Master's Research Project Report

An Investigation of Machine Learning and a consideration on its application to Theorem Proving

1310065 Ho, Dung Tuan

Supervisor : Professor Kazuhiro Ogata
Main Examiner : Professor Kazuhiro Ogata
Examiners : Professor Kunihiko Hiraishi
Associate Professor Toshiaki Aoki

School of Information Science
Japan Advanced Instituted of Science and Technology

February, 2016

Contents

1 Introduction

Systems Verification logically checks if systems satisfy desired properties to make them reliable. The techniques used are largely classified into *Model Checking* and *(Interactive) Theorem Proving*. This project focuses on *Interactive Theorem Proving (ITP)* that often requires *eureka* steps. One typical eureka step is to discover a non-trivial lemma, called *Lemma Conjecturing*. Some techniques have been proposed such that lemmas can be systematically or automatically discovered, and successfully applied to some specific applications [4, 5, 6, 7, 8]. None of those techniques, however, can discover all lemmas for all possible proof problems (not only mathematical but also engineering ones, i.e., systems verification. In general, it is necessary to understand proof targets profoundly to some extent to discovery non-trivial lemmas. Proof targets are systems and/or system behaviours in Systems Verification. Human users often rely on some information to conjecture such lemmas. This information characterises some important aspects of reachable states of systems. But, it is time-consuming to extract the information from a large amount of reachable states. We predict that *Machine Learning* [2] can help to do so since the technique can be applied to big data. In ITP, Machine Learning may extract some patterns from a large number of reachable states. The patterns expresses some characteristics of the reachable states such that they can help human users to get better understanding of not only the reachable states but also the system's state machine. Then, the understanding can hopefully leads the human users to conjecture some non-trivial lemmas for some specific proof problems. The aim of the project is to learn some advanced techniques of machine learning, confirming that the technique can be used to extract useful information from reachable states of systems such that the information helps human users to conjecture non-trivial lemmas.

Our ultimate goal is to systematically assist the human users through one of most intellectual activities in ITP, Conjecturing Lemmas. There are many researchers in Systems Verification who has conducted or proposed many approaches or methods to archive the goal in some specific problems. The same goal but our approach is mainly related to Machine leaning such that applying a machine learning technique into ITP in which it can extract the information from a large amount of reachable states, hopefully it is useful to lemma conjecture. In Machine Learning aspect, the approach is considered to a learning task in which it consists of problem instances and extracted patterns. The problem instances are reachable states of a state machine of an under verification system and extracted patterns are the reachable states' characteristics. Therefore, first and foremost, the objects needs to be formally represented. In other words, we must to formally represent our database consists of the reachable states (*input*) and the characteristics (*output*) which we want to obtain. Unfortunately, most machine-learning techniques/systems represent input in form of *propositional logic* or *attribute-value* representations such that the database is represented in tabular form [3]. In other words, the database is expressed on tables in which each row corresponds to an example, and each column corresponds to an attribute. Furthermore, the examples have exactly one value specified for each of the attributes. Unfortunately, most ITP systems use some variant of first-order logic as

its representation in which there are some kinds of structured data recursively defined, e.g. list, queue. It is impossible to transform the definitions of such structured data to propositional ones [19].

Fortunately, there is a research area called *Inductive Logic Programming (ILP)* [9] stayed in the intersection of *Machine learning* and *Logic Programming* [20] such that ILP inherits the goal of Machine learning: synthesis logic programs, acquire knowledge from data and ILP also inherits the way of data representations using *clausal logic* [21], a subset of first order logic. In general, our learning task is to characterize reachable states of a state machine of a under verification system such that the extracted characteristic are useful to construct some lemmas which can be used to discharge some non-trivial verification cases. The learning task has two main ingredients related to different research areas: ILP and ITP in which ITP provides databases consisting of not only examples but also background knowledge and ILP extracts knowledge also in form of first order logic with respected to the database based on theory which is combination of logic theory and machine learning.

To demonstrate our approach, we have conducted several experiments on two communication protocols: *Alternating Bit Protocol* (ABP) and *Simple Communication Protocol* (SCP). We had successfully conducted the verifications for such protocols, with respected to *Reliable Communication Property*, including conjecture lemmas. To conjecture such lemmas, we relied on a source, called State Pattern, obtained by manually observing sequences of reachable states. The comparisons between the State Patterns and the knowledge acquired from an ILP system are shown in the report to illiterate that the knowledge express the characteristics of the reachable states. Moreover, we also show a consideration that is possible to get better understanding the system or to conjecture some lemmas from the knowledge.

1.1 Outline of the Report

In Chapter 2, we first give a overview of Systems Verification, then Algebraic Formal Method with CafeOBJ. 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. 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

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

[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.

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). Because of the representation problem, we introduce ILP in the second part of Chapter 2. Moreover, we give an overview of an implemented approach in Progol, an state-of-art ILP system, which we use to conduct the experiments on two the Communication Protocols.

After Chapter 2, the rest of the text can be structured in two parts. Each is a chapter. Chapter 3 - Framework, we describe our contribution, a framework which is a combination of Machine learning and Interactive Theorem Proving such that ITP provides a system specifications suited for a verification on an ITP system, i.e. CafeOBJ [1], as an input to our framework. By using other tools and methods to extract and generate the components of a ILP learning task, which are suitable to our purpose, then we can get the hypothesised clauses.

At the last part, chapter 4 - Application, we shows the applications of our framework to two Communication Protocols [16], Alternating Bit Protocol and Simple Communication Protocol. Each application consists of several experiment with differences of database used. But first, we show their verification on CafeOBJ such that we need to conjecture some lemmas to complete the verification. Conjecturing lemma is one of most intellectual activity in ITP, we must to understand the under consideration system and the proof in some extent. Usually, we rely on some reliable source to get better understanding. For the case of two the Protocols , we rely on the series of pictures showing reachable states. But the number of these pictures is finite such that any reachable state can be classified into one of the pictures. We call the pictures are State Patterns. Each hypothesised clause we get from the experiments will be compared with these State Patterns.

And finally, the last chapter, we summary the contents and show some related work.

1.2 Bibliographical Note

Some of the basic theory of this work has been published before. Following list contains the key articles:

- CafeOBJ
  - CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, Volume 6 of AMAST Series in Computing, World Scientific, 1998 by Razen Diaconescu and Kokichi Futatsugi

  - Proof Scores in the OTS/CafeOBJ Method, Formal Methods for Open Object-Based Distributed Systems, Volume 2884 of the series Lecture Notes in Computer Science, pages 170-184 by Kazuhiro Ogata and Kokichi Futatsugi
  - Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method, Essays Dedicated to Joseph A. Goguen, 2006, Pages 596-615 by Kazuhiro Ogata and Kokichi Futatsugi

- Inductive Logic Programming
  - Inductive Logic Programming: Theory and methods, The Journal of Logic Programming, Volumes 19-20, 1994, Pages 629-679 by Stephen Muggleton and Luc de Raedt

- Communication protocols
  - Distributed Algorithms, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA by Nancy A. Lynch

1.3 Acknowledgement

This work would not have been possible without of many people. I thank my friend Nguyen, Thang Ngoc and Professor. Zhang Min. Nguyen, Thang Ngoc gave me the importance of data representation in Machine Learning. And Professor. Zhang Min insisted the current representation of reachable states, in form of logic terms. Without the key information, I cannot find Inductive Logic Programming, which leads to many potential goals for may research.

I thank my supervisor Professor. Kazuhiro Ogata. He provided me with many ideas and with hints on the Master Project. The discussions with Prof. Ogata provided me with a better understanding of the area of Interactive Theorem Proving. Moreover, he gives me the motivation and support to continue to research on this area.

2 Preliminaries

2.1 Systems Verification

2.1.1 Formal Verification

Nowadays, computers become more popular and important in our lives. Many computer softwares and hardware systems have been developed for countless tasks on a wide-ranged applications, some of them critical. They are the programs that run modern economies and mistakes in hardware and software design can have serious economic and commercial repercussions. Flaws in such systems have caused many serious consequences such as loss of human life and money. The more critical the system, the more stringent and rigorous the design and implementation process should be. However, computer systems have become so complex and more complex in the future that the way of exhaustive manual testing cannot be suitable for cover the system behaviours on extreme situation.

Because of the increasing of complexity of systems, it requires that the step of designing in software development must be considered in advance such that a designer of the system must to verify about the design, in the sense that the design and implementation of the system satisfies its specification, i.e. the system does what it is supposed to do and nothing undesirable happens. This relates to a brad area of research is known as *Formal Verification* in computer science and effectively how it works today. There are many researches and activities which have been conducting tools which are used to check the systems. The mathematical underpinnings of the tools ensure (in theory at least) that the truth of individual assertions can be combined into an assertion for the whole system. In other words, they ensure *compositionality*.

The verification problem is unsolvable [22]. Even if the problem were in theory solvable for a given system, there are two common reasons that make a solution impractical: (1) the behaviour of the system or parts of the system may not be easy to control to a mathematical treatment, (2) the system may simply be too complex for verification to be feasible. This still leaves a large class of systems for which a verification attempt can be successful. Such systems include computer programs as well as electronic circuits and network protocols.

The systems one is attempting to verify exist in the physical world. A physical confirmation of mathematical properties is of course impossible. Formal verification techniques instead work with mathematical descriptions (or models) of the system. This is made possible by the uniquely logical nature of computer- based systems even at low levels of abstraction. This ensures that the assumptions justifying a mathematical abstraction of the physical process are never so unrealistic that the verification is not useful.

2.1.2 Interactive Theorem Proving

Theorem proving is one of popular techniques which dominates proof-based approaches to Formal Verification. Here the system under consideration is modelled as a set of mathematical definitions in some formal mathematical logic. The desired properties of

the system are then derived as theorems that follow from these definitions. The method of derivation or proof borrows heavily from standard results in mathematical logic. However, techniques have been developed to automate much of this process by using computers to handle obvious or tedious steps in the proof. Unfortunately, the proof system of a theorem prover for a system of practical size can be extremely large [24]. Furthermore, the generated proofs can be large and difficult to understand.

The undecidability of validity in first order logic implies that automated theorem proving using a first order theory cannot fully automated [23]. Moreover, because theorem provers are often used to reason inductively, the theorems to be proved need to be formulated as formulas involving mathematical induction. In practice, the theoretical results require that a human must to interact with the theorem prover in advance to derive non-trivial theorem. The term *Interactive Theorem Proving* is used to denote a theorem proving system that requires human intervention.

The user interacts with the theorem prover in a variety ways. In general, there are two most important interactions requiring the user. (1) First and foremost, the user is responsible for representing and encoding the problem domain of the system so that useful results can be derived by the proof system. (2) The user also plays an important role to guide the theorem prover in its search for a proof. The guidance takes the form of setting immediate lemmas that should be proved on the way to the final proof [11], as well as selecting heuristic and strategies at various steps of the proofs. Related to our experience with Interactive Theorem Proving on CafeOBJ, we usually consider to the possibility to generate counterexample such that some lemmas need to be conjectured or just continually applied Case Splitting. *Lemma Conjecturing* is one of the most intellectual activities which require the user have an understanding of not only the current proof but also the under consideration system in some extent to result the final proof. The user needs to decide *what* needs to prove and *how* to prove it. An interactive theorem proving system is a parallel process between human mathematical reasoning and computer support reasoning.

2.2 Algebraic Formal Methods with CafeOBJ

2.2.1 Overview

Algebraic Formal Methods is one of the major formal methods such that *Algebraic Specification* seeks to systematically develop more sufficient a model of a system by formally defining type of data and mathematical operations on those data types. And, the model abstracts the system behaviors, formalized on the data types, allowing for automation restricting operations to this limited set of behaviors and data types. An algebraic specification achieves these goals by defining one or more data types, and specifying a collection of functions that operate on those data types. These functions can be divided into two classes: (1)*constructor operators* - functions that create or initialize the data elements, or construct complex elements from simpler ones, (2)*defined operators* - functions that operate on the data types, and are defined in terms of the constructor functions.

CafeOBJ is a recent developments of algebraic specification show that evolution of systems can be neatly modeled by *rewriting logic* algebraically in which CafeOBJ pro-

vides an unified basis for system design, specification, and verification. CafeOBJ is also a multi-paradigm specification language which is a modern successor of the most noted algebraic specification language OBJ. CafeOBJ adopts *rewriting logic* as its underlying logic. *Rewriting logic* is a simple computational logic that covers a wide-range of (potentially non-deterministic) methods of replacing subterms of a formula with other terms. What is considered is *rewriting systems* which consist of a set of objects and relations on how to transform those objects.

To formally verify that a system satisfies a desired property with CafeOBJ, the system is first formalized as a state machine M together with a state predicate p expressing the property. CafeOBJ 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. The state machine M consists of a set S of states that includes the initial states I and a binary relation T over states, called transitions. Each system state is characterized by observable values observed with functions called observers, and each transition is expressed as functions defined in term of changes of observable values with equations. 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 $s \in R_M$. What we have to do is to prove that p is an invariant of M. 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).

2.2.2 Specification and Verification of TAS - A mutual exclusion Protocol

TAS is a system in which multiple processes execute a parallel program which is used to explain how to specify a system and how to verify that it has properties. The program supposedly solves the mutual exclusion problem, namely that is allows at most one process to enter the critical section, where resources such as I/O devices that have to be accessed by at most one process at any given time are used. TAS written in an Algol-like language is shown in Fig. 1 (a). A process repeatedly executes this program, namely if the process at Critical Section (or at cs) executes $lock := \text{false}$, it moves to the next loop. TAS uses *lock*, which is shared by all processes, to control processes such that there is at most one process in 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.

2.2.2.1 Observational Transition Systems

We assume that there exists a universal state space called Υ. We also suppose that each data type used has been defined beforehand, including the equivalence between two data values v_1, v_2 denoted by $v_1 = v_2$. A system is modelled by observing, from the outside of each state of Υ, only quantities that are relevant to the system and how to change the quantities by state transition. An OTS (observational transition system) can be used to model a system in this way. An OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

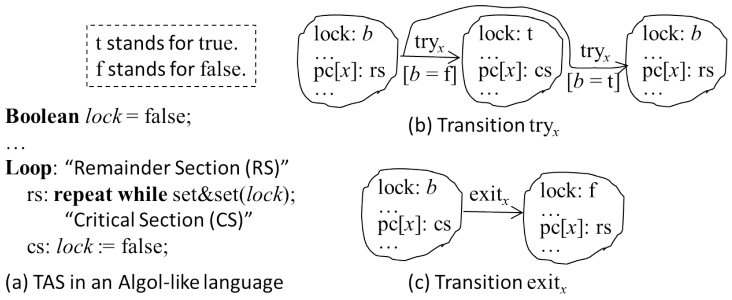


Figure 1: TAS and a state machine M_{TAS} formalizing TAS

- $\mathcal{O}$: A set of observable values. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \to \mathcal{D}$, where $\mathcal{D}$ us a data type and may be different for each observable value. Given an OTS $\mathcal{S}$ and two values $v_1, v_2 \in \Upsilon$, the equivalence between two states, denote by $v_1 =_{\mathcal{S}} v_2$, w.r.t. $\mathcal{S}$ is defined as $v_1 =_{\mathcal{S}} v_2 \stackrel{\mathrm{def}}{=} \forall o \in \mathcal{O}.o(v_1) = o(v_2)$.

- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subset \Upsilon$.

- $\mathcal{T}$: A set of conditional transition rules. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon/ =_{\mathcal{S}}$ on equivalence classes of Υ w.r.t. $=_{\mathcal{S}}$. Let $\tau(v)$ be the representative element of $\tau([v])$ for each $v \in \Upsilon$ and it is called *the successor state* of v w.r.t. τ. The condition c_τ for a transition rule $\tau in \Upsilon$, which is a predicate on states, is called *the effective condition*. The effective condition is supposed to satisfy the following requirement: given a state $v \in \Upsilon$, if c_τ is false in v, namely τ is not *effective* in v, then $v =_{\mathcal{S}} \tau(v)$.

An OTS is described in CafeOBJ. Observable values are denoted by CafeOBJ observations, and transition rules by CafeOBJ actions.

An execution of S is an infinite sequence $v_0, v_1, \dots$ of states satisfying:

- *Initiation*: $v_0 \in \mathcal{I}$.

- *Consecution*: For each $i \in \{0, 1, \dots\}$, $v_{i+1} =_{\mathcal{S}} \tau(v_i)$ for some $\tau \in \mathcal{T}$.

A state is called reachable w.r.t. $\mathcal{S}$ iff it appears in an execution of $\mathcal{S}$. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all the reachable states w.r.t. $\mathcal{S}$.

All properties considered in this section are invariants, which are defined as follows:

$$\texttt{invariant}\ p \stackrel{\mathrm{def}}{=} (\forall v \in \mathcal{I}.p(v)) \wedge (\forall v \in \mathcal{R}_{\mathcal{S}}.\forall \tau \in \mathcal{T}.(p(v) \implies p(\tau(v)))),$$

which means that the predicate p is true in any reachable state of $\mathcal{S}$. Let $\boldsymbol{x}$ be all free variables except for one for states in p. We suppose that invariant p is interpreted as $\forall \boldsymbol{x}.(\texttt{invariant}\ p)$.

