International Journal of Future Generation Communication and Networking
Vol. 8, No. 5 (2015), pp. 177-188
http://dx.doi.org/10.14257/ijfgcn.2015.8.5.18

An Automatic Software Requirement Analysis Model based on Planning and Machine Learning Techniques

Tienan Zhang[*]

School of Computer & Communication, Hunan Institute of Engineering
[*]ztnan1974@126.com

Abstract

In the past years, the scale of software is growing quickly as more and more organizations begin to deploy their business on Internet. As a result, requirement analysis becomes a challenging issue and conventional approaches might significantly increase the costs of software development. Therefore, automatic requirement analysis techniques have attract more and more attentions, which allows for modeling and analyzing requirements formally, rapidly and automatically, avoiding mistakes made by misunderstanding between engineers and users, and saving lots of time and manpower. In this paper, we propose an approach to acquiring requirements automatically, which adopts automated planning techniques and machine learning methods to convert software requirement into an incomplete planning domain. By this approach, we design an algorithm called Intelligent Planning based Requirement Analysis (IPRA), to learn action models with uncertain effects. A concrete experiment is conducted to investigate the proposed algorithm, and the results indicate that it can obtain a complete planning domain and convert it into software requirement specification.

Keywords: *Requirement Analysis; Machine Learning; Intelligent Planning; Cloud Computing; Uncertain Theory*

1. Introduction

Acquisition of software requirements based on ontology [1] is one of hot topics currently, which focuses on inducing users to offer system information with normal situation examples. Since those examples are collected randomly, it is difficult to make sure that a group of situation examples can cover the whole system, and induce users to offer requirement information completely and exactly, therefore this method cannot be applied generally. In this paper, we focus on applying artificial intelligent methods to acquire software requirement specification automatically, which will make great difference in practice to avoid incomplete information and misunderstanding.

Table 1. An Action Model in Slippery-Gripper Domain

```
action pickup
  : parameters(?b ?g)
  : preconditions((block ?b) (gripper ?g))
   : effects( <(0.95 (gripper-dry ?g) (and (holding-block ?b))),
                 (0.05 (gripper-dry ?g) (and(not(holding-block ?b))))>
                   <(0.5 (not (gripper-dry ?g)) (and (holding-block ?b))),
                        (0.5 (not (gripper-dry ?g)) (and(not(holding-block
                                          ?b))))>)
```

In [2], the authors proposed an algorithm called LAMPS to learn action models, however it can only be applied when actions have certain effects. However, actions usually have multiple effects with uncertainty. For example, in slippery-gripper domain, whenever gripper is dry or wet, there are two possible effects which are holding block or not holding block. Therefore, action model of action "pickup" can be described formally as shown in Table 1. When gripper is dry, the probability of holding block is 0.95, and that of not holding block is 0.05; when gripper is wet, the probability of holding block and not holding block are both 0.5.

Generally, we normally assume that action models with conditional effects and probabilistic effects could be built manually by experience, but in fact, it is difficult even for experts. It requires that experts not only should grasp logic of domain, but also have enough prior knowledge. Therefore, we propose an algorithm called Intelligent Planning based Requirement Analysis (IPRA) to learn action models with conditional effects and probabilistic effects and apply this algorithm to acquire software requirement automatically. Compared with previous action model learning algorithms, IPRA make the following contributions: (1) obtained action models by IPRA could have uncertain effects, including conditional effects and probabilistic effects. In practice, effects of actions are usually uncertain and conditional, with multiple possibilities. (2) state information of the planning traces could be incomplete. It is difficult to obtain complete state information in reality. IPRA can be applied with incomplete state information.

The rest of this paper is organized as follows. In Section 2, we introduce related work. In Section 3 and Section 4, we make problem definition and present the steps of algorithm IPRA in detail. In Section 5, we construct experiments in four planning domains to estimate the error rates of learned action models by IPRA, and apply IPRA algorithm to acquire software requirement specification. In Section 6, we summarize this paper and discuss our future works.

