partially extracts the characteristics of *State Pattern 2* and *State Pattern 3* as well. The fifth clause partially extracts the characteristics of *State Pattern 1*. The sixth clause partially extracts the characteristics of *State Pattern 6*. But, some very important characteristics on *dc* and *ac* cannot be extracted by any clauses learned.

We suspected that we did not use enough background knowledge so that some very important characteristics on *dc* and *ac* could be extracted in the experiment in which the last set of clauses were learned. The important characteristics on *dc* is that *dc* contains at most one gap such that two adjacent pairs $\langle b,i\rangle$ and $\mathrm{next}(\langle b,i\rangle)$ appear at most once in *dc*, where $\mathrm{next}(\langle b,i\rangle) = \langle \neg b, i+1\rangle$. We have added two predicates gap0 and gap1 whose definitions are found in Appendix A. Then, the following set of clauses have been learned by Progol:

$$\begin{aligned}
&\mathrm{state}(A,B,C,D,[\mathrm{p}(A,B)\,|\,E],[A\,|\,F]) \text{ :- } \mathrm{neg}(A,C),\mathrm{gap0}(\mathrm{p}(A,B),E)\ .\\
&\mathrm{state}(A,B,C,D,[\mathrm{p}(A,B)\,|\,E],[C\,|\,F]) \text{ :- } \mathrm{mk}(B,D),\mathrm{gap0}(\mathrm{p}(A,B),E)\ .\\
&\mathrm{state}(A,B,A,C,[\mathrm{p}(D,E)\,|\,F],[A\,|\,G]) \text{ :- } \mathrm{neg}(A,D),\mathrm{succ}(E,B),\mathrm{mk}(B,[B\,|\,C])\ .\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{gap1}(\mathrm{p}(A,B),F)\ .\\
&\mathrm{state}(A,B,A,C,[\mathrm{p}(A,B)\,|\,D],[A\,|\,E]) \text{ :- } \mathrm{mk}(B,[B\,|\,C]),\mathrm{gap0}(\mathrm{p}(A,B),D)\ .
\end{aligned}$$

The third clause more precisely extracts the characteristics of *State Pattern 6* showing in Fig. 6, including the important characteristics of *dc*. Since the third clause is the only one in which $\mathrm{gap1}(\mathrm{p}(A,B),F)$ holds, we can conjecture a lemma from this clause. $\mathrm{gap1}(\mathrm{p}(A,B),F)$ can be rephrased as follows: $dc = ps1@(p1,p2,ps2) \wedge p1 \neq p2 \wedge (p3 \in ps1 \Rightarrow p3 = p1) \wedge (p4 \in ps4 \Rightarrow p4 = p2)$,

where @ is the concatenation function of queues. Therefore, we can conjecture the following:

$(dc = ps1@(p1, p2, ps2) \wedge p1 \neq p2 \wedge (p3 \in ps1 \Rightarrow p3 = p1) \wedge$
$(p4 \in ps4 \Rightarrow p4 = p2)) \Rightarrow (p2 = \langle sb, d \rangle \wedge buf = d - 1, \ldots, 0)$

This lemma is very useful to prove that ABP enjoys the reliable communication protocol.

Honestly speaking, it is not easy to systematically come up with gap0 and gap1 from the formal specification of ABP. This has something to do with what is called *Predicate Invention* [8]. It is one piece of our future work to systematically discover some predicate, such as gap0 and gap1 that do not explicitly appear in formal specifications. We anticipate that Meta-interpretive learning and its implementation *Metagol* [9] will help us do so.

6 Related Work

ML has been used to find lemmas in ACL2 [10]. Their tool can calculate the similarity between the current proof and other proofs in a given proof library containing many existing proofs that have already been proved. Their tool finds an existing proof whose structure is most likely to be similar to that of the current proof, and proposes lemmas for the current proof that are constructed from the lemmas used for the existing proof.

ILP has been successfully integrated with model checking [11]. Although a model checker systematically finds a counterexample demonstrating that a system specification (or a model) does not enjoy a property, human users are supposed to revise the system specification so that the revised version can enjoy the property. They propose a way to systematically conduct such a revision for the system specification with an ILP system that uses a counterexample found by a model checker as a negative example, a witness constructed according to the property concerned as a positive example, and a system specification as the background knowledge.

Our way to use an ILP system is different from the two above mentioned studies. We use an ILP system to extract the characteristics of the reachable states of a state machine as learned clauses (hypotheses) from which lemmas could be conjectured.