2.2.2.2 Specification

Two kinds of observation values and two kinds of transition rules are used to specify TAS, which are as follows:

- Observable values
  - $lock$ denotes the boolean value shared by all process, which is initially empty;
  - pc_i ($i \in Pid$) denotes the section of a command that process i will be execute next, which is initially rs.
- Transition rules
  - try_i ($i \in Pid$) denotes the command corresponding to Remainder Section (rs).
  - $exit_i$ ($i \in Pid$) denotes the command corresponding to Critical Section (cs).

Pid is a set of process IDs.

The OTS modeling the system is specified in module `TAS`, which imports modules `LOC` and `PID`. The signature of `TAS` is as follows:

```
[Sys]
-- any initial state
op init : -> Sys {constr}
-- observations
op lock : Sys -> Bool
op pc : Sys Pid -> Loc
-- actions
op try : Sys Pid -> Sys {constr}
op exit : Sys Pid -> Sys {constr}
```

The state space Υ is represented by the sort `Sys`, observable value $lock$ and pc_i by observations `lock` and `pc`, respectively, and transition rules try_i and $exit_i$ by actions `try` and `exit`, respectively. Constant `init` denotes any initial state of OTS. A operator (started with `op`, declared a attribute `constr`, is a constructor operator such that the operator define recursively/inductively the set of terms which constitute a sort, in this module, sort `Sys`.

Equations defining the three actions show, where `S` is a CafeOBJ variable for `Sys`, and `P` and `Q` for `Pid`. Action `try` is defined with equations as follows:

```
op c-try : Sys Pid -> Bool
eq c-try(S,P) = (pc(S,P) = rs and not lock(S)) .
--
eq lock(try(S,P)) = true .
ceq pc(try(S,P),Q) = (if P = Q then cs else pc(S,Q) fi) if c-try(S,P) .
ceq try(S,P) = S if not c-try(S,P) .
```

Operator `c-try` denotes the effective condition of transition rule try_i.

Action `exit` is defined with equations as follows:

```
op c-exit : Sys Pid -> Bool
eq c-exit(S,P) = (pc(S,P) = cs) .
eq lock(exit(S,P)) = false .
ceq pc(exit(S,P),Q) = (if P = Q then rs else pc(S,Q) fi) if c-exit(S,P) .
ceq exit(S,P) = S if not c-exit(S,P) .
```

Operator `c-exit` denotes the effective condition of transition rule $exit_i$.

2.2.2.3 Verification

We verify TAS has the following invariant:

- Invariant 1 *Mutual Exclusion*

$$\mathrm{pc(s,i)} = \mathrm{cs\,and\,pc(s,j)} = \mathrm{cs\,implies\,i} = \mathrm{j}. \tag{1}$$

  This invariant means that at most one process can executes Critical Section at any given time. To prove the invariant, we need one more invariant, which is as follows:

- Invariant 2

$$\mathrm{pc(s,i)} = \mathrm{cs\,implies\,lock(s)}. \tag{2}$$

How to Construct Proof Scores We briefly describe how to construct proof scores of invariants [10]. Suppose that all predicates and action operators takes only states as their arguments for simplicity. Invariants are often proved by induction on the number of transition rules applied. Suppose that we prove that the system has invariant $p_1(s)$ by induction on the number of transition rules applied, where s is a free variable for states.

It is often impossible to prove invariant $p_1(s)$ alone. Suppose that it is possible to prove invariant $p_1(s)$ together with $n-1$ other predicates. Let the $n-1$ other predicates be $p_2(s), \ldots, p_n(s)$. That is, we prove invariant $p_1(s) \wedge \cdots \wedge p_n(s)$. Let $p_(s)$ be $p_1(s) \wedge \cdots \wedge pn(s)$.

Let us consider an inductive case in which it is shown that any transition rule denoted by CafeOBJ action operator a preserves $p(s)$. To this end, it is sufficient to show $p(s) \implies p(a(s))$. This formula can be proved compositionally. The proof of the formula is equivalent to the proofs of the n formulas:

$$\begin{aligned} p(s) &\implies p_1(a(s)) \\ &\cdots \\ p(s) &\implies p_n(a(s)) \end{aligned}$$

Moreover, it suffices to prove the following n formulas, if possible, instead of the previous n formulas:

$$p_1(s) \implies p_1(a(s))$$
$$\cdots$$
$$p_n(s) \implies p_n(a(s))$$

But, some of them may not be proved because their inductive hypotheses are too weak. Let $p_i(s) \implies p_i(a(s))$, where $0 \leq i \leq n$, be one of such formulas. Let SIH_i be a formula that is sufficient to strengthen the inductive hypothesis $p_i(s)$. SIH_i can be $p_{i_1}(s) \wedge \cdots \wedge p_{i_k}$, where $1 \leq i_1, \ldots, i_k \leq n$. Then, all we have to do is to prove $(SIH_i \wedge p_i(s)) \implies p_i(a(s))$.

Besides, we may have to split the case into muptiple subcases in order to prove $(SIH_i \wedge p_i(s)) \implies p_i(a(s))$. Suppose that the case is split into l subcases. The l subcases are denoted by l formulars $case_1^i, ..., case_l^i$, which should satisfy $(case_1^i \vee, dots, \vee case_l^i) = true$. Then, the proof can be replaced with the l formulas:

$$(case_1^i \wedge SIH_i \wedge p_1(s)) \implies p_1(a(s))$$
$$\cdots$$
$$(case_l^i \wedge SIH_i \wedge p_l(s)) \implies p_1(a(s))$$

SIH_i may not be needed for some subcases.

Proof scores of invariants are based what has been discussed. Let us consider that we write proof scores of the n invariants discussed. We first write a module, say `INV`, where $pi(s)(i = 1, \ldots, n)$is expressed as a CafeOBJ term as follows:

```
op inv1 : H -> Bool
...
op invn : H -> Bool
eq inv1(S) = p1(S) .
...
eq invn(S) = pn(S) .
```

where `H` is a hidden sort and `S` is a CafeOBJ variable for `H`. Term `pi(S) (i = 1, ...,n)` denotes $p_i(s)$.

We are going to mainly describe the proof of the ith invariant. Let init denote any initial state of the system. To show that $p_i(s)$ holds in any initial state, the following proof score is written:

```
open INV
     red invi(init) .
close
```

We next write a module, say `ISTEP`, where two constants `s`, `s'` are declared, denoting any state and the successor state after applying a transition rule in the state, and the predicates to prove in each inductive case are expressed as a CafeOBJ term as follows:

```
op istep1 : -> Bool
...
op istepn : -> Bool
```

```
eq istep1 = inv1(s) implies inv1(s') .
...
eq istepn = invn(s) implies invn(s') .
```

In each inductive case, the case is usually split into multiple subcases. Suppose that we prove that any transition rule denoted by CafeOBJ action operator a preserves $p_i(s)$. As described, the case is supposed to be split into the l subcases $case^i_1, ..., case^i_l$. Then, the CafeOBJ code showing that the transition rule preserves $p_i(s)$ for $case^i_j (j = 1, \ldots, l)$ looks like this:

```
open ISTEP
-- Declare constants denoting arbitrary objects.
-- Declare equations denoting caseij .
-- Declare equations denoting facts if necessary.
eq s' = a(s) .
red istepi .
close
```

Constants may be declared for denoting arbitrary objects. Equations are used to express $case^i_j$. If necessary, equations denoting facts about data structures used, etc. may be declared as well. The equation with *s'* as its left-hand side specifies that s' denotes the successor state after applying the transition rule denoted by `a` in the state denoted by s.

If `istepi` is reduced to true, it is shown that the transition rule preserves $p_i(s)$ in this case. Otherwise, we may have to strengthen the inductive hypothesis in the way described. Let $\mathrm{SIH_i}$ be the term denoting SIH_i. Then, instead of $istep_i$, we reduce the term $(SIH_i \,\mathrm{and}\, \mathrm{inv_i(s)}) \,\mathrm{implies}\, \mathrm{inv_i(s')}$, or SIH_i implies $istep_i$.

Proof Scores In module `INV`, the following operator is declared and defined:

```
op inv1 : Sys Pid Pid -> Bool
op inv2 : Sys Pid -> Bool
eq inv1(S,P,Q) = ((pc(S,P) = cs and pc(S,Q) = cs) implies (P = Q)) .
eq inv2(S,P) = (pc(S,P) = cs implies lock(S)) .
```

In the module, constants `i` and `j` for `Pid` are declared.

In module `ISTEP`, the following operator denoting the predicate to prove in each inductive case is declared and defined:

```
op istep1 : -> Bool
op istep2 : -> Bool
eq istep1 = inv1(s,p,q) implies inv1(s',p,q) .
eq istep2 = inv2(s,p) implies inv2(s',p) .
```

Fig. 2 shows a snip of a proof tree for invariant inv1(s). Given a state s and a process identifier k, try(s, k) is the state obtained by applying transition try_k in s, exit(s, k) is the

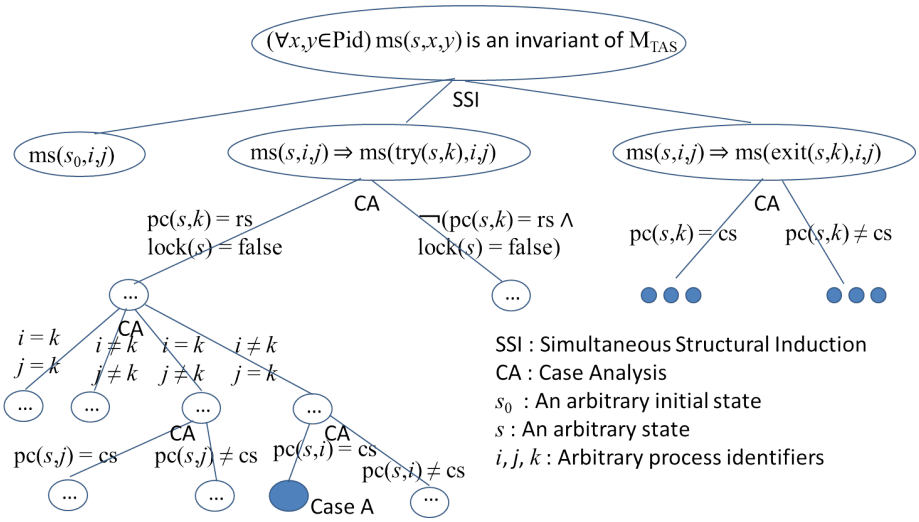


Figure 2: A snip of a proof tree that mx(s) is an invariant of M_{TAS}

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-cases is to show $\mathrm{inv1}(s_0, i, j)$, $\mathrm{inv1}(s, i, j) \Rightarrow \mathrm{inv1}(\mathrm{try}(s, k), i, j)$ and $\mathrm{inv1}(s, i, j) \Rightarrow \mathrm{inv1}(\mathrm{exit}(s, k), i, j)$, 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[3].

Let us consider the case marked Case A in Fig. 2 in which $\mathrm{inv1}(s, i, j) \Rightarrow \mathrm{inv1}(\mathrm{try}(s, k), i, j)$ reduces false. Therefore, Case A needs a lemma. Let $\mathrm{inv2}(s, i)$ be such a lemma. We will soon describe how to conjecture the lemma. $\mathrm{inv2}(s, i) \Rightarrow (\mathrm{mx}(s, i, j) \Rightarrow \mathrm{inv1}(\mathrm{try}(s, k), i, j)$ reduces true, discharging Case A, provided that we prove that $(\forall x \in \mathrm{Pid})\, \mathrm{inv2}(s, x)$ is an invariant of M_{TAS}. The proof needs $\mathrm{inv1}(s, i, j)$ as a lemma. This is why we use simultaneous structural induction. The following code is the proof score of Case A using `inv2` as a lemma.

```
open ISTEP .
-- arbitrary objects
op k : -> Pid .
-- assumptions
-- eq c-try(s,k) = true .
eq pc(s,k) = rs .
eq lock(s) = false .
--
eq i = k .
```

[3]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.

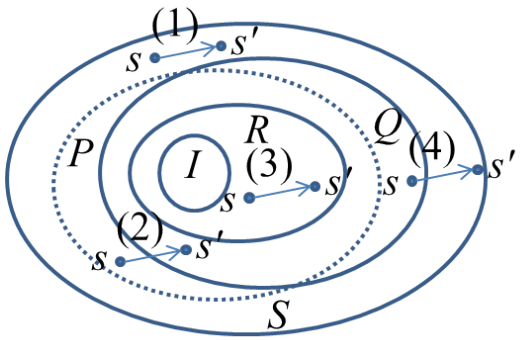


Figure 3: Some possible situations when proving that p is an invariant of M

```
eq (j = k) = false .
eq pc(s,j) = cs .
eq s' = try(s,k) .
red inv2(s,k) implies istep1 .
close
```

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 $\mathrm{lem1}(s, i)$.

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.

Finding an inductive invariant q (or conjecturing a lemma q) is one of the most intellectual activities in ITP[4]. This activity requires human users to profoundly understand

[4] 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.

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).

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.

2.3 Inductive Logic Programming (ILP)

2.3.1 The ILP learning task

The task of ILP is concerned with the inference of theories (hypotheses) from observation (data) in machine learning such that the set of hypotheses is in form of a language which is a subset of first order logic. In which, it is related to first-order theory induction. A first-order theory can be represented by a logic program, hence the word "programming" in ILP. Note that the first-order language and therefore also the class of logic programs is provably as computationally powerfully as the Turing machine, in other words, a logic program can implement any computable function. A special feature of the usually considered ILP learning task is that the learner is also provided a *background knowledge* , which is also a first-order theory.

By using logic programming languages as the representation mechanism for hypotheses and observations, ILP can overcome two main limitation of classical machine learning techniques: the use of a limited knowledge representation formalism (essentially a propositional logic) and the difficulties in using substantial background knowledge in learning

process. The first limitation is important because many domains of expertise can be expressed in first-order logic, or a variant of first-order logic and not in a propositional one. For example, logic synthesis is one such domain. Most of logic programs cannot be defined using only propositional logic. The second limitation is also crucial because one of the well-established findings of artificial intelligence is that the use of background knowledge is essential for archiving intelligent behaviour. From computational logic, ILP inherits its representation formalism, its programming language semantics and various well-established techniques. In contrast to most other approaches to inductive learning, ILP is interested in properties of inference rules, in convergence of algorithms and in the computational complexity of procedures. Many ILP systems benefit from using the results of computational logic. ILP extends the theories and practice of computational logic by investigating induction rather than deduction as the basic mode of inference. Whereas computational logic theory describes deductive inference of logic formulae provided by the user, ILP theory describes the inductive inference of logic programs from observations and background knowledge. In general setting, the ILP's learning task is defined as Definition 1.

Definition 1 *Given background knowledge B and observations O. The observations $O = O^+ \wedge O^-$ consists of positive observation O^+ and negative observation O^-. The aim is then to find a hypothesis H such that the following condition hold.*

- ***Prior Satisfiability.*** $B \wedge O^- \not\models \square$
- ***Posterior Satisfiability.*** $B \wedge H \wedge O^- \not\models \square$
- ***Prior Necessity.*** $B \not\models O^+$
- ***Posterior Sufficiency.*** $B \wedge H \models O^+$

To formalize fully an ILP task, the relation between B, H, E^+ and E^- need to be defined. This depends on several factors which include the chosen *logic programming language* (e.g. definite or normal logic programs), *logic programming semantics* (e.g. well-founded or stable). ILP methods often require some form of bias on the solution search space to restrict the computation to hypotheses. Forms of bias include *language bias* and *search bias* (e.g. top-down or bottom-up). A *Mode Declaration* is a form of language bias that specifies the syntactic form of the hypotheses that can be learned. It contains head declarations and body declarations that describe predicates that may appear, the desired input and output and number of instantiations, e.g. *recall*.

2.3.2 Mode-Directed Inverse Entailment and Progol

Progol, an state-of-art ILP system, uses an approach to the general problem of ILP called Mode-Directed Inverse Entailment (MDIE). The general problem of ILP can be summarized as follows. Given a background knowledge B and examples E find the simplest consistent hypotheses H such that

$$B \wedge H \models E$$

If we rearrange the above using the law of contraposition we get the more suitable form

$$B \wedge \bar{E} \models \bar{H}$$

In general B, H and E can be arbitrary logic program but if we restric H and E to be single Horn clauses, $\bar{H}$ and $\bar{E}$ above will be ground skolemized unit clauses. If $\bar{\perp}$ is the conjunction of ground literals which are true in all models of $B \wedge \bar{E}$, we have

$$B \wedge \bar{E} \models \bar{\perp} \models \bar{H}$$

and so

$$H \models \perp$$

A subset of the solutions for H can then be found by considering those clauses which θ-subsumes $\perp$. The complete set of candidates for H could in theory be found from those clauses which imply $\perp$. As yet Progol does not attempt to find a fuller set of candidates (bypassing the undecidability of implication between clauses with bounds on the number of resolution steps in the Prolog interpreter). Prolog searches the latter subset of solution for H that θ-subsumes $\perp$.

In general $\perp$ can have infinite cardinality. Progol uses the head and body mode declarations together with other settings to build the most specific clause and hence to constrain the search for suitable hypotheses.