2. Related Work

Automated planning systems achieve goals by producing sequences of actions from given action models that are provided as input. In [3], Fikes and Nils designed STRIPS system to introduce definitions of STRIPS operators, which made significant difference in the research of automated planning. In [4], the authors designed the first nonlinear planning system SNLP of the world. In [5], Kautz converted planning into SAT problem, which effectively solved partial planning problem and showed new direction of automated planning. In [6], the authors designed the first graph planner system Graphplan to solve planning problem, and proposed concept of graph plan. In 1998, Malik proposed Plan Domain Definition Language (PDDL) [7], and then PDDL gradually became a general standard of representing domain models and was applied broadly in international planning competitions. In [8], the authors proposed PPDDL1.0 to solve probabilistic planning problems with uncertain effects.

A traditional way of building action models is to ask domain experts to analyze a task domain and manually design domain description that includes a set of complete action models. Then planning system is proceeded to generate action sequences to achieve goals. At the same time, many researchers have proposed some algorithms to learn action models. According to whether state information is complete, these algorithms could be divided into two parts. Some algorithms are, to learn action models from plan races with complete state information [9-16], which means for each action, we obtain the state information before and after it happens in advance, and then learn preconditions and effects of action model by statistics and reasoning. Gil *et al.*, [9] build EXPO system, bootstrapped by an incomplete STRIPS-like domain description with the rest being filled in through experience. Oates *et al.*, [10] use a general classification system to learn preconditions and effects of actions.

Schmill *et al.*, [11] learn action models by approximate computation in relative domains. Wang et al. [12] propose an approach to learn action model automatically by observing planning traces and refine the operators through practice in a learning-by-doing paradigm. Pasula *et al.*, [13, 14] present how to learn stochastic action models without conditional effects. Holmes *et al.*, [15] model synthetic items based on experience to build action models. Walsh *et al.*, [16] propose an efficient algorithm to learn action models for describing Web services.

The other algorithms are to learn action models with incomplete state information [2, 17-29], which means preconditions and effects can be learned when the state information before and after each action is not fully observed. Yang *et al.*, [17] build an action model learning system-ARMS by using SAT, which can handle the case with incomplete state information, and acquire STRIPS action models. Amir et al. propose the algorithm SLAF [18] to learn action models with incomplete state information, which resembles version spaces and logical filtering and identifies all the models that are consistent sequence of executed actions. To improve the performance of SLAF, Shahaf *et al.*, [19] build an efficient algorithm to learn preconditions and effects of deterministic action models. Lai *et al.*, [22] construct AMLS system to learn action models from planning traces with incomplete state information, under the framework of genetic algorithm. For acquiring action models with better expression ability, Zhuo *et al.*, [2] build a novel learning algorithm LAMP, which applies Markov logic networks to obtain action models with quantifiers and logical implications. This paper proposes IPRA algorithm on the basis of LAMP, focusing on learning action models with conditional effects and probabilistic effects in indeterminate planning domains.

3. Problem Description

A STRIPS-like planning problem with conditional effects and probabilistic effects can be defined as a four-tuple $<S,s_0,s_g,O>$, where S represents a set of states, and each state is a set of propositions; s_0 represents the initial state and s_g represents the goal state which is the final state following with a series of states transition, starting with initial state; O represents a set of action models with conditional effects and probabilistic effects. In this paper, we note O as a three-tuple $<a, PRE, CPEFF>$, where a represents an action schema with action name and parameters, PRE represents preconditions, $CPEFF$ represents conditional effects and probabilistic effects.

Normally, $CPEFF$ can formally be expressed as $<(p_{i1},c_i,e_{i1})...(p_{ij},c_i,e_{ij})\ldots(p_{in},c_i,e_{in})>$, where c_i represents the ith condition composed of literal and conditions $c_{i\,(1\le i\le k)}$ are mutually exclusive, the corresponding jth effect is represented by e_{ij} with probability p_{ij}, which is a conjunction of literal and $\sum_{j=1}^{n} p_{i_j}=1, p_{ij}\ge 0$. In the case when condition c_i is empty, conditional effects are exactly equal to probabilistic effects. If preconditions of an action are satisfied in state s, then the action can be applied in state s, and its effects can be selected according to conditions and probabilities. A possible action sequence is denoted as $<a_1,a_2,\ldots,a_n>$, transferring from initial state s_0 to goal state s_g. Furthermore, we call $(s_0,a_1,s_1,a_2,\ldots,s_n,a_n,s_g)$ as a planning trace, where the middle state s_i might be null, and a_i represents action schema.