7 Conclusion and Future Work

We have reported on case studies in which Progol has been mainly used to extract the characteristics of the reachable states of M_{SCP} and M_{ABP}. We have compared them with the state patterns we had manually learned from our ITP experiences for SCP and ABP. We have not formally proved that the four and six state patterns exactly cover all reachable states of M_{SCP} and M_{ABP}, respectively. But, our experiences on conjecturing lemmas based on the state patterns say that

they are most likely to do so and very useful for lemma conjecture. It would be possible to generate different state patterns by combining and dividing those state patterns. From a lemma conjecture point of view, however, the four and six state patterns for SCP and APB are very useful. The learned hypotheses (a set of clauses) for SCP is very close to the four state patterns if not exactly the same. If gap0 and gap1 are used, the learned hypotheses for ABP is also close to the six state patterns. Otherwise, the learned hypotheses for ABP do not capture the important characteristics on *dc* appearing in *State Pattern 6*.

We have also described how to conjecture lemmas based on the learned hypotheses. This demonstrate that our approach is likely to be promising for lemma conjecture.

But, there are lots more things left to do. In our experiments, gap0 and gap1 have been provided by human beings. It is not trivial to come up with such predicates because they do not explicitly appear in an equational specification written in CafeOBJ. It is called predicate invention to come up with new predicates from a program or specification in which those predicates are not explicitly used. *Metagol* has implemented a mechanism with which predicate invention is doable. One piece of our future work is to come up with a method in which *Metagol* is mainly used to invent new predicates, such as gap1 and gap1. We need to conduct more case studies in which our approach is applied to other protocols and algorithms, such as *Paxos* and the *Chandy-Lamport snapshot algorithm*. Another piece of our future work is to come up with how to select the best one among several sets of learned axioms without knowing any oracles in advance and/or how to integrate multiple sets of learned axioms so as to obtain a better one.

Acknowledgement The authors wish to thank anonymous referees who commented on a draft of this paper. They also wish to thank some participants of ILP 2015, especially Stephen Muggleton and Gerson Zaverucha, who gave us useful comments on our presentation at ILP 2015 that helped us revise the draft of this paper.

References

1. Ogata, K., Futatsugi, K.: Proof score approach to analysis of electronic commerce protocols. IJSEKE **20**(2) (2010) 253–287
2. Barrett, C.: Decision procedures: An algorithmic point of view. J. Autom. Reasoning **51**(4) (2013) 453–456
3. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: 6th FMOODS. (2003) 170–184
4. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. In: Algebra, Meaning, and Computation. (2006) 596–615
5. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. J. Log. Program. **19/20** (1994) 629–679
6. Zhang, M., Ogata, K., Nakamura, M.: Translation of state machines from equational theories into rewrite theories with tool support. IEICE Trans. Inf. & Syst. **94-D**(5) (2011) 976–988

7. Muggleton, S.: Learning from positive data. In: 6th ILP. (1996) 358–376
8. Stahl, I.: Predicate invention in ILP - an overview. In: ECML-93. (1993) 313–322
9. Muggleton, S.H., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. Machine Learning **100**(1) (2015) 49–73
10. Heras, J., Komendantskaya, E., Johansson, M., Maclean, E.: Proof-pattern recognition and lemma discovery in ACL2. In: 19th ILP. (2013) 389–406
11. Alrajeh, D., Russo, A., Uchitel, S., Kramer, J.: Integrating model checking and inductive logic programming. In: 21st ILP. (2011) 45–60

Appendix A: Mode Declarations and Predicate Descriptions

toppqu(A, B)	Pair B is on top of queue A of pairs
topbqu(A, B)	Boolean value B is on top of queue A of Boolean values
mk(A, B)	B is the ordered list of natural numbers from number A to 0
neg(A, B)	Boolean value A is the negation of Boolean value B
fst(A, B)	A is the first element of pair B
snd(A, B)	A is the second element of pair B
succ(A, B)	Natural number B is the successor of natural number A
memp(A, B)	Pair A is in queue B of pairs
memb(A, B)	Boolean value A is in queue B of Boolean values

Mode declaration 1
:- modeh$(1, \text{state}(+bool, +pnat, +bool, +nlist, \text{c}(\text{p}(+bool, +pnat)), \text{c}(+bool)))$?
Mode declaration 2
:- modeh$(1, \text{state}(+bool, +pnat, +bool, +nlist,$
$[\text{p}(+bool, +pnat) \mid +pqueue], [+bool \mid +bqueue]))$?

next$(\text{p}(B1, N), \text{p}(B2, \text{s}(N)))$:- neg$(B1, B2)$.
gap0$(P, [])$:- bnpair(P) .
gap0$(P, [P \mid T])$:- gap0(P, T).
gap1$(P, [])$:- bnpair(P) .
gap1$(P1, [P2 \mid T])$:- $(P1 \setminus== P2, \text{next}(P2, P1), \text{gap1}(P1, T)); \text{gap0}(P1, T)$.