A mode declaration has either the form *modeh(n, atom)* or *n, atom*, where n, the recall, is an integer greater than zero or "*" and atom is a ground atom. Terms in the atom are either normal or place-marker. A normal term is either a constant or function symbol followed by a bracketed tuple of terms. A place-maker is either +type, -type, #type where type is a constant.

The recall is used to bound the number of alternative solutions for instantiating the atom. A recall of "*" indicates all solutions - in practice a large number. +type, -type, #type correspond to input variables, output variables and constants respectively.

Progol imposes a restriction upon the placement of input variables in hypothesized clauses. Suppose the clause is written as $h :- b1, \ldots, b_n$ where h is the head atom and b_i, $1 \leq i \leq n$ is either of +type in h or -type are body of atoms. Then every variable of +type in any atom b_i is either of +type in h or -type in some atom b_j where $1 \leq j \leq i$. This imposes a quasi-order on the body atoms and ensures that the clause is logically consistent in its use of input and output variables.

3 The Framework of Tools used

We have designed a framework which is a collection of the tools used to characterize the reachable states of a state machine. The architecture of the framework is shown in Fig. 4. In general, conducting a learning process in machine learning usually require the following tasks: data collecting, data requirement/data transformation, feature selection and learning. Fortunately, we use ILP as the machine learning technique for our purpose, we do not need to consider to the feature selection since the representation of data in ILP is first-order logic, not propositional logic as other classical machine learning techniques/systems which are using. Therefore, our framework mainly focus to three tasks: data collection, data refinement (data transformation) and learning. Our framework takes a CafeOBJ system specification in form of equational logic as its input. Every required components for a ILP learning task will be generated from the specification. Unfortunately, we uses Progol, a state-of-art ILP system for our learning task, which uses a Prolog interpreter. Prolog is a logic programming language in form of clausal logic, a subset of first order logic. Basically, we need to transform our data to the suitable form for Progol doing the task of characterization.

3.1 Data Collection

As mentioned in Definition 1, we need to collect background knowledge B and the examples E consisting of positive examples E^+ and negative examples E^-. In Fig. 4, the system specification of a state machine consists of two parts: data structure definitions and state machine specification by using the data structures. For example, natural numbers are recursively defined in term of Peano style as follows:

```
[Nat]
op 0 : -> Nat {constr}
op s : Nat -> Nat {constr}
```

In Peano style, natural numbers are considered as zero and non-zero numbers. The constructor operator `0` is a constant standing for zero. And the other constructor operator `s` is a function defining a non-zero number such that the function takes a natural number as argument and return to a its successor number. For example, `s(0)` is interpreted as number `1`, `s(s(s(s(s(0)))))` is interpreted as number `5`, etc. In general, natural numbers are a primitive data type and supported in many systems or programming languages. But when it is defined in Peano style as recursively structure, it is hard to be learned by a classical machine learning system since a recursively data structure are usually costly at computation resource and it is impossible to transform to a propositional form. Progol does not need to transform to the propositional one since Prolog and CafeOBJ share the same first order logic theory. Moreover, Progol also has such problem at computation resource limitation but we can limit the size of a natural number term during the learning task. Another structure data usually used in a CafeOBJ system specification is list and one possible definition of list of natural numbers in CafeOBJ as follow:

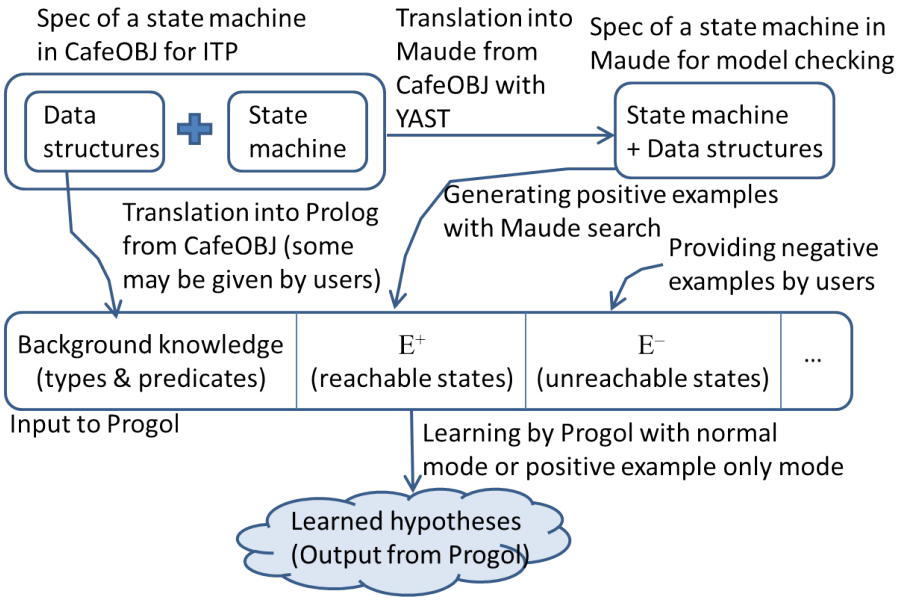


Figure 4: Architecture of proposed method

```
pr(NAT)
[List]
--
op nil : -> List {constr}
op __ : Nat List -> List {constr}
--
op hd : List -> Nat
op tl : List -> List
eq hd(X:Nat L:List) = X .
eq tl(X:Nat L:List) = L .
```

The definition import the above definition of natural numbers. The list are recursively defined as empty list (represented by constructor operator `nil`) and non-empty lists represented by constructor operator __. The operator __ takes two arguments, one is natural number and other is a list in order, then returns to a non-empty list. For examples, `0 s(0) s(s(0)) nil` is a non-empty list. There are two other operators `hd` and `tl` which are considered as helper functions such that operator `hd` will take a non-empty list as its argument then returns to the top number of the list but `tl` returns to a list excepting the top number. For example, we have `0 s(0) s(s(0)) nil`, `hd` returns to `0` and `tl` returns to `s(0) s(s(0)) nil`.

We can directly use the data structure definitions as the background knowledge B but the definitions are in form of CafeOBJ language, we need to transform them to Prolog language, a set of Horn clauses of clausal logic at the next task, data transformation.

Since our learning task is to characterize reachable states of a state machine, the reachable states are considered as positive examples E^+ and the unreachable states are con-

sidered as negative examples E^-. Unfortunately, it is impossible to generate any system states from a CafeOBJ system specification since the system specification in from of equational logic suited for theorem proving. In general, model checking is also a formal verification in systems verification is related to system states during verification. Fortunately, Maude a model checker which use a sibling language with CafeOBJ such that its specification in form of rewriting theory suited for model checking. Although some existing strategies to translate an equational theory specification to a rewriting system one are inefficient and rarely used for model checking in practice, Zhang Min et al. [12] had developed a tool, called YAST, implementing a strategy for the translation. The strategy is proved that translated specifications by the strategy are more efficient than those by existing strategies. Let consider to the specification of TAS's state machine in Section 2.2.2, we have the specification generated by YAST. And we use the Maude *search* command to generate the system states of TAS. For example, we want to collect 1000 reachable states of TAS with respected to three processes `p1`, `p2` and `p3` from the initial state by the following command:

```
search [1000] init =>* S:State
```

then we have the following sequence output.

```
Solution 1 (state 0)
states: 1  rewrites: 12 in 0ms cpu (0ms real) (352941 rewrites/second)
S:State --> lock : false
(pc[p1]: rs)
(pc[p2]: rs)
(pc[p3]: rs)

Solution 2 (state 1)
states: 2  rewrites: 65 in 0ms cpu (0ms real) (371428 rewrites/second)
S:State --> lock : true
(pc[p1]: cs)
(pc[p2]: rs)
(pc[p3]: rs)

Solution 3 (state 2)
states: 3  rewrites: 68 in 0ms cpu (0ms real) (309090 rewrites/second)
S:State --> lock : true
(pc[p1]: rs)
(pc[p2]: cs)
(pc[p3]: rs)
...
```

At the beginning, `Solution 1 (state 0)`, variable `lock` is `false` and all three processes `p1`, `p2` and `p3` are at Remainder Section. If `p1` go to Critical Section, we have

`Solution 2 (state 1)`, and if `p2` go to Critical Section, we have `Solution 3 (state 2)`. All these states are reachable states of TAS. But for unreachable states E^- generation, we cannot use Maude since only reachable states are generated from a state machine specification if the state machine is well-defined. One possible solution to generate such system states, we need to assign randomly the corresponding values to each observable data of a system state. Then, an invariant express some characteristics of reachable states of a state machine is applied to select to unreachable one. For example, let consider an unreachable state in which all three processes are at critical section, such unreachable state can be obtained since the invariant, has been proved in Section 2.2.2, returns *false*. It is difficult to collect such unreachable states for the system such that we do not obtain any invariant. Fortunately, the using ILP system, Progol, is implemented an learning mode such that it can learn from a data lacking of negative examples E^-.

In summary, 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. 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.

3.2 Data Tranformation

Background knowledge B and examples E are in form of Horn clauses in which each clause has at most one positive literal. In other words, the clause has at most one literal in its head part. And Prolog is used as the logic language programming to represent background knowledge B, examples E and the hypotheses H learn from E with respected to B. We can collect B from the structure data definitions of a system specification and collect E^+ from the state machine definition of a state machine (the state machine definition is translated to the one suited from model checking) but E^+ is more complex.

Let consider to the above definition of natural numbers in Peano style which is converted into the following Prolog clauses:

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

This two clauses defines a type `pnat` in the background knowledge B, such information to declare about the structure of the clauses/hypotheses we want to learn. We will discuss in detail in the next section. The logic programming language Prolog has a specific syntax for list/queue, but any kind of objects can be an element of such list. In other words, there is no constraint in the type of elements. For example, `[a | 1 | cafeob | maude`

`| s(0)]` is a Prolog list/queue. However, in a CafeOBJ system specification, there is only one type of elements appearing on a queue/list. Then, to define such kind of queues/lists in Prolog, we need to check the type of each element in a list. Let consider to the above definition of list of natural numbers, we can convert it into the following Prolog clauses:

```
nlist(nil).
nlist([X | L]) :- pnat(X), nlist(L)

hd([X | L], X).
tl([X | L], L).
```

At the second clause defining predicate `nlist`, its body check if the top element `X` is a natural number and recursively check if the tail `L` is a list of natural number. There are two other clauses without body defining `hd` and `tl`, respectively. Each the clause has two arguments in which the first one is its input and the second one is its output.

Because each reachable state generated by Maude is an informal output. We need to pick all Maude term to construct a system state which will be expressed as a fact in Prolog language. Fact is a Horn clause without body but a literal in it head. For example, we have the following TAS reachable state.

```
  Solution 1 (state 0)
states: 1  rewrites: 12 in 0ms cpu (0ms real) (352941 rewrites/second)
S:State --> lock : false
(pc[p1]: rs)
(pc[p2]: rs)
(pc[p3]: rs)
```

It can be converted to the following Horn clauses:

```
dict(pc123,p1,rs).
dict(pc123,p2,rs).
dict(pc123,p3,rs).
state(false,pc123).
```

Since we have three processes and each process will be located at a section in a moment, we need to express such information. One possible solution is to use structure of dictionary, e.i. at the initial state, `p1` is at `rs`, we use the dictionary `pc123` storing the key `p1` and the value`rs`. And the dictionary `pc123` has three different pairs of key and value. `pc123` is the second argument of predicate `state` and the first argument is the value of variable `lock`. The definition of predicate `state` is the set of clauses which we ask Progol to construct from example E with respected to Background knowledge B.

3.3 Learning

In section Data Collection and Data Transformation, we have collected the background knowledge B and the examples E from a CafeOBJ system specification. We have enough sufficient components for an ILP learning. In our case, we use Progol implementing the Mode-Directed Inverse Entailment such that we need to define some mode declarations in the input before feed into Progol. As mentioned, Mode declaration is the information that Progol requires to construct the most specific clause for each example. Each most specific clause constraints the search space consisting of candidate hypotheses. The following is set of the mode declarations using to learn a definition of predicate `state`.

```
:- modeh(1,state(#bool,+dictionary))?
:- modeb(1,dict(+dictionary,+process,cs))?
:- modeb(1,dict(+dictionary,+process,rs))?
:- modeb(1,dict(+dictionary,-process,cs))?
:- modeb(1,dict(+dictionary,-process,rs))?
:- modeb(1,+process = +process)?
```

There are two kind of mode declaration: head mode declaration `modeh` and body mode declaration `modeb`. There is only one head mode declaration but several body mode declarations. The first argument of each mode declaration is `recall` and the second one is the atom which can be appeared in the hypothesis clauses, followed by its structure. For example, in `modeh`, we have the second argument `state(#bool,+dictionary)`, then each clause in the learn hypothesis set will have exactly a literal `state` followed by `true` or `false` and a constant of type `dictionary`. The following clause is one of possible clauses returned by Progol:

```
state(true, A) :- dict(A,B,cs), dict(A,C,cs), B = C.
```

The above clause expresses the characteristic expressed by the invariant which is proved in Section CafeOBJ such that when the variable `lock` is `true`, only one process is allowed to enter Critical Section.

Beside background knowledg B, examples E and the mode declarations, Progol also requires us to provide some of runtime parameter settings. For instance, to use the learning from positive only learning mode, we declare the following clause.

```
:- set(posonly)?
```

Learning from positive example only is inspired by a classical example such that children can learn natural language grammars almost exclusively from positive example. Stephen Muggleton at el. [15] proposed that within a Bayesian framework, not only grammars, but also logic programs are learnable with arbitrarily low expected error from positive examples only. In addition, he show that the upper bound for expected error of a learner which maximises the Bayes' posterior probability when learning from positive example is within a small additive term of one which does the same from a mixture of positive and

negative examples. The approach was implemented together the normal learning mode in Progol.

Since Progol use a Prolog interpreter which is a implementation of SLD-resolution such that a proof is searched by deep first search and from left to right, there are some proofs which cannot to terminate. The non-termination problem happen when the provide background knowledge is not satisfied a example, but the interpreter still checks and retrieves the search. Therefore, Progol needs to constraint the computation resource, e.g. the maximum depth `h` (default is 30) of any proof, the maximum depth `r` (default is 400) of resolutions (unifications), the number of search nodes in the hypothesis space (default is 200), the maximum number of atoms in the body of any hypothesised clause, etc.

3.4 Framework of Assessment

Our task is characterization of reachable states of a state machine. The complexity and number of characteristics of reachable states are explosive because of enormous amount of observable values specifying the reachable states. And each observable value is an instance of a recursively defined data structure, this makes the learnt clauses obtained from our framework is hard to understand by human-being. Then, we can not evaluate that the set of clauses are correct in showing that a system state is reachable or not. Moreover, we cannot recognize if which clause is interesting to conjecture lemma.

We can check these clauses by preparing multiple databases consisting of reachable states and unreachable states, then applying some statistical techniques to evaluate the correction of each clauses. But it does not show if the clauses can be used in ITP or not. We have conducted the experiments on ABP and SCP in which the case studies were successfully verified that the protocols satisfy Reliable Communication Property. And the verifications require several lemmas to be conjectured to completed the proofs in CafeOBJ. To show that the results of my framework is useful to let the user get better understanding about the system, we will use ABP and SCP to make comparison.

During the verifications, we carefully consider each reachable state of the protocol, then manually draw a picture of the state to capture the characteristics. Finally, we obtained a series of pictures for each protocol such that any reachable state can be classified into one of the pictures. And we reply on the pictures as a source to get better understanding, then it is possible to conjecture several lemmas from them. We call the series of pictures are *State Patterns*. There are four state patterns for SCP as Figure 8 and six state patterns for ABP as 9. The way to conjecture lemmas from these state patterns is described in detail in the next section.

To evaluate the clauses obtained from the characterization tasks of SCP/ABP, the comparison between the state patterns and the learnt clauses will be made such that there are two comparison directions: soundness (Figure 5) and completeness (Figure 6). For the soundness direction, we compare whether if the characteristics captured in the state patterns are also expressed in the learnt clauses. And for the reversed direction, completeness, we compare whether if the characteristics expressed in the learnt clauses are captured in the state patterns. Depend on how many state patterns/learnt clauses capture/express the characteristics, we can judge the quality of each experiment. More-

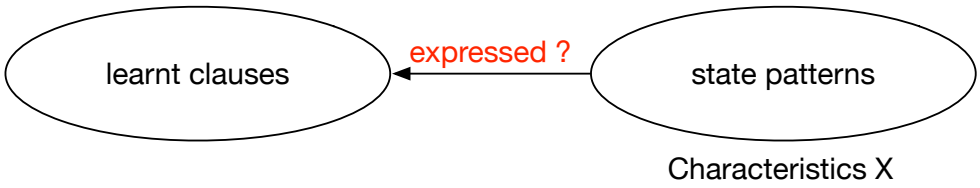


Figure 5: Comparison between the state patterns with the learnt clauses

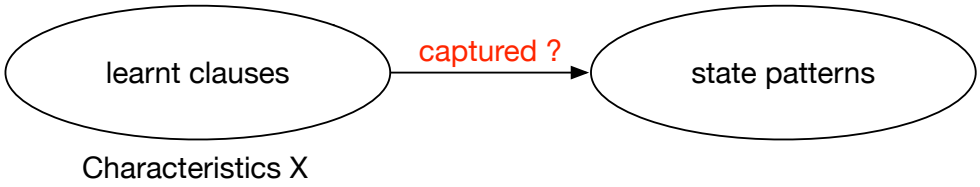


Figure 6: Comparison between the learnt clauses with the state patterns

over, it is impossible to extract automatically some characteristics in state patterns/learnt clauses, we need to manually extract and consider each characteristic from them.