Action model learning with conditional effects and probabilistic effects can be described as follows. Given planning traces set T, propositions set P as input, algorithm IPRA outputs all the action models with conditional effects and probabilistic effects in A. We show an example of action model learning with conditional effects and probabilistic effects in Table 2. The example is chosen from the domain slippery-gripper, an indeterminate planning domain, where input information is shown in Table 2, and output information is shown in Table 1.

Table 2. An Example Input in IPRA

Input: Predicates *P* *(block ?b) (gripper ?g) (gripper-dry ?g) (holding-block ?b) (block-painted ?b) (gripper-clean ?g)*			
Input: Action Schemas *A* *(pickup ?b ?g) (dry ?g) (paint ?b ?g)*			
Input: Plan Traces *T*			
	Trace 1	Trace 2	Trace 3
Initial state	*(gripper G) (block B) (gripper-clean G) (gripper-dry G)*	*(gripper G) (block B) (gripper-clean)*	*(gripper G) (block B) (gripper-clean)*
Action 1	(paint B G)	(pickup B G)	(pickup B G)
Observation 1		*not(holding-block B)*	*(holding-block B)*
Action 2	(pickup B G)	(dry G)	(paint B G)
Observation 2			
Action 3		(pickup B G)	
Observation 3			
Action 4		(paint G)	
Goal state	*(gripper-clean G) (holding-block B) (block-painted B)*	*not(gripper-clean G) (holding-block B) (block-painted B)*	*(gripper-clean G) (holding-block B) (block-painted B)*

4. Framework of IPRA Algorithm

The motivation of our algorithm IPRA is to transform the action model learning problem into weights learning problem in MLNs, and obtain action models with conditional effects and probabilistic effects. The frameworks of algorithm IPRA is shown in Table 3. In the following subsections, we will show a detailed description of each step of the algorithm IPRA.

Table 3. Framework of Algorithm IPRA

Input: 1. A set of plan traces *T*; 2. A set of action schema *A*; 3. A set of predicates *P*. **Output:** 1. A set of action models with conditional effects and probabilistic effects.
Main Steps: 1. Encode each plan trace as a set of propositions; 2. Generate candidate formulas, using *A* and *P*; 3. Apply MLNs to learn weights of all the candidate formulas; 4. Choose some of candidate formulas according to given threshold, and convert weighted candidate formulas to action models with conditional effects and probabilistic effects as output.

4.1. Encode Plan Traces

In the first step of algorithm IPRA, we encode all the plan traces as a set of proposition databases *DBs* with plan traces *T* as input. Firstly, we use propositions to represent each state of plan traces. For example, consider domain slippery-gripper in table 1, which includes two objects *B* and *G*. Present state s_1, describing that *B* is a block, *G* is a gripper, and *G* is clean, can be represented as *(block B s_1)* $\wedge$ *(gripper G s_1)* $\wedge$ *(gripper-clean G s_1)*. Secondly, we can consider an action as transition of states, then action can be encoded as the conjunction of propositions. For example, the action *(pickup B G s_1)* in table 1 can be treated as transition from the state *(block B s_1)* $\wedge$ *(gripper G s_1)* $\wedge$ *(gripper-clean G s_1)* to the state *(holding-block B s_2)*, then the action *(pickup B G s_1)* can be encoded as: *(block B s_1)* $\wedge$ *(gripper G s_1)* $\wedge$ *(gripper-clean G s_1)* $\wedge$ *(pickup B G s_1)* $\wedge$ *(holding-block B s_2)*.

According to the above method, we can encode each plan trace into a conjunction of grounded literals, and then convert them into a database, where each record in a *DB* is a ground literal, and records are related as conjunction. For the sake of simplicity, we use *i* to denote the state symbol s_i. As an example, we encode the plan traces in Table 2 as database, shown in Table 4. In the paper, we make open world assumption, which means the grounded literal not shown in Table 4 is considered as unknown.