Hypotheses specified in the set of Horn clauses returned from each experiment heavily rely on a set of reachable states. That means, the number of instances of a characteristic decide if the characteristic appears at the results. Moreover, because of the limitation of resource computation, that means the size of each term specifying the observable values for reachable states is too big, the characteristic may not be extracted. Then, we have conducted multiple experiments with respected to multiple database with different at setting. And each experiment will result a set of clauses.

The number of experiments for each case study are from 20 - 40 experiments and each will return about 5 clauses. There are some clauses repeatedly appearing in multiple experiments. That means the probabilistic values of the clauses which expressing some characteristics of reachable states are higher than the other. In other word, the clauses cover more reachable states appearing in the databases than other. This relates to completeness property of an ILP learning task such that $B \wedge H \models E^+$. By relying on this property, we can reduce the number of clauses, also the number of experiments, which we need to consider on the assessment part.

When applied to all set of clauses returned from all conducted experiments, Algo. 1 will reduce the number of sets of clauses. If a clause from a set is new, then the set is added to the final sets.

Algorithm 1 Reducing number of sets of clauses from all conducted experiments

E_{in} := all sets of clauses
$E_{out} := \emptyset$
$C_{out} := \emptyset$
for all $e \in E_{in}$ **do**
    for all $c \in e$ **do**
        if $c \notin C_{out}$ **then**
            add e to C_{out}
            $E_{out} := E_{out} \cup e$

4 Application

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 our framework to extract some characteristics of the reachable states of their state machines formalizing the protocols. Before showing the experiments on these two protocols, we introduce the specification and verifications of these protocols in CafeOBJ. During the verifications, we conjectured some lemmas to discharge some non-trivial cases when writing their proof scores. To complete the task of conjecture lemmas, we reply on a series of snapshots capturing all characteristics of their reachable states. Then, we call these snapshots as *state patterns*. To conjecture lemmas for a non-discharged case, we need carefully consider the assumption described in the case, then map it to a corresponding state pattern. We must to combine the characteristics showing in the state pattern with the proof of the case related to the rewrite theory in CafeOBJ to conjecture a useful lemma. Our framework chracterizes the reachable states with respected to the state machine and background knowledge provided by a CafeOBJ system specification such that the learnt hypothesised clauses chareterize some approximate state patterns of the state patterns we were using in the verifications. Therefore, we will make a comparison between the learnt clauses one each experiment with the state patterns such that ILP or machine learning is useful to characterize the reachable states of a state machine. And we also consider if it is possible to conjecture some useful lemmas from such clauses.

4.1 Specification and Verification of Communication Protocols

Communication Protocol is a class of algorithms designed to manage the data transmitted between senders and receivers through unreliable channels such that packets may be duplicated and dropped. There are many such algorithms proposed so far. In general, Sender wants to send a sequence of data to Receiver. Let consider the sequenced data expresses a sequence of ordered natural numbers started with zero. Packets are sent from Sender to Receiver through a unreliable channel, called `data channel` as shown in Figure 7 , e.i. internet network, the unreliability means each packet on the channel may be lost, duplicated or modified. Each packet contains a copy of the number in the numbers which

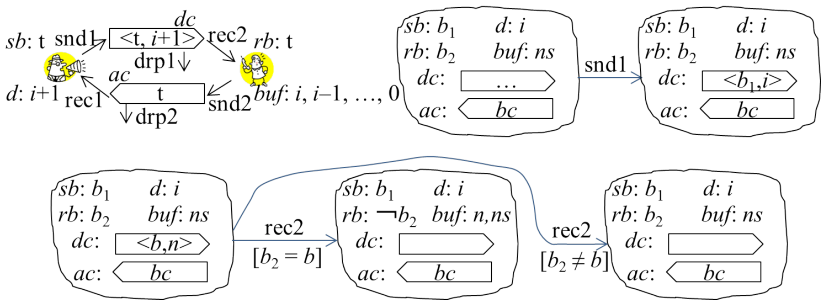


Figure 7: SCP and part of a state machine M_{SCP} formalizing SCP

Sender wants to send to Receiver.

Because of the unreliability of the channel, Sender repeatedly send multiple duplication of the sending packet to the channel, hopefully one of them can reach Receiver. But, when Receiver has received the packet, Sender must to know that and then start to send the packets of a new number. One possible solution is that Receiver needs to send back an acknowledgement message to Sender such that it can notify Sender. This communication requires another channel from Receiver to Sender. Unfortunately, this channel is also unreliable. This opens the same problem with the delivery of packets from Sender to Receiver such that Sender must notify to Receiver that the message has been get. In this section, we take into account two simplified versions of TCP: SCP and ABP, where SCP is a simplified version of ABP. They uses a bit on each site, Sender and Receiver, to control the communication process. Figure 7 shows the sketch of a communication protocol for SCP and ABP, we constrain the unreliable actions happened in the channels to only duplication and/or drop.

One property which both these protocols should enjoys is *Reliable Communication Protocol* such that whenever the sending number is i, the data up to i or $i-1$ has been successfully delivered to Receiver from Sender without any duplication and drop. In two these next section, we shows the verifications of SCP and ABP one by one.

4.1.1 Simple Communication Protocol

Fig. 7 shows a snapshot (a state) of a state machine M_{SCP} formalizing SCP in which the capacity of a channel is 1 such that the channel only has at most one packet/message at a time . Therefore, a packet/message in the channel only be dropped. Since Sender wants to send data to Receiver, it starts to put the packet consist of a pair of a bool value and a number into the data channel, called *dc*. The number is the current number which Sender is delivering and the bool value is the value of Sender's bit. Action `snd1` is that Sender repeatedly put its packets into *dc*. For example, at the beginning, the pair is `false` and `0`. Now, *dc* has the packet, then Receiver gets the packet by action `rec2` in which Receiver compares the bool value with its bit. If they are satisfied its condition, Receiver's bit will be updated by its compliment and the number is store to a buffer. While Sender repeatedly puts its packets into *dc* by action `snd1`, Receiver also repeatedly puts its acknowledgement messages into *Acknowledgement Channel*, called *dc*, by action

`snd2` such that each message consists of the current value of Receiver's bit. The same process as action `rec2`, Sender gets the message from *ac* by `rec1`, if any. Sender's bit will be updated and the sending number will updated instead by the next if the condition is satisfied. The behaviours of the protocol is controlled by these bits in an alternating mechanism such that the protocol is satisfied *Reliable Communication Protocol*. The property is proved as an invariant of SCP by a formal verification in CafeOBJ.

We first describe the basic data types used to specify the state machine of SCP. The sorts and the corresponding data constructors are as follows:

- Bool denotes the sort of Boolean values.
- Nat denotes the sort of Natural Numbers. These numbers are defined in Peano style as same as the above definition.
- BNPair denotes the sort of pairs of a Boolean value and a natural number. Given a boolean value `false` and a natural number `n`, `< false ; n >` is a BNPair pair.
- PCell denotes the sort of *dc*. Given a pair `< false ; n >`, then `c(< false ; n >)` is a channel has a packet, otherwise, it is `pempty`. Both `c` and `pempty` are constructor operators of PCell.
- BCell denotes the sort of *ac*. `c(false)` is a channel has a message, otherwise, it is `bempty`. Both `c` and `bempty` are constructor operators of BCell.
- List denotes the sort of list of natural numbers.

Six kinds of observable values and eight kinds of transitions rules are used to specify SCP, which are as follows:

- Observable values
  - $cell1$ ($cell1 \in PCell$) denotes *dc*, which is initially empty or `pempty`.
  - $cell2$ ($cell2 \in BCell$) denotes *ac*, which is initially empty or `bempty`.
  - sb ($sb \in Bool$), $Bool$ is a build-in module in CafeOBJ, denotes Sender's bit, which is initially `false`.
  - sb ($sb \in Bool$) denotes Receider's bit, which is initially `false`.
  - nxt ($nxt \in PNat$) denotes the sending number at Sender, which is initially zero.
  - buf ($buf \in List$) denotes Receiver's buffer storing the received number from Sender, which is initially empty or `nil`.
- Transition rules
  - $send1$ denotes the action of Sender repeatedly putting its packets into *dc*.
  - $send2$ denotes the action of Receiver repeatedly putting its messages into *ac*.

- *rec*1 denotes the action of Sender getting a message from *ac* and may update its values if any
- *rec*2 denotes the action of Receiver getting a message from *dc* and may update its values if any
- *drop*1 denotes the action of removing a packet from *dc* if any
- *drop*2 denotes the action of removing a packet from *ac* if any

Transition `rec2` is defined as follows:

```
op c-rec2 : Sys -> Bool .
eq c-rec2(S) = not(cell1(S) ~ pempty) .

ceq cell1(rec2(S)) = pempty if c-rec2(S) .
eq  cell2(rec2(S)) = cell2(S) .
eq  sb(rec2(S)) = sb(S) .
ceq rb(rec2(S))  = (if rb(S) ~ fst(get(cell1(S)))
then not(rb(S)) else rb(S) fi)
if c-rec2(S) .
eq  nxt(rec2(S)) = nxt(S) .
ceq buf(rec2(S)) = (if rb(S) ~ fst(get(cell1(S)))
then (snd(get(cell1(S))) | buf(S))
else buf(S) fi)
if c-rec2(S) .
ceq rec2(S)  = S if not(c-rec2(S)) .
```

The operator `c-rec2` denotes the effective condition of any transition rule denoted by `rec2`. `c-rec2(S)` means mean that in a state `S`, Receiver will check if *dc* (`cell1(S)`) is non-empty. If the condition holds, Receiver's bit (`rb(rec2(S))`) and buffer (`buf(rec2(S))`) of the next state `rec2(S)` are updated.

We briefly describe the proof scores showing that no duplication and drop in the buffer, which means SCP enjoys *Reliable Communication Protocol*. This can be expressed by the following invariant rcp:

$$(sb(S) = rb(S)\, implies\, mk(nxt(S)) = (nxt(S)|buf(S))\, and\, (not(sb(S) = rb(S))\ implies\ mk(nxt(S)) = buf(S)). \tag{3}$$

where `mk` is predicate using to generate a list of number from a specific number as the first argument. The list is in decrement order, the predicate `mk` is defined as follows:

```
eq mk(0) = 0 nil .
eq mk(s(X)) = s(X) mk(X) .
```

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. 8, 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:

$$\begin{aligned} &not(cell1(S) = pempty) and\, rb(S) = fst(get(cell1(S)))\, implies \\ &\qquad next(S) = snd(get(cell1(S))). \end{aligned} \tag{4}$$

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.

It is impossible to prove rcp alone.We need totally 4 more invariants, which are as follows:

$$\begin{aligned} &not(cell2(S) = bempty)\, implies \\ &(sb(S) = get(cell2(S))\, or\, rb(S) = get(cell2(S))). \end{aligned} \tag{5}$$

$$\begin{aligned} &not(cell1(S) = pempty)\, and\, rb(S) = fst(get(cell1(S)))\, implies \\ &\qquad sb(S) = fst(get(cell1(S))). \end{aligned} \tag{6}$$

$$\begin{aligned} &not(cell1(S) = pempty)\, and\, not(cell2(S) = bempty) \\ &(sb(S) = get(cell2(S))\, or\, not(rb(S) = fst(get(cell1(S))))). \end{aligned} \tag{7}$$

In general, The proof of (3) uses (4), (5) and (6) as lemmas. The proofs of (5) and (7) use (6) as a lemma, the proof of (7) also uses (5) as a lemma. The proofs of (4) uses (7) as a lemma. The proofs of (6) uses (7) as a lemma.

4.1.2 Alternating Bit Protocol

SCP is a simplified version of ABP such that ABP's channels are unbounded. Therefore, the unreliable property is not only drop but also duplication. To keep the verification to not be so complex, we specify those channels as queues such that the drop and duplication happens on only the top element of a queue. These actions nondeterministically occur when a channel contains packets/messages.

The data types used to specify the state machine of ABP are the same as SCP but in stead of PCell and BCell, we use two new data types to specify the unbounded channels:

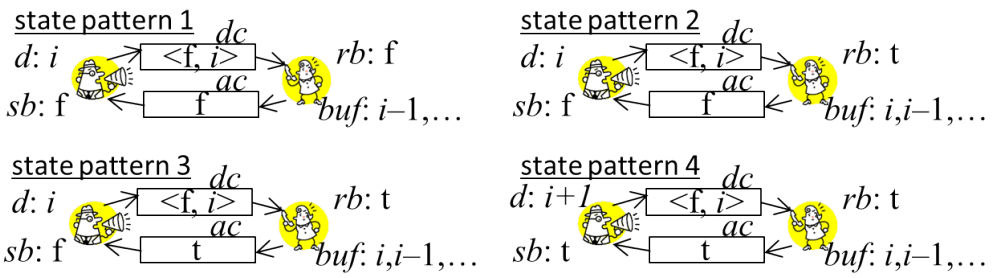


Figure 8: Four state patterns of M_{SCP}

- DChan denotes the sort of queues of pairs of a boolean value and a natural number for Data Channel. Given two pairs `< false ; n1 >` and `< false ; n2 >` where `n1` and `n2` are numbers, we have a queue `< false ; n1 > , < false ; n2 >` in which `_,_` is a constructor operator of DChan.
- AChan denotes the sort of queues of boolean values for Acknowledgement Channel. Given two boolean values `b1` and `b2`, we have a queue `b1 , b2` in which `_,_` is a constructor operator of AChan

Similar as SCP, ABP has six kinds of observable values and two more kinds of transition rules used to specify, which are follows:

- Observable values
  - $d-chan$ ($d-chan \in DChan$) denotes dc, which is initially `pempty`.
  - $a-chan$ ($a-chan \in AChan$) denotes ac, which is initially `pempty`.
  - sb ($sb \in Bool$), $Bool$ is a build-in module in CafeOBJ, denotes Sender's bit, which is initially `false`.
  - sb ($sb \in Bool$) denotes Receider's bit, which is initially `false`.
  - nxt ($nxt \in PNat$) denotes the sending number at Sender, which is initially zero.
  - buf ($buf \in List$) denotes Receiver's buffer storing the received number from Sender, which is initially empty or `nil`.
- Transition rules
  - $send1$ denotes the action of Sender repeatedly putting its packets into the end of queuedc.
  - $send2$ denotes the action of Receiver repeatedly putting its messages into the end of queueac.
  - $rec1$ denotes the action of Sender getting a message from the top of queue ac and may update its values if any

- *rec*2 denotes the action of Receiver getting a message from the top of queue *dc* and may update its values if any
- *drop*1 denotes the action of removing the top packet of queue *dc* if any
- *drop*2 denotes the action of removing the top message of queue *ac* if any
- *dup*1 denotes the action of duplicating the top packet of queue *dc* if any
- *dup*2 denotes the action of duplicating the top message of queue *ac* if any

Transition `rec2` is defined as follows:

```
op c-rec2 : Sys -> Bool .
  eq c-rec2(S) = not(d-chan(S) ~ empty) .

  ceq d-chan(rec2(S)) = get(d-chan(S)) if c-rec2(S) .
  eq  a-chan(rec2(S)) = a-chan(S) .
  eq  sb(rec2(S))  = sb(S) .
  ceq rb(rec2(S))
      = (if rb(S) ~ fst(top(d-chan(S)))
      then not fst(top(d-chan(S))) else rb(S) fi)
      if c-rec2(S) .
  eq  nxt(rec2(S))  = nxt(S) .
  ceq buf(rec2(S))
      = (if rb(S) ~ fst(top(d-chan(S)))
      then (snd(top(d-chan(S))) buf(S)) else buf(S) fi)
      if c-rec2(S) .
  ceq rec2(S)        = S if not c-rec2(S) .
```

The operator `c-rec2` denotes the effective condition of any transition rule denoted by `rec2`. `c-rec2(S)` means mean that in a state `S`, Receiver will check if *dc* (`d-chan(S)`) is non-empty. If the condition holds, Receiver's bit (`rb(rec2(S))`) and buffer (`buf(rec2(S))`) of the next state `rec2(S)` are updated.

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. 9, and lemmas can be conjectured from the six state patterns. Finally, we need totally 10 more invariants, which are as follows:

$$\begin{aligned} &not(a - chan(S) = empty)\, implies \\ &((sb(S) = top(a - chan(S)))\, or\, (rb(S) = top(a - chan(S)))). \end{aligned} \tag{8}$$

$$\begin{aligned} &(not(d - chan(S) = empty)\, and\, rb(S) = fst(top(d - chan(S)))) \\ &\qquad implies \\ &(sb(S) = fst(top(d - chan(S)))\, and\, next(S) = snd(top(d - chan(S)))). \end{aligned} \tag{9}$$

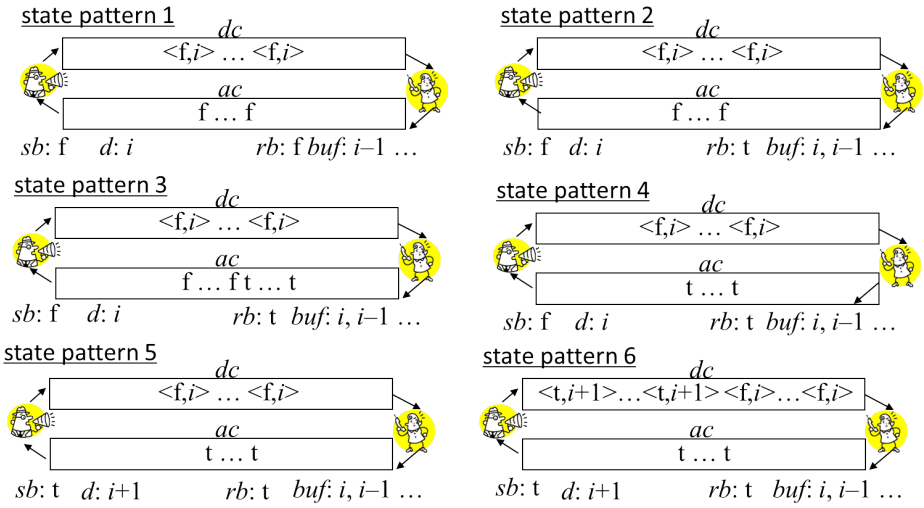


Figure 9: Six state patterns of M_{ABP}

$$\begin{aligned}(not(a-chan(S)=empty)\,and\,not(sb(S)=top(a-chan(S)))\\ and\,BIT\,in\,a-chan(S))\\ implies\,(top(a-chan(S))=BIT).\end{aligned}\tag{10}$$

$$\begin{aligned}(not(a-chan(S)=empty)\,and\,BIT\,in\,a-chan(S)\,and\,not(sb(S)=BIT))\\ implies\,(rb(S)=BIT).\end{aligned}\tag{11}$$

$$\begin{aligned}(not(d-chan(S)=empty)\,and\,rb(S)=\ fst(top(d-chan(S)))\\ and\,PAIR\,in\,d-chan(S))\\ implies\,(top(d-chan(S))\ =PAIR).\end{aligned}\tag{12}$$

$$\begin{aligned}(not(d-chan(S)=empty)\,and\,PAIR\,in\,d-chan(S)\,and\,rb(S)=\ fst(PAIR))\\ implies\,(sb(S)=fst(PAIR)\,and\,next(S)=snd(PAIR)).\end{aligned}\tag{13}$$

$$\begin{aligned}((a-chan(S)=BFIFO1@(BIT1,BIT2,BFIFO2)\,and\,not(BIT1=BIT2))\\ implies\,((BIT3\,in\,BFIFO2\,implies\,BIT2=BIT3)\,and\,BIT2=rb(S))).\end{aligned}\tag{14}$$

$$\begin{aligned}((d-chan(S)=PFIFO1\,@\,(PAIR1,PAIR2,PFIFO2)\\ and\,not(PAIR1=PAIR2))\\ implies\,((PAIR3\,in\,PFIFO2\,implies\,PAIR2=PAIR3)\\ and\,PAIR2=<\,sb(S);next(S)\,>)).\end{aligned}\tag{15}$$

$$((sb(S)=rb(S))\,implies(BIT\,in\,a-chan(S)\,implies\,BIT=rb(S))).\tag{16}$$

$$(not(sb(S) = rb(S)) \\ implies (PAIR \in d - chan(S) \, implies \, PAIR =< sb(S); next(S) >)). \tag{17}$$

The proof of (3) uses (8) and (9) as lemmas. The proof of (8) uses (9), (10) and (11) as lemmas. The proof of (9) uses (12) and (13) as lemmas. The proof of (10) uses (14) as a lemma. The proof of (11) uses (8), (9) and (10) as lemmas. The proof of (12) uses (15) as a lemma. The proof of (13) uses (12) and (15) as lemmas. The proof of (14) uses (16) as a lemma. The proof of (15) uses (8) and (17) as lemmas. The proof of (16) uses (9) and (10) as lemmas. The proof of (17) uses (8) and (12) as lemmas.

4.2 Experiments on ILP

We shows the experiments and results of two case studies: SCP and ABP, using our framework. On each case study, we introduce the input contents consisting of Background knowledge B, Examples E, Mode Declarations and Settings. Moreover, we provide some constraints in each set of examples such that it is effect to the results. There are some constraints which optimize the results. Then, we discuss about the results: compare them with the state patterns shown in Figure 8 and Figure 9 ,and also consider whether if these hypothesised clauses are useful to conjecture some lemmas, hopefully can be used to discharge some cases in the proofs.

4.2.1 Characterizing M_{SCP}

4.2.1.1 Background knowledge B

A system state of SCP is defined by six observable values: $cell1 \in PCell$, $cell2 \in BCell$, $sb, rb \in Bool$, $nxt \in PNat$ and $buf \in List$. The background knowledge needs to be provide all the definitions from SCP's system specification. Although the Prolog interpreter of Progol has a built-in definition of Boolean, it cannot be used since we use Boolean values as some compound functions' arguments in which the uses may cause some conflicts in the interpreter. Therefore, we define the type `bool` with two constants: `t` (stand for `true`) and `f` (stand for `false`), as follows:

```
bool(t).
bool(f).
```

The definition of type `pnat` consists two clauses:

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

Then, we can define the type of pairs of a boolean value and a natural number as follows:

```
bnpair(p(B,N)) :- bool(B), pnat(N).
```

We use function `p/2` followed by a bool argument and a number argument to define an object of `pnat`. The body of the clause check if the arguments' types are correct. The definitions of types `pcell` for Data Channel and `bcell` are as follows:

```
pcell(c(p(B,N))) :- bool(B), pnat(N).
bcell(c(B)) :- bool(B) .
```

Similar to type `pnat`, we use a function `c/1` to define a cell. Depend on each type of an argument, it can be decided as an object of `pcell` or `bcell`. Finally, we have a definition of type `nlist`, standing for lists of natural numbers, as follows:

```
nlist([]).
nlist([H|T]) :- pnat(H), nlist(T).
```

The first clause defines that an empty list by using the syntax of empty list provided in Prolog. We also use Prolog's syntax of non-empty list to define the non-empty list of `nlist` such that the first argument `H` is the top element of the list and `T` is the tail queue of the list. And the type of each argument must be check at the body part to ensure that `[H|T]` is an object of `nlist`.

The data structure definitions of the system specification do not contain only constructor operators but also they have some defined operator such that we can observe values which construct the data. In SCP's system specification, we have these operators: `fst` - gets a bool value of a pair, `snd` - gets a number of a pair and `mk` - constructs a order list from a number. These operators are converted to Prolog clauses as follows:

```
fst(p(B,N),B) :- bnpair(p(B,N)).
snd(p(B,N),N) :- bnpair(p(B,N)).

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

Although there are some trivial operators such that they do not need to explicitly declare in a system specification, we need to explicitly provide them in the background knowledge in form of Prolog clauses. For example, a operator states the relation between two boolean constants, a operator checks two numbers whether if one is the successor of the other. We provide these clauses to the background knowledge as follows:

```
neg(f,t).
neg(t,f).

succ(X,s(X)) :- pnat(X).
```

4.2.1.2 Positive Examples E^+

As mentioned in Chapter 3, we use YAST to convert our SCP's system specification suited for theorem proving in CafeOBJ to the one suited for model checking in Maude. Then, using the build-in command `search` of Maude to generate a finite number of reachable states. However, a reachable state of SCP is informally represented as follows:

```
Solution 3 (state 9)
states: 10  rewrites: 178 in 0ms cpu (0ms real) (689922 rewrites/second)
S:State --> buf : (0 | nil)
nxt : 0
rb : true
sb : false
PC:PCell --> c(< false ; 0 >)
BC:BCell --> c(true)
```

We needs to convert them to Prolog facts which are used as positive examples during ILP learning process. As the convention mentioned in Chapter 3, we convert them to the ground terms of predicate `state/6` in which arity 6 is the number of observable values specifying a system state. The above output can be converted to a fact `state` as follows:

```
state(t,s(0),t,[0],c(p(f,0)),c(t)).
```

where the first argument is `sb` (Sender's bit is `true`), the second one is `nxt` (Sending number is `s(0)`, standing for 1), the third one is `rb` (Receiver's bit is `true`), the fourth one is `buf` (Receiver got `0` on its buffer), the fifth one is `d-chan` (Data Channel has packet containing pair `< false ; 0 >`) and the last one is `a-chan` (Acknowledgement Channel has a message which is `true`).

4.2.1.3 Mode Declarations and Settings

Mode declarations describe the relations (predicates) between objects of given types which can be used either in the head (`modeh` declarations) or body (`modeb` declarations). Modes also describe the forms of these atoms that can be used in a clause. For the head of any clause defining `state`, we might give one of the following head mode declarations:

$$:-modeh(1, state(+bool, +pnat, +bool, +nlist, +pcell, +bcell))? \tag{18}$$

$$:-modeh(1, state(+bool, +pnat, +bool, +nlist, c(p(+bool, +pnat)), c(+bool)))? \tag{19}$$

In these mode declarations, any clause defining `state` has six arguments in order Sender's bit, Sending numbder, Receiver's bit, Receiver's buffer, Data Channel and Acknowledgement Channel. Each argument of (18) is replaced by a input variable because of `+` such that the variable must appear on the body part of the clause. If we use (18) to

define the head of clause, then we must to define a set consisting of the following body mode declarations:

$$: -modeb(1, neg(+bool, -bool))? \tag{20}$$

$$: -modeb(1, succ(+pnat, -pnat))? \tag{21}$$

$$: -modeb(1, neg(-bool, +bool))? \tag{22}$$

$$: -modeb(1, succ(-pnat, +pnat))? \tag{23}$$

$$: -modeb(1, neg(+bool, +bool))? \tag{24}$$

$$: -modeb(1, succ(+pnat, +pnat))? \tag{25}$$

$$: -modeb(1, mk(+pnat, +nlist))? \tag{26}$$

$$: -modeb(1, mk(+pnat, [+pnat| + nlist]))? \tag{27}$$

$$: -modeb(1, mk(+pnat, -nlist))? \tag{28}$$

$$: -modeb(1, mk(+pnat, [-pnat| - nlist]))? \tag{29}$$

$$: -modeb(1, fst(+bnpair, -bool))? \tag{30}$$

$$: -modeb(1, snd(+bnpair, -pnat))? \tag{31}$$

$$: -modeb(1, get(+pcell, -pnat))? \tag{32}$$

$$: -modeb(1, get(+bcell, -bool))? \tag{33}$$

Each out put variable, declared by `-type`, will introduce a new variable from input variables. Because Progol computation resource is limited, we can optimize by modifying the mode declarations such that we use head mode declaration (19). In which, we declare the functions using to construct some structure data such as `bnpair`, `bcell` and `pcell`, then we do not need to declare body mode declaration (30) (31) (32) (33). Furthermore, the head mode (19) only considers to the reachable states such that Data Channel and Acknowledge Channel are not empty. From our experience, these reachable states which

have empty channels are not interested since their characteristics are not useful to conjecture lemma. As show as in the state patterns of Figure 8, there are no such reachable states.

```
:- set(posonly)?
:- set(h,300000)?
:- set(r,4000000)?
:- set(nodes,20000)?
:- set(c,15)?
```

As mentioned, we use the learning for positive only mode, then we enable it by using the clause `:- set(posonly)?`. Since the characteristics of SCP's reachable states are the correlations between the observable values and such state has six values, some of them are instances of the recursively structured data such as Natural number, list and queues, the default settings of Progol is not sufficient to process the learning task. We must to explicitly config such parameters in the settings parts, as shown as on above clauses, we use big numbers for depth of proof `h`, depth of resolutions `r`, search nodes `nodes` and clause length `c`.

4.2.1.4 Experiments

By applying learning form positive example (reachable states) only, we have conducted around 40 experiments with respected to different database and collected several sets of clauses. There are some clauses repeatedly appearing on several experiments. We carefully observe and minimize their appearance in the experiments with respected to the completeness criteria of an ILP learning task. Finally, we obtain the follow set of clauses.

Result 1

```
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- succ(F,B), mk(B,[B|C]).
```

This clauses are collected from experiments in which each set of positive examples E^+ are generated from a specific initial state representing a state pattern in Figure 8 and a specific number of examples. Moreover, we constraints the size of any natural number instance since it is significant to consume the computation cost. The detail of each experiment is as follows:

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	38	$nxt < 11$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	77	$nxt < 21$
`state(t,s(0),t,[0],c(p(f,0)),c(t)).`	4	36	$nxt < 11$
`state(t,s(0),t,[0],c(p(f,0)),c(t)).`	4	76	$nxt < 21$
`state(t,s(0),t,[0],c(p(f,0)),c(t)).`	4	276	$nxt < 71$
`state(t,s(0),t,[0],c(p(f,0)),c(t)).`	4	316	$nxt < 81$
`state(t,s(0),t,[0],c(p(f,0)),c(t)).`	4	356	$nxt < 91$
`state(t,s(0),t,[0],c(p(f,0)),c(t)).`	4	396	$nxt < 101$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause defines State Pattern 4 and State Pattern 1.

Result 2

```
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), succ(E,B).
state(A,B,A,C,c(p(A,B)),c(A)) :- mk(B,[B|C]).
```

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	78	$nxt < 21$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause exactly defines State Pattern 4. And the third clause exactly defines State Pattern 1.

Result 3

```
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(B,[B|C]).
state(A,B,A,C,c(p(A,B)),c(A)) :- mk(B,[B|C]).
```

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	78	$nxt < 31$
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	78	$nxt < 61$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	117	$nxt < 31$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	4	116	$nxt < 31$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause exactly defines State Pattern 4. And the third clause exactly defines State Pattern 1.

Result 4

```
 state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- mk(B,[B|C]).
```

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	158	$nxt < 41$
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	278	$nxt < 71$
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	318	$nxt < 81$
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	358	$nxt < 91$
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	398	$nxt < 101$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	157	$nxt < 41$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	277	$nxt < 71$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	317	$nxt < 81$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	357	$nxt < 91$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	387	$nxt < 101$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	4	156	$nxt < 41$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause defines State Pattern 4 and State Pattern 1, but `E` is a free-variable.

Result 5

```
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(E,C).
state(A,B,A,C,c(p(A,B)),c(A)) :- mk(B,[B|C]).
```

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	198	$nxt < 51$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	197	$nxt < 51$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause exactly defines State Pattern 4. And the third clause exactly defines State Pattern 1.

Result 6

```
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(B,[B|C]).
state(A,B,A,C,c(p(A,B)),c(A)).
```

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(t)).`	2	237	$nxt < 61$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause exactly defines State Pattern 4. And the third clause exactly defines State Pattern 1.

Result 7

```
state(A,B,C,D,c(p(A,B)),c(C)) :- mk(B,[B|D]).
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(E,C).
```

Initial State	State Pattern	# States	Size of a number
`state(t,s(0),t,[0],c(p(t,s(0))),c(t)).`	1	237	$nxt < 51$
`state(t,s(0),t,[0],c(p(t,s(0))),c(t)).`	1	237	$nxt < 35$
`state(f,0,t,[0],c(p(f,0)),c(t)).`	3	237	$nxt < 51$

The first clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The second clause exactly defines State Pattern 4. And the third clause exactly defines State Pattern 1.

Result 8

```
state(A,B,C,D,c(p(A,B)),c(C)) :- mk(B,[B|D]).
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), succ(E,B).
```

Initial State	State Pattern	# States	Size of a number
`state(t,s(0),t,[0],c(p(t,s(0))),c(t)).`	1	237	$nxt < 51$

The first clause defines State Pattern 1 and State Pattern 4.The second clause clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The third clause exactly defines State Pattern 4.

Result 9

```
state(A,B,C,D,c(p(A,B)),c(A)) :- mk(B,[B|D]).
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(B,[B|C]).
```

Initial State	State Pattern	# States	Size of a number
`state(f,0,t,[0],c(p(f,0)),c(f)).`	2	238	$nxt < 61$
`state(t,s(0),t,[0],c(p(t,s(0))),c(t)).`	1	395	$nxt < 101$
`state(t,s(0),t,[0],c(p(t,s(0))),c(t)).`	1	115	$nxt < 31$

The first clause defines State Pattern 1 and State Pattern 4.The second clause clause defines State Pattern 2 and State Pattern 3 since `buf D` is a list of numbers generated by `nxt B` (`mk(B,D)`), but `E` is a free-variable. The third clause exactly defines State Pattern 4.

Result 10

```
state(A,B,C,D,c(p(A,B)),c(E)).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(E,C).
```

Initial State	State Pattern	# States	Size of a number
`state(t,s(0),t,[0],c(p(t,s(0))),c(t)).`	1	155	$nxt < 41$

The first clause defines State Pattern 1, State Pattern 2 and State Pattern 3.The second clause exactly defines State Pattern 4.

4.2.1.5 Evaluation

Let consider to the following clause:

```
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(B,[B|C]).
```

This clause express characteristic (1), (4) and (5) in Table 1 showing the relation between three the bits: sb, rb and the bit on achan (by `state(A,B,A,C,c(p(D,E)),c(A))`), characteristic (15) shows that the bit in top packet of dchan and *sb* are complement (by literal `neg(A,D)`). Moreover, the clause also expresses characteristic (8) showing that the list made by sending number *nxt* is the same as the list made by *next* and the buffer *buf*. The characteristic is captured by the State Pattern 4 as Figure 10 . This is an example of comparison for checking that the clause is complete.

To the other direction, let consider Figure 8, we can observe a characteristic which the state patterns capturing is about the relation between *sb* and *rb*. Whenever they are the same, the list made by *nxt* and the list made by *next* and the buffer *buf* are the same. Otherwise, the list made by *nxt* and *buf* are the same. This characteristic is expressed in the following clauses of Experiment 4:

```
state(A,B,C,D,c(p(A,B)),c(E)) :- neg(A,C), mk(B,D).
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), succ(E,B).
state(A,B,A,C,c(p(A,B)),c(A)) :- mk(B,[B|C]).
```

We applied the same assessment technique to all other characteristics which we can manually observe. Then, all considered characteristics are captured/expressed. In other words, we have successfully characterized reachable states of SCP.

On these experiments, there are some clauses defining exactly a snapshot on the series of snapshots (Result 3 and Result 12). The size of term such as natural numbers make the computation costly, even it may be wrong and effected to final results. Since we fix the size, it causes an bound number of reachable states in database for a experiment. Let consider to the Result 3.

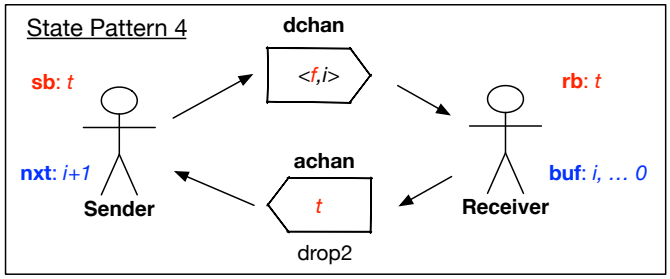


Figure 10: State Pattern 4 captures the characteristics expressing on the clause `state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D), mk(B,[B|C]).`

```
state(A,B,C,D,c(p(A,B)),c(E)) :- mk(B,D) .
state(A,B,A,C,c(p(D,E)),c(A)) :- neg(A,D),mk(B,[B|C]) .
state(A,B,A,C,c(p(A,B)),c(A)) :- mk(B,[B|C]) .
```

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 $\{\text{state}(\overrightarrow{x}) \coloneq cond_i(\overrightarrow{x}) \mid i = 1, \ldots, n\}$ 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, $\text{state}(A, B, A, C, \text{c}(\text{p}(A, B)), \text{c}(A)) :- \text{mk}(B, [B|C])$ is written as $\text{state}(A, B, A2, C, D, E) :- \text{mk}(B, [B|C]), A2 = A, D = \text{c}(\text{p}(A, B)), E = C(A)$. 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. **??**.

With respected to the background knowledge provided by the system specification, Progol has successfully characterize all the characteristics in The four state patterns. Unfortunately, there is no experiment such that each each hypothesised clause characterize exactly a state pattern. Therefore, to understand about the state machine, we must consider to the different results from several experiment. For example, combining Result 12 and Result 13, we have a set of clauses characterize all State patterns.

4.2.2 Characterizing M_{ABP}

4.2.2.1 Background knowledge B

A system state of ABP is defined by six observable values: $d-chan \in DChan$, $a-chan \in AChan$, $sb, rb \in Bool$, $nxt \in PNat$ and $buf \in List$. The background knowledge needs to be provide all the definitions from ABP's system specification. We can reuse the definition of $Bool$, $PNat$ and $List$ from SCP experiments. But $DChan$ and $AChan$ must to be defined as follows:

```
pqueue([]).
pqueue([H|T]) :- bnpair(H), pqueue(T).

bqueue([]).
bqueue([H|T]) :- bool(H), bqueue(T).
```

where `pqueue` and `bqueue` are the predicates defining $DChan$ and $AChan$, respectively.

Similar to define `nlist`, we use the syntax of list in Progol. Since the ABP's behaviours does not show the differences between list and queue, then we can define them as same as `nlist`, just different at their predicate names and the condition parts checking the types of arguments. Since the new data types are defined, their defined operators in ABP's system specification are also.

```
toppqu([H|T], H) :- bnpair(H).
topbqu([H|T], H) :- bool(H).

memberp(A,[A|B]):- bnpair(A),pqueue([A|B]),!.
memberp(A,[B|C]) :- bnpair(A), pqueue([B|C]), memberp(A,C).
```

```
memberb(A,[A|B]):- bool(A),bqueue([A|B]),!.
memberb(A,[B|C]) :- bool(A), bqueue([B|C]), memberb(A,C).
```

where `toppqu/2` gets a non-empty queue as the first argument and return the top element as the second argument. We provide two different clauses defining `toppqu/2` since we have two kinds of queue and the body parts checking the types of arguments. One popular operator of queue/list are the operator checking whether if an element is in a list/queue, then we provide two predicates `memberp/2` and `memberb/2` for `pqueue` and `bqueue`, respectively.

4.2.2.2 Positive Examples E^+

We use our framework such that the YAST tool to generate the reachable states of ABP's state machine. One output of an ABP state is as follows:

```
Solution 2 (state 1013)
states: 1014  rewrites: 71266 in 53ms cpu (53ms real) (1334369 rewrites/second)
S:State --> sb : true
rb : true
buf : (0 nil)
PQ:DChan --> < false ; 0 > ; < false ; 0 > ; < false ; 0 > ; < false ; 0 > ; <
    true ; s(0) > ; (empty).PEmpty
BQ:AChan --> true ; true ; true ; true ; true ; (empty).BEmpty
N:Nat --> s(0)
```

The above state is converted to a fact of `state` as follows:

```
state(t,s(0),f,[0],[p(f,0),p(f,0),p(f,0),p(f,0),p(t,s(0))],[t,t,t,t,t]).
```

where the first argument is `sb` (Sender's bit is `true`), the second one is `nxt` (Sending number is `s(0)`, standing for 1), the third one is `rb` (Receiver's bit is `false`), the fourth one is `buf` (Receiver got `0` on its buffer), the fifth one is `d-chan` (Data Channel has packet containing pair `< false ; 0 > ; < false ; 0 > ; < false ; 0 > ; < false ; 0 > ; < true ; s(0) > ; empty`) and the last one is `a-chan` (Acknowledgement Channel has a message which is `true ; true ; true ; true ; true ; empty`).

4.2.2.3 Mode Declarations and Settings

For the structure of predicate `state` which we want to learn, we define the mode declarations. We can declare the body mode declaration as follows:

$$:-modeh(1, state(+bool, +pnat, +bool, +nlist, +pqueue, +bqueue))? \tag{34}$$

$$:-modeh(1, state(+bool, +pnat, +bool, +nlist, [p(+bool, +pnat), p(+bool, +pnat)|+pqueue], [+bool|+bqueue]))? \tag{35}$$

In these mode declarations, any clause defining `state` has six arguments in order Sender's bit, Sending numbder, Receiver's bit, Receiver's buffer, Data Channel and Acknowledgement Channel. Each argument of (34) is replaced by a input variable because of `+` such that the variable must appear on the body part of the clause. If we use (34) to define the head of clause, then we must to define a set consisting of the following body mode declarations:

$$: -modeb(1, neg(+bool, -bool))? \tag{36}$$

$$: -modeb(1, succ(+pnat, -pnat))? \tag{37}$$

$$: -modeb(1, neg(+bool, +bool))? \tag{38}$$

$$: -modeb(1, succ(+pnat, +pnat))? \tag{39}$$

$$: -modeb(1, mk(+pnat, +nlist))? \tag{40}$$

$$: -modeb(1, mk(+pnat, [+pnat| + nlist]))? \tag{41}$$

$$: -modeb(1, mk(+pnat, -nlist))? \tag{42}$$

$$: -modeb(1, mk(+pnat, [-pnat| - nlist]))? \tag{43}$$

$$: -modeb(*, memberp(p(+bool, +pnat), +pqueue))? \tag{44}$$

$$: - : -modeb(*, memberb(+bool, +bqueue))? \tag{45}$$

$$: - : -modeb(1, topbqu(+bqueue, +bool))? \tag{46}$$

$$: - : -modeb(1, toppqu(+pqueue, +bnpair))? \tag{47}$$

We can optimize by modifying the mode declarations such that we use head mode declaration (35). In which, we declare the functions using to construct some structure data such as `bnpair`, `bqueue` and `pqueue`, then we do not need to declare body mode declaration (44) (45) (46) (47). Furthermore, the head mode (35) only considers to the reachable states such that Data Channel and Acknowledge Channel are not empty. From our experience, these reachable states which have empty channels are not interested since their characteristics are not useful to conjecture lemma. As show as in the state patterns of Figure 9, there are no such reachable states.

```
:- set(posonly)?
:- set(h,300000)?
:- set(r,4000000)?
:- set(nodes,20000)?
:- set(c,15)?
```

As mentioned, we use the learning for positive only mode, then we enable it by using the clause `:- set(posonly)?`. Since the characteristics of ABP's reachable states are the correlations between the observable values and such state has six values, some of them are instances of the recursively structured data such as Natural number, list and queues, the default settings of Progol is not sufficient to process the learning task. We must to explicitly config such parameters in the settings parts, as shown as on above clauses, we use big numbers for depth of proof `h`, depth of resolutions `r`, search nodes `nodes` and clause length `c`.

4.2.2.4 Experiments

We have conducted around 30 experiments with the mode declaration modes: head mode declaration (35) and body mode declarations (36) - (43) with respected to the background knowledge provided by ABP's system specification on CafeOBJ. Each experiment is different at the number of reachable states and the initial state used to generate the reachable states. To optimise the computation cost of Progol, we constraint the size of an instance of natural number as less than 11 but greater than 0, the size of Data channel and Acknowledgement channel from 5 to 10. We shows each experiment with respected to the features of using reachable states and then compare each clause with the state pattern in Figure 9. In general, there are two classes of conducted experiments: about 20 experiments w.r.t. background knowledge from the specification (class 1) and about 10 experiments w.r.t. background knowledge from the specification and some user-defined functions (class 2) such as *gap*0 and *gap*1. By apply completeness criteria, we collected around 53 clauses for class 1 and around 39 clauses for class 2.

Experiment 1

- Database constraints
  - 10000 states
  - Initial states - an instance of State Pattern 4

    ```
    state(t,s(0),f,[0],[p(f,0)],[t]).
    ```

- Clauses

```
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,C), succ(F,B), mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- mk(B,D).
state(A,B,A,C,[p(D,E)|F],[A|G]) :- neg(A,D), succ(E,B), mk(B,[B|C]).
```

The first clause is a wrong description since if `neg(A,C)`, then `mk(B,D)` is true but the clause has `mk(B,[B|D])`.The second clause exactly defines State Pattern 1. The third clause defines State Pattern 2, State Pattern 3 and State Pattern 4. The fourth clause defines State Pattern 5 and State Pattern 6.

Experiment 2

- Database constraints
  - 1500 states
  - Initial states - an instance of State Pattern 4

    ```
    state(t,s(0),f,[0],[p(f,0)],[t]).
    ```

- Clauses

```
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- mk(B,D).
```

The first clause defines State Pattern 5 and State Pattern 6. The second clause exactly defines State Pattern 1. The third clause defines State Pattern 2, State Pattern 3 and State Pattern 4.

Experiment 3

- Database constraints
  - 1500 states
  - Initial states - an instance of State Pattern 3

    ```
    state(t,s(0),f,[0],[p(t,s(0))],[t]).
    ```

- Clauses

```
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,C), mk(F,D), memberb(A,H).
state(A,B,C,D,[p(C,E)|F],[A|G]) :- mk(E,D), memberp(p(A,B),F).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- mk(B,D).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
```

The first clause defines State Pattern 2 and State Pattern 3.The second clause defines State Pattern 2, State Pattern 3, State Pattern 4, State Pattern 5 and State Pattern 6. The third clause defines State Pattern 2, State Pattern 3 and State Pattern 4. The fourth clause exactly defines State Pattern 1. The fifth clause defines State Pattern 5 and State Pattern 6.

Experiment 4

- Database constraints
  - 1394 states
  - Initial states - an instance of State Pattern 4

```
state(t,s(0),f,[0],[p(f,0),p(f,0),p(f,0),p(f,0),p(f,0)],
                                                    [t,t,t,t,t]).
```

- Clauses

```
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,C), mk(F,D),
                                   memberp(p(E,F),G), memberb(A,H).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), mk(B,[B|D]),
                                   memberp(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,D), memberp(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[C|F]) :- mk(B,D), memberp(p(A,B),E),
                                   memberb(C,F).
state(A,B,A,C,[p(A,B)|D],[A|E]) :- mk(B,[B|C]), memberp(p(A,B),D),
                                   memberb(A,E).
state(A,B,A,C,[p(A,B)|D],[E|F]) :- mk(B,C), memberp(p(A,B),D).
```

The first clause defines State Pattern 2, State Pattern 3 and State Pattern 4.The second clause defines State Pattern 5 and State Pattern 6. The third clause is a wrong description. The fourth clause defines State Pattern 2, State Pattern 3 and State Pattern 4. The fifth clause exactly defines State Pattern 4. The sixth clause exactly defines State Pattern 1. The seventh clause defines State Pattern 1, State Pattern 4 and State Pattern 5.

Experiment 5

- Database constraints
  - 1500 states
  - Initial states - an instance of State Pattern 2

```
state(t,s(0),f,[s(0),0],[p(t,s(0))],[t,t,f]).
```

- Clauses

```
state(A,B,C,D,[p(C,E)|F],[A|G]) :- neg(C,H), succ(E,B), mk(B,[B|D]),
                                   memberb(H,G).
state(A,B,C,D,[p(C,E)|F],[A|G]) :- mk(E,D), memberp(p(A,B),F).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- mk(B,D).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
```

The first clause exactly defines State Pattern 6.The second clause defines State Pattern 2, State Pattern 3, State Pattern 4 and State Pattern 5. The third clause defines State Pattern 2, State Pattern 3 and State Pattern 4. The fourth clause defines State Pattern 1, State Pattern 4 and State Pattern 5. The fifth clause exactly defines State Pattern 1.

Experiment 6

- Database constraints
  - 1500 states
  - Initial states - instance of Snapshot 1

    ```
    state(f,s(s(0)),f,[s(0),0],[p(f,s(s(0))),p(f,s(s(0))),
    p(f,s(s(0))),p(f,s(s(0))),p(f,s(s(0)))],[f,f,f,f,f]).
    ```

- Clauses

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C).
state(A,B,C,D,[p(C,E)|F],[A|G]) :- neg(C,H), succ(E,B), mk(B,[B|D]),
                                   memberb(H,G).
state(A,B,C,D,[p(C,E)|F],[A|G]) :- mk(E,D), memberp(p(A,B),F).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- mk(B,D).
state(A,B,A,C,[p(D,E)|F],[A|F]) :- mk(E,C).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
```

The first clause defines State Pattern 2 and State Pattern 3.The second clause exactly defines State Pattern 6. The third clause defines State Pattern 2, State Pattern 3, State Pattern 4, State Pattern 5 and State Pattern 6. The fourth clause defines State Pattern 2, State Pattern 3 and State Pattern 4. The fifth clause defines State Pattern 5 and State Pattern 6. The sixth clause exactly defines State Pattern 1. The seventh clause defines State Pattern 5 and State Pattern 6.

Experiment 7

- Database constraints
  - 1394 states
  - Initial states - an instance of State Pattern 3

    ```
    state(t,s(0),f,[0],[p(t,s(0)),p(t,s(0)),
    p(t,s(0)),p(t,s(0)),p(t,s(0))],[t,t,t,t,t]).
    ```

- Clauses

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), mk(B,[B|D]),
                                   memberp(p(A,B),E).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,D), memberp(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[C|F]) :- mk(B,D), memberp(p(A,B),E),
                                    memberb(C,F).
state(A,B,A,C,[p(A,B)|D],[A|E]) :- mk(B,[B|C]), memberp(p(A,B),D),
                                   memberb(A,E).
state(A,B,A,C,[p(A,B)|D],[E|F]) :- mk(B,C), memberp(p(A,B),D).
```

The first clause exactly defines State Pattern 1.The second clause defines State Pattern 5 and State Pattern 6. The third clause defines State Pattern 2 and State Pattern 3. The fourth clause exactly defines State Pattern 4. The fifth clause exactly defines State Pattern 1. The sixth clause defines State Pattern 1, State Pattern 5 and State Pattern 6.

Experiment 8

- Database constraints
  - 1394 states
  - Initial states - an instance of State Pattern 2

    ```
    state(t,s(0),f,[s(0),0],[p(t,s(0)),p(t,s(0)),
    p(t,s(0)),p(t,s(0)),p(t,s(0))],[t,t,t,t,f,f]).
    ```

- Clauses

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), mk(B,D),
                                   memberp(p(A,B),E).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), mk(B,[B|D]),
                                   memberp(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), mk(B,[B|D]),
                                   memberp(p(A,B),E).
state(A,B,A,C,[p(A,B)|D],[A|E]) :- mk(B,[B|C]), memberp(p(A,B),D),
                                   memberb(A,E).
state(A,B,A,C,[p(A,B)|D],[E|F]) :- mk(B,C), memberp(p(A,B),D).
```

The first clause defines State Pattern 1 and State Pattern 3.The second clause defines State Pattern 5 and State Pattern 6. The third clause defines State Pattern 1, State Pattern 5 and State Pattern 6. The fourth clause exactly defines State Pattern 4. Two last clauses define exactly State Pattern 1.

Experiment 9

- Database constraints
  - 1394 states
  - Initial states - an instance of State Pattern 1

    ```
    state(f,s(s(0)),f,[s(0),0],[p(f,s(s(0))),p(f,s(s(0))),
    p(f,s(s(0))),p(f,s(s(0))),p(f,s(s(0)))],[f,f,f,f,f]).
    ```

- Clauses

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]), memberp(p(A,B),E),
                                   memberb(A,F).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,D), memberp(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- neg(A,F), mk(B,D),
                                   memberp(p(A,B),E), memberb(F,G).
```

The first clause exactly defines State Pattern 1.The second clause exactly defines State Pattern 6. The third clause defines State Pattern 2, State Pattern 3 and State Pattern 4. The fourth clause exactly defines State Pattern 4. The last clause define exactly State Pattern 4.

Experiment 10

- Database constraints
  - 1394 states
  - Initial states - an instance of State Pattern 3

    ```
    state(t,s(0),f,[0],[p(t,s(0)),p(t,s(0)),
    p(t,s(0)),p(t,s(0)),p(t,s(0))],[t,t,t,t,t]).
    ```

  - providing user-defined predicates `gap0` and `gap1`

- Clauses

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), gap0(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[C|F]) :- mk(B,D), gap0(p(A,B),E).
state(A,B,A,C,[p(D,E)|F],[A|G]) :- neg(A,D), succ(E,B), mk(B,[B|C]),
                                   gap1(p(A,B),F).
state(A,B,A,C,[p(A,B)|D],[A|E]) :- mk(B,[B|C]), gap0(p(A,B),D).
```

The first clause exactly defines State Pattern 3.The second clause defines State Pattern 2, State Pattern 2 and State Pattern 4. The third clause exactly defines State Pattern 6. The fourth clause exactly defines State Pattern 1.

4.2.2.5 Evaluation

We also apply the framework of assessment. For the completeness assessment, we have 77% characteristics of clauses in class 1 are correctly captured by the state patterns and 74% for class 2. But, when checking the soundness assessment, there are some characteristics appearing in the state patterns but there is no clause expressed them. Let consider the following case.

There are some clauses defining exactly a snapshot such as Experiment 1, Experiment 4 and Experiment 6. If we constraint the number of each snapshot and the 1st state in the database, the result will be different. Let consider to the hypothesised clauses of Experiment 6:

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C).
state(A,B,C,D,[p(C,E)|F],[A|G]) :- neg(C,H), succ(E,B), mk(B,[B|D]),
                                   memberb(H,G).
state(A,B,C,D,[p(C,E)|F],[A|G]) :- mk(E,D), memberp(p(A,B),F).
state(A,B,C,D,[p(A,B)|E],[F|G]) :- mk(B,D).
state(A,B,A,C,[p(D,E)|F],[A|F]) :- mk(E,C).
state(A,B,C,D,[p(A,B)|E],[A|F]) :- mk(B,[B|D]).
state(A,B,C,D,[p(E,F)|G],[A|H]) :- neg(A,E), succ(F,B), mk(B,[B|D]).
```

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. 9. 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 next($\langle b, i\rangle$) appear at most once in *dc*, where next($\langle b, i\rangle$) = $\langle \neg b, i+1\rangle$. We have added two predicates gap0 and gap1 whose definitions as follows

```
gap0(P,[]) :- bnpair(P).
gap0(P,[P|T]) :- gap0(P,T).

gap1(P1,[P2,P2|T]) :- next(P2,P1), gap1(P1,[P2|T]).
gap1(P1,[P2|T]) :- next(P2,P1), gap0(P1,T).
```

Then, the following set of clauses have been learned by Progol in Experiment 10:

```
state(A,B,C,D,[p(A,B)|E],[A|F]) :- neg(A,C), gap0(p(A,B),E).
state(A,B,C,D,[p(A,B)|E],[C|F]) :- mk(B,D), gap0(p(A,B),E).
state(A,B,A,C,[p(D,E)|F],[A|G]) :- neg(A,D), succ(E,B), mk(B,[B|C]),
                                   gap1(p(A,B),F).
state(A,B,A,C,[p(A,B)|D],[A|E]) :- mk(B,[B|C]), gap0(p(A,B),D).
```

The third clause more precisely extracts the characteristics of *State Pattern 6* showing in Fig. 9, including the important characteristics of *dc*. Since the third clause is the only one in which gap1(p(A, B), F) holds, we can conjecture a lemma from this clause. gap1(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* [**?**]. 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* [**?**] will help us do so.

5 Conclusion

5.1 Summary

Interactive Theorem Proving is a formal method technique using to check if a system is satisfied some expected properties such that the system's implementation can adapt to many criterions in which there is no any undesirable things happening during its execution. Such undesirable things may cause many critical problems such as loss of life, money and cost so much time to recover. One of the most intellectual activities in ITP is *Lemma Conjecture*, which usually requires human users interactive with the system under verification with respected to the understanding of not only the system behaviours but also the proof problems. To obtain such understanding, the users must to rely on some reliable sources. From our experience, one possible source we can rely on is the set of reachable states of a state machine of the system. The correlations among of the reachable states are called *characteristics* of the reachable states. Such characteristics are expressed in some system properties which we are proving. Note that if we just arbitrarily choose some states and some finite execution paths starting with those chosen states, the characteristics may not be useful to Lemma Conjecture. We need to consider to a number of reachable states as large (may be infinite) as possible. However, human beings is not able to tackle such task. Fortunately, acquiring knowledge/patterns from such big database is the task of machine learning. Our project is a consideration of a application of machine learning to Interactive Theorem Proving, in which the problem instances are reachable states and the knowledge we want to obtain is the characteristics of reachable states. Because the data representation in most of traditional machine learning techniques is propositional logic but our system specifications are based on first order logic, this requires a way to convert between two kinds of representations. This is a non-trivial task for a way converting from first order logic terms to propositional ones. To deal such representation problem, we introduce *Inductive Logic Programming*, a research area stayed at intersection of machine learning and logic programming. This machine learning technique treats the database in form of first order logic and it has been developing many approach to deal with first-order representation.

We have conducted a framework which is a combination of tools used such that YAST and Maude are used to generate reachable states from a CafeOBJ system specification. However, Progol accepts only input in form of Horn clauses, we needs to convert YAST's output to a set of facts in which each fact is a Prolog clause without body. This step can process automatically by writing a convert program but we must to manually convert from the definitions of a specification to a set of Prolog clauses using as the background knowledge. Although Prolog and CafeOBJ shares the essential first order logic, they are different on syntaxes and implementation, we needs manually convert them and/or explicitly some basic definitions which automatically importing to CafeOBJ. Our framework requires the input prepared in advance. In order words, we consider not only to the conversion to get background knowledge and database but also to limitation of resource computation of Progol, we needs to explicitly describe the parameters for the option.

Moreover, to optimize the computation we need to give some constraints for some types of data such as its instance's size. In general, our framework still partially replies on human user for the purpose.

To demonstrate our framework for the task of characterizing reachable states of a state machine of a verifying system, 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.

In general, our framework has characterized all characteristics of SCP such that its verification can be completed by providing such obtained characteristics to the users, making them understand precisely SCP's state machine and be possible to conjecture some useful lemmas. But for the case of ABP, it is more complex than ABP because of the unbounded channels, the framework cannot characterize some important characteristics for the verification with respected to the current background knowledge provided by its CafeOBJ system specification as mentioned above. Honestly, we have also conducted several other experiments for several systems other than ABP and SCP. But their are the toy case studies such that we have already done their verifications and be able to compare the obtained characteristics with our conjectured lemmas. We need to do the task of characterization with our framework with more complex systems such as distributed systems, network protocol. But, with the result we obtained and the framework we are using, we can conclude that Machine Learning can be apply to some non-trivial tasks of Interactive Theorem Proving such as Lemma Conjecture.

5.2 Related Work

ML has been used to find lemmas in ACL2 [6]. 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 [18]. 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. Our learning task is considered to characterize reachable states of a state machine, in other words, extract some patterns from reachable states, we do not require to prepare any existing database in advance such as ACL2, our database is generated on-the-fly from a system specification. And our learning task relates to classification problem of Machine Learning such that the pattern obtained can let us know whether if a system state is reachable. The integration with model checking in [18] so relates to one of the most intellectual activities in ITP, that is writing a system specification such that some results can be derived by a proof system or a theorem prover such as CafeOBJ. But our project motivation is related to lemma conjecturing, this step is usually required during a verification with respected to a sufficient system specification.

There are many researches which have been conducted for the purpose of conjecturing lemmas. Some of them uses the concept of fixed point computation. Since the fixed point computation is an undecidable, these researches have been tried to compute the approximate ones, e.g. Creme [8]'s method using as lemmas state predicates whose invariant proofs may not be completed. The reachable states of a state machine is also considered for generating and strengthening invariants by Tiwari et al. [27] based on computing under- and over-approximations of the reachable state. The bottom-up method, which performs an abstract forward propagation to compute the set of all reachable configurations, is used to compute under-approximation, and the top-down method, which starts from an invariant candidate and performs an abstract backward propagation to compute a strengthened invariant, is used to compute over-approximation.

There are a inductive theorem prover which supports some form of automated lemma discovery such as ACL2 [6] using a top-down approach by which lemmas are discovered from failed proof-attempts. ACL's method is considered to a provided existing proof set in which it will calculate the similarity of the current proof with some proofs in the set by a machine learning technique, named clustering. Then it picks the most similar one's lemmas to calculate lemmas to the current proof by randomly generating terms based on its structure. There a another theorem prover which also supports to automate lemma discovery that is HipSpec [5], but it implemented a bottom-up theory exploration approach. HipSpec automatically tries to discover a background theory for the relevant functions, building up something like the human-create lemma libraries available for interactive provers such as ACL2.

The approaches of Tiwari et al., Creme and HipSpec are symbolic methods whose limitation are slow on large inputs due to the increase in the search space by deduction inferences, usually rely on having access to good counter-example finders for filtering of candidate conjectures. In our case, the approach is also related to a search space but the search space are bounded by a most specific clause generated by Mode-directed inverse

entailment implemented by Progol. The approach also reply on the deduction theory in logic since the data representation in form of first-order logic but it is combined with many machine learning approach such as statistic or probabilistic for the purpose of learning from data. Moreover, the most advantage of our machine learning approach is dealing with a big number of data since a set of reachable states may be infinite.

5.3 Future Work

We have proposed a approach using ILP, a machine learning technique suited for learning data in form of first-order logic, to characterize reachable states of a state machine such that the state machine is used in a verification of ITP. The obtained characteristics in form of Horn-clause are compared with State Patterns in form of series pictures we were used in completed verifications. The comparison opens many future works for the application of machine learning into ITP in which ILP is the main approach since the essence of ITP is logic programming. In general, we consider to two main targets related to two most intellectual activities in ITP has been mentioned, they are Lemma Conjecturing and System Specifying.

Lemma Conjecturing is our current target, but we have not archived. There are many ways to Conjecture Lemma which are related to Invariant Generation and our approach is also. But, relied on our experience during the process of verification in ITP, we focus on reachable states of a state machine. If we can understand the reachable states in some extent with respected to the proof targets, we are able to conjecture some non-trivial lemma, hopefully, they are useful to discharge our undischarged proofs. In practical, the understanding can be improved by the characteristics of reachable states. In our experiments, we have obtained the characteristics in form of clausal logic clauses. However, the experiments with ABP, some important characteristics can not be obtained without a suitable background knowledge, that means the current background knowledge provided by ABP's system specification is quite 'weak' for an ILP learning task. This has been show when we improve the background knowledge with two predicate gap0 and gap1. 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 gap0 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.

Assuming that we already proposed a characterization method that can be applied to a wide-range system in which the user can systematically conjecture some lemmas, which are useful to discharge some induction cases/subcases of a verification. We can come up to Design/Implement a new ILP system on top of Maude. While our system specifica-

tions are written by CafeOBJ language with equational theory, mainly replied on pattern matching, Progol or Metagol use Prolog as a language to represent its data such that Prolog usually applies resolution in its proofs. Therefore, we need to convert from CafeOBJ language to Prolog for the learning task. Then, after the learning task, the clauses extracted may be need to convert back to CafeOBJ language if the characteristics they expressed can be used as a lemmas for induction cases/induction cases on CafeOBJ system. This difference and the back and forward translation costs the computation resources and some features belong to each underlying theory may be not guaranteed. Moreover, since Maude language is a sibling language of CafeOBJ language, its translation between them is guarantee. Furthermore, Maude is equipped with unification that generalizes pattern matching. Hence, resolution can be implemented on top of Maude. And Maude admits associative and commutative reduction with unification as well as pattern matching. But, there is not any implementation of Prolog that admits associative and commutative resolution. So, an ILP system that admits associative and commutative resolution may be implemented on top of Maude. Specifications of distributed systems in Maude often use associative and commutative data structures. Therefore, to capture the characteristics of specifications of distributed systems in Maude, an ILP system that admits associative and commutative resolution. The new ILP system accepts a CafeOBJ/Maude system specification as an input, and then each components of an ILP learning task will be generated. We do not need to manually convert Background Knowledge as current Research Status since all clauses in Background knowledge are written in Maude language. And reachable states will be automatically generated by search command and prepare in advance.

In other words, one possible our target is to design and implement a new ILP system based on order-sorted equational/rewriting logic. First, we need come up with a method to precisely characterize reachable states of a state machine. One possible target is Predicate Invention with the new ILP system, Metagol. When we successfully adapt to the task of characterization, a new ILP system can be designed and implemented on top of Maude since our ultimate goal is conjecture lemmas for a verification on CafeOBJ. The new system will take the advantage of resource computation and optimize the learning process with some theory provide by Maude such as associative and commutative resolution. To archive this goal, we require a basic understanding ML and logic programming. Especially, we need to adapt to the most achievements not only ILP but also Theorem Proving, especially ITP on CafeOBJ and Maude. The new ILP system requires us a good skill at meta-programming to build an application on top of Maude.

System Specifying activity also requires human interaction in ITP such that the users must understand system requirements in some extent and then come up with a way to specify system's state machine in a specification language, e.g. CafeOBJ, Maude, etc. such that it is suitable for a verification of a ITP prover. As mentioned on Section Related Work, there is an approach to this target [18] but it is related to Model Checking, as same as ITP, also Formal verification but it is a model-based approach. This approach uses a model checker to generate each essential component for an ILP learning, then correcting the current system specification iteratively. Let consider to ITP, we also need to specify each essential component for an ILP learning task with ones in ITP. Honestly,

we are considering the problem but this is an interesting application for our future work. If we have a correct system specification, which means that we may have a suitable system specification to generate a suitable background knowledge for purpose of Lemma Conjecturing.

References

[1] R. Diaconescu, K. Futatsugi, CafeOBJ Report, World Scientific, 1998

[2] C. Bishop Pattern Recognition and Machine Learning , Information Science and Statistics, 1st edition, 2006

[3] L. De Readt Logical and Relational Learning, Springer, 2008

[4] A. Ireland, A. Bundy, Productive use of failure in inductive proof JAR 16 (1996) 79-111

[5] K. Claessen, M. Johansson, D. Rosen, N. Smallbone, Automating inductive proofs using theory exploration. In: CADE-24. LNCS 7898 (2013) 392-406

[6] J. Heras, E. Komendantskaya, M. Johansson, E. Maclean, Proof-pattern recognition and lemma discovery in ACL2. In: LPAR-19. LNCS 8312 (2013) 389-406

[7] E. Komendantskaya, J. Heras, G. Grov, Machine learning in proof general: Interfacing interfaces. In: UITP. EPTCS 118 (2013) 15-41

[8] M. Nakano, K. Ogata, M. Nakamura, K. Futatsugi, Creme: An automatic invariant prover of behavioral specifications. IJSEKE 17 (2007) 783-804

[9] S. Muggleton, L. De Raedt, Inductive logic programming: Theory and methods, The Journal of Logic Programming 19, 629-679

[10] K. Ogata, K. Futatsugi, Proof scores in the OTS/CafeOBJ method, FMOODS'03, LNCS 2884,pp.170-184,Springer,2003

[11] K. Ogata, K. Futatsugi, Some tips on writing proof scores in the OTS/CafeOBJ method, Essays Dedicated to Jeseph A. Goguen, LNCS 4060, pp. 596-615, 2006

[12] M. Zhang, K. Ogata, M. Nakamura, Translation of state machines from equational theories into rewrite theories with tool support. IEICE Transactions 94-D (2011) 976-988

[13] L. De Raedt, Logical and Relational Learning, Springer, 2008

[14] S. Muggleton, Inverse entailment and Progol, New generation computing 13 (3), 1995, 245-286

[15] S Muggleton, Learning from positive data, Inductive logic programming Workshop 1996, 358-376

[16] N. Lynch, Distributed algorithms, Morgan Kaufmann, 1996

[17] F. Clavel,M. Duran, All about Maude, LNCS 4350, Springer, 2007.

[18] D. Alrajeh, A. Russo, S. Uchitel, J. Kramer, Integrating model checking and inductive logic programming, ILP'11, 45–60, (2011).

[19] L. De Readt Attribute-value learning versus inductive logic programming: The missing links, Volume 1446 of the series Lecture Notes in Computer Science, pp 1-8, 2005

[20] L. Sterling and E. Y. Shapiro, The Art of Prolog, MIT Press, 1986.

[21] T. Richards, Clausal Form Logic an Introduction to the Logic of Computer Reasoning, Monograph Collection, 1989.

[22] A. M. Turing, On computable numbers, with an application to the entscheidungsproblem, Proc. Lond. Math. Soc., 42(2):230?265, 1936.

[23] M. Sipser, Introduction to the Theory of Computation. Springer-Verlag, 2003

[24] M. Kaufmann and J.S. Moore, Some Key Research Problems in Automated Theorem Proving for Hardware and SoftwareVerification, Spanish Royal Academy of Science(RAMSAC), Volume 98, 181?196, 2004

[25] I. Stahl, Predicate invention in ILP - an overview, ECML-93, 313- 322, 1993

[26] S.H. Muggleton, D. Lin and A. Tamaddoni-Nezhad, Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited, Machine Learning, Volume 100, 49-73, 2015

[27] A. Tiwari, H. Rueb, H. Saidi, and N. Shankar, A technique for invariant generation, 7th TACAS,LNCS 2031, Springer,2001, pp.113-127.

Contributions

[1] Tuan Dung Ho, Min Zhang and Kazuhiro Ogata, A Case Study on Extracting the Characteristics of the Reachable States of a State Machine formalizing a Communication Protocol with Inductive Logic Programing, (accepted as Long Paper Short Presentation through Peer Review), 25th International Conference on Inductive Logic Programming, 2015

[2] Tuan Dung Ho, Min Zhang and Kazuhiro Ogata, An Approach to Application of Inductive Logic Programming to Characterization of Reachable States, IEICE Technical Report, Vol. 114, No. 416, ISSN 0913-5685, pp.61-66, 2015

Appendices

Progol's input for SCP experiment

```
% Settings

% :- set(posonly)?
% :- unset(cover)?
:- set(h,300000)?
:- set(r,4000000)?
:- set(nodes,20000)?
:- set(c,15)?
% :- set(i,5)?
:- set(noise,100)?


% Mode declarations

:- modeh(1,state(+bool,+pnat,+bool,+nlist,c(p(+bool,+pnat)),c(+bool)))?
%:- modeh(1,state(+bool,+pnat,+bool,+nlist,+pcell,+bcell))?

:- modeb(1,neg(+bool,-bool))?
:- modeb(1,succ(+pnat,-pnat))?
:- modeb(1,neg(-bool,+bool))?
:- modeb(1,succ(-pnat,+pnat))?
:- modeb(1,neg(+bool,+bool))?
:- modeb(1,succ(+pnat,+pnat))?
:- modeb(1,mk(+pnat,+nlist))?
:- modeb(1,mk(+pnat,[+pnat|+nlist]))?
:- modeb(1,mk(+pnat,-nlist))?
:- modeb(1,mk(+pnat,[-pnat|-nlist]))?


% Types

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

bool(t).
bool(f).

bnpair(p(B,N)) :- bool(B), pnat(N).
```

```
pcell(c(p(B,N))) :- bool(B), pnat(N).
bcell(c(B)) :- bool(B) .

nlist([]).
nlist([H|T]) :- pnat(H), nlist(T).

% Background knowledge

fst(p(B,N),B) :- bnpair(p(B,N)).
snd(p(B,N),N) :- bnpair(p(B,N)).

neg(f,t).
neg(t,f).

succ(X,s(X)) :- pnat(X).

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


% Examples

[states]?
```

Progol's input for ABP experiment

```
% Settings

:- set(posonly)?
% :- unset(cover)?
:- set(h,300000)?
:- set(r,4000000)?
:- set(nodes,20000)?
:- set(c,15)?
% :- set(i,5)?
:- set(noise,100)?


% Mode declarations

:- modeh(1,state(+bool,+pnat,+bool,+nlist,[p(+bool,+pnat)|+pqueue],[+bool|+bqueue]))?
%:- modeh(1,state(+bool,+pnat,+bool,+nlist,+pqueue,+bqueue))?
%:- modeh(1, state(+bool,+pnat,+bool,+nlist,+pqueue,+bqueue))?
%:- modeh(1,state(+bool,+pnat,+bool,+nlist,+pqueue,+bqueue))?

:- modeb(1,neg(+bool,-bool))?
:- modeb(1,succ(+pnat,-pnat))?
:- modeb(1,neg(+bool,+bool))?
:- modeb(1,succ(+pnat,+pnat))?
:- modeb(1,mk(+pnat,+nlist))?
:- modeb(1,mk(+pnat,[+pnat|+nlist]))?
:- modeb(1,mk(+pnat,-nlist))?
:- modeb(1,mk(+pnat,[-pnat|-nlist]))?
:- modeb(*,memberp(p(+bool,+pnat),+pqueue))?
:- modeb(*,memberb(+bool,+bqueue))?

%% :- modeb(1,neg(+bool,-bool))?
%% :- modeb(1,succ(+pnat,-pnat))?
%% :- modeb(1,neg(-bool,+bool))?
%% :- modeb(1,succ(-pnat,+pnat))?
%% :- modeb(1,neg(+bool,+bool))?
%% :- modeb(1,succ(+pnat,+pnat))?
%% :- modeb(*,gap0(p(+bool,+pnat),+pqueue))?
%% :- modeb(*,gap1(p(+bool,+pnat),+pqueue))?

% Types

pnat(0).
```

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

bool(t).
bool(f).

bnpair(p(B,N)) :- bool(B), pnat(N).

pqueue([]).
pqueue([H|T]) :- bnpair(H), pqueue(T).

bqueue([]).
bqueue([H|T]) :- bool(H), bqueue(T).

nlist([]).
nlist([H|T]) :- pnat(H), nlist(T).

% Background knowledge

fst(p(B,N),B) :- bnpair(p(B,N)).
snd(p(B,N),N) :- bnpair(p(B,N)).

toppqu([H|T], H) :- bnpair(H).
topbqu([H|T], H) :- bool(H).

memberp(A,[A|B]):- bnpair(A),pqueue([A|B]),!.
memberp(A,[B|C]) :- bnpair(A), pqueue([B|C]), memberp(A,C).

memberb(A,[A|B]):- bool(A),bqueue([A|B]),!.
memberb(A,[B|C]) :- bool(A), bqueue([B|C]), memberb(A,C).

neg(f,t).
neg(t,f).

succ(X,s(X)) :- pnat(X).

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

next(p(B1,N),p(B2,s(N))) :- bnpair(p(B1,N)),neg(B1,B2).

gap0(P,[]) :- bnpair(P).
gap0(P,[P|T]) :-  bnpair(P), gap0(P,T).
```

```
gap1(P,[]) :- bnpair(P).
gap1(P1,[P2|T]) :- bnpair(P1),
bnpair(P2), ((P1 \== P2, next(P2,P1), gap1(P1,T)); gap0(P1,T)).


% Examples

[states]?
```

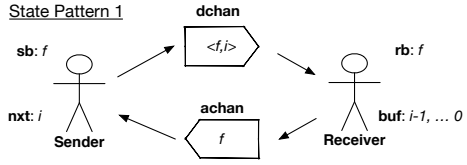
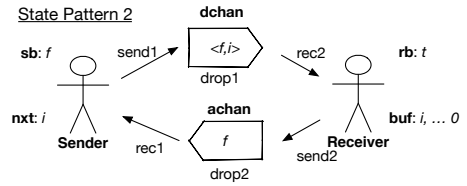
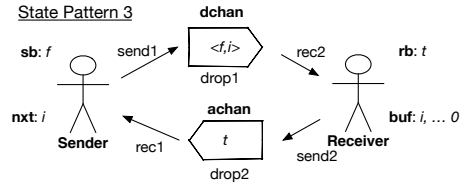
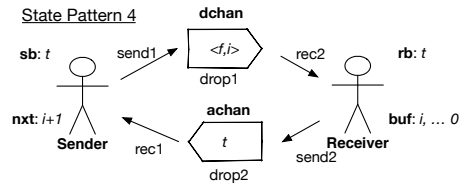
State Pattern	Characteristic
	(1) $sb = rb$ (2) $dchan = c(\langle b, n\rangle) \wedge sb = b$ (3) $dchan = c(\langle b, n\rangle) \wedge rb = b$ (4) $achan = c(b) \wedge sb = b$ (5) $achan = c(b) \wedge rb = b$ (6) $dchan = c(\langle b, n\rangle) \wedge achan = c(b') \wedge b = b'$ (7) $dchan = c(\langle b, n\rangle) \wedge nxt = n$ (8) $mk(nxt) = nxt \,\vert\, buf$
	(2) $dchan = c(\langle b, n\rangle) \wedge sb = b$ (4) $achan = c(b) \wedge sb = b$ (6) $dchan = c(\langle b, n\rangle) \wedge achan = c(b') \wedge b = b'$ (7) $dchan = c(\langle b, n\rangle) \wedge nxt = n$ (9) $sb \neq rb$ (10) $dchan = c(\langle b, n\rangle) \wedge rb \neq b$ (11) $achan = c(b) \wedge rb \neq b$ (12) $mk(nxt) = buf$
	(2) $dchan = c(\langle b, n\rangle) \wedge sb = b$ (5) $achan = c(b) \wedge rb = b$ (7) $dchan = c(\langle b, n\rangle) \wedge nxt = n$ (9) $sb \neq rb$ (10) $dchan = c(\langle b, n\rangle) \wedge rb \neq b$ (11) $achan = c(b) \wedge rb \neq b$ (12) $mk(nxt) = buf$ (13) $dchan = c(\langle b, n\rangle) \wedge achan = c(b') \wedge b \neq b'$ (14) $achan = c(b) \wedge sb \neq b$
	(1) $sb = rb$ (4) $achan = c(b) \wedge sb = b$ (5) $achan = c(b) \wedge rb = b$ (8) $mk(nxt) = nxt \,\vert\, buf$ (10) $dchan = c(\langle b, n\rangle) \wedge rb \neq b$ (13) $dchan = c(\langle b, n\rangle) \wedge achan = c(b') \wedge b \neq b'$ (15) $dchan = c(\langle b, n\rangle) \wedge sb \neq b$ (16) $dchan = c(\langle b, n\rangle) \wedge nxt \neq n$

Table 1: Characteristics captured by the four state pattern in Figure 8.

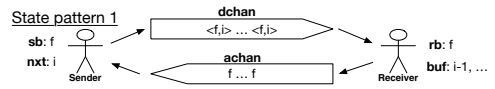
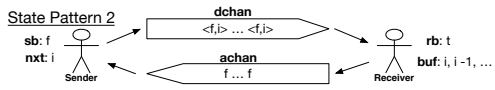
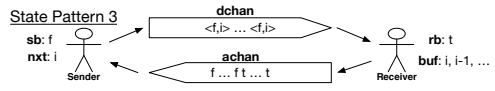
State Pattern		Characteristic
	(1)	$sb = rb$
	(2)	$dchan = (\langle b, n\rangle \mid q) \wedge sb = b$
	(3)	$dchan = (\langle b, n\rangle \mid q) \wedge rb = b$
	(4)	$achan = (b \mid q) \wedge sb = b$
	(5)	$achan = (b \mid q) \wedge rb = b$
	(6)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = b' \mid q' \wedge b = b'$
	(7)	$dchan = (\langle b, n\rangle \mid q) \wedge nxt = n$
	(8)	$mk(nxt) = nxt \mid buf$
	(9)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge b = b' \wedge n = n'$
	(10)	$b \in achan \wedge b' \in achan \wedge b = b'$
	(11)	$\langle sb, nxt\rangle \in dchan$
	(12)	$sb \in achan$
	(13)	$rb \in achan$
	(2)	$dchan = (\langle b, n\rangle \mid q) \wedge sb = b$
	(4)	$achan = (b \mid q) \wedge sb = b$
	(6)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = b' \mid q' \wedge b = b'$
	(7)	$dchan = (\langle b, n\rangle \mid q) \wedge nxt = n$
	(9)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge b = b' \wedge n = n'$
	(10)	$b \in achan \wedge b' \in achan \wedge b = b'$
	(11)	$\langle sb, nxt\rangle \in dchan$
	(12)	$sb \in achan$
	(14)	$sb \neq rb$
	(15)	$dchan = (\langle b, n\rangle \mid q) \wedge rb \neq b$
	(16)	$achan = (b \mid q) \wedge rb \neq b$
	(17)	$mk(nxt) = buf$
	(18)	$rb \notin achan$
	(2)	$dchan = (\langle b, n\rangle \mid q) \wedge sb = b$
	(4)	$achan = (b \mid q) \wedge sb = b$
	(6)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = b' \mid q' \wedge b = b'$
	(7)	$dchan = (\langle b, n\rangle \mid q) \wedge nxt = n$
	(9)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge b = b' \wedge n = n'$
	(11)	$\langle sb, nxt\rangle \in dchan$
	(12)	$sb \in achan$
	(13)	$rb \in achan$
	(14)	$sb \neq rb$
	(15)	$dchan = (\langle b, n\rangle \mid q) \wedge rb \neq b$
	(16)	$achan = (b \mid q) \wedge rb \neq b$
	(17)	$mk(nxt) = buf$
	(19)	$b \in achan \wedge b' \in achan \wedge (b = b' \vee b \neq b')$
	(20)	$achan = (q1@q2) \wedge q1 = (b \mid q) \wedge q2 = (b' \mid q')$ $\wedge b \neq b' \wedge b \notin q2 \wedge b' \notin q1$

Table 2: Characteristics captured by the four state pattern in Figure 9.

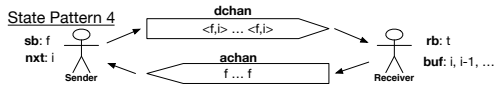
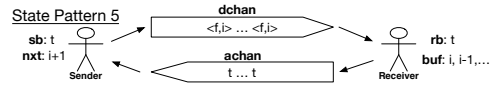
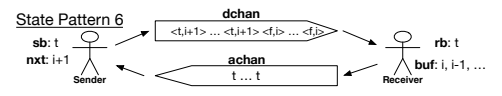
State Pattern	Characteristic	
	(2)	$dchan = (\langle b, n\rangle \mid q) \wedge sb = b$
	(4)	$achan = (b \mid q) \wedge sb = b$
	(6)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = b' \mid q' \wedge b = b'$
	(7)	$dchan = (\langle b, n\rangle \mid q) \wedge nxt = n$
	(9)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge b = b' \wedge n = n'$
	(10)	$b \in achan \wedge b' \in achan \wedge b = b'$
	(11)	$\langle sb, nxt\rangle \in dchan$
	(13)	$rb \in achan$
	(14)	$sb \neq rb$
	(15)	$dchan = (\langle b, n\rangle \mid q) \wedge rb \neq b$
	(17)	$mk(nxt) = buf$
	(21)	$achan = (b \mid q) \wedge rb = b$
	(22)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = (b' \mid q') \wedge b \neq b'$
	(23)	$sb \notin achan$
	(24)	$dchan = (\langle b, n\rangle \mid q) \wedge sb \neq b$
	(4)	$achan = (b \mid q) \wedge sb = b$
	(5)	$achan = (b \mid q) \wedge rb = b$
	(6)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = b' \mid q' \wedge b = b'$
	(25)	$dchan = (\langle b, n\rangle \mid q) \wedge nxt \neq n$
	(9)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge b = b' \wedge n = n'$
	(26)	$\langle sb, nxt\rangle \notin dchan$
	(12)	$sb \in achan$
	(13)	$rb \in achan$
	(1)	$sb = rb$
	(8)	$mk(nxt) = nxt \mid buf$
	(24)	$dchan = (\langle b, n\rangle \mid q) \wedge sb \neq b$
	(4)	$achan = (b \mid q) \wedge sb = b$
	(5)	$achan = (b \mid q) \wedge rb = b$
	(6)	$dchan = (\langle b, n\rangle \mid q) \wedge achan = b' \mid q' \wedge b = b'$
	(25)	$dchan = (\langle b, n\rangle \mid q) \wedge nxt \neq n$
	(9)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge b = b' \wedge n = n'$
	(11)	$\langle sb, nxt\rangle \in dchan$
	(12)	$sb \in achan$
	(13)	$rb \in achan$
	(1)	$sb = rb$
	(8)	$mk(nxt) = nxt \mid buf$
	(27)	$\langle b, n\rangle \in dchan \wedge \langle b', n'\rangle \in dchan \wedge$ $\wedge((b = b' \wedge n = n') \vee (b \neq b' \wedge n \neq n'))$
	(28)	$dc = (q1@q2) \wedge \langle b, n\rangle \in q1 \wedge \langle b', n'\rangle \in q2$ $\wedge b \neq b' \wedge (n+1) = n' \wedge \langle b, n\rangle \notin q2 \wedge \langle b', n'\rangle \notin q1)$

Table 3: Characteristics captured by the four state pattern in Figure 9. (cont.)