Table 4. Encodings of Plan Traces as Databases

DB1	DB2	DB3
(gripper G 0) *(block B 0)* *(gripper-clean G 0)* *(gripper-dry G 0)* *(paint B G 0)* *(pickup B G 1)* *(gripper-clean G 2)* *(holding-block B 2)* *(block-painted B 2)*	*(gripper G 0)* *(block B 0)* *(gripper-clean G 0)* *(pickup B G 0)* *not(holding-block B 1)* *(dry G 1)* *(pickup B G 2)* *(paint G 3)* *not(gripper-clean G 4)* *(holding-block B 4)* *(block-painted B 4)*	*(gripper G 0)* *(block B 0)* *(gripper-clean G 0)* *(pickup B G 0)* *(holding-block B 1)* *(paint B G 1)* *not(gripper-clean G 2)* *(holding-block B 2)* *(block-painted B 2)*

4.2. Generate Candidate Formulas of Actions

In STRIPS model, if a predicate is a negative effect of an action, then the predicate should be a precondition of the action; and a predicate can not be both positive effect and negative effect of an action. Considering the two characteristics, we describe an action model in two parts:

(1) Preconditions. If predicate *p* is a precondition of action *a*, then *p* must be satisfied when the action *a* is executed, which can be described formally as:

$$\forall i,\bar{x},\bar{y},a(\bar{x},i) \rightarrow p(\bar{y},i), \tag{1}$$

where $\bar{x}$, $\bar{y}$ are parameters, and *i* is the state symbol. In formula (1), since $p(\bar{y},i)$ is a necessary condition, not a sufficient condition, we choose $p(\bar{y},i)$ from candidate formulas with weights bigger than some threshold as preconditions in action model.

(2) Conditional effects. If predicate *p* is a positive effect of action *a* with condition *c*, then *p* should be added to the next state after the action *a* when condition *c* is satisfied, which can be described formally as:

$$\forall i,\bar{x},\bar{y},a(\bar{x},i) \rightarrow p(\bar{y},i) \wedge p(\bar{y},i+1) \wedge c(\bar{z},i) \tag{2}$$

where $\bar{x}, \bar{y}, \bar{z}$ are parameters, and *i* is the state symbol. If predicate *q* is a negative effect of action *a* with condition *c*, then *q* is satisfied when *a* is executing and condition *c* is satisfied, but not satisfied after action *a*, which can be described formally as:

$$\forall i,\bar{x},\bar{y},a(\bar{x},i) \rightarrow q(\bar{y},i) \wedge \neg q(\bar{y},i+1) \wedge c(\bar{z},i) \tag{3}$$

where $\bar{x}, \bar{y}, \bar{z}$ are parameters, and *i* is the state symbol. Similarly, suppose action *a* has a positive effect *p* and *a* negative effect *q* with condition *c*, then it can be described formally as

$$\forall i,\bar{x},\bar{y},a(\bar{x},i) \rightarrow \neg p(\bar{y},i) \wedge p(\bar{y},i+1) \wedge q(\bar{y},i) \wedge \neg q(\bar{y},i+1) \wedge c(\bar{z},i) \tag{4}$$

where $\bar{x}, \bar{y}, \bar{z}$ are parameters, and *i* is the state symbol, which means that effects of an action can be described as conjunction of some atomic formulas.

Applying formula (1) and (4), we can acquire candidate formulas of preconditions and conditional effects. For example, in slippery-gripper domain, candidate formulas of preconditions and conditional effects of action *pickup* are shown (5) and (6)

$$\begin{cases} \forall i,b,g,(pickup \quad b \quad g \quad i) \rightarrow (gripper \quad g \quad i) \\ \forall i,b,g,(pickup \quad b \quad g \quad i) \rightarrow (block \quad b \quad i) \\ \forall i,b,g,(pickup \quad b \quad g \quad i) \rightarrow (gripper\text{-}dry \quad g \quad i) \\ \forall i,b,g,(pickup \quad b \quad g \quad i) \rightarrow (holding\text{-}block \quad b \quad i) \end{cases} \tag{5}$$

$$\begin{cases} \forall i.b.(pickup \quad b \quad g \quad i) \rightarrow (gripper\text{-}dry \quad g \quad i) \wedge \neg(holding\text{-}block \quad b \quad i) \\ \qquad \wedge (holding\text{-}block \quad b \quad i+1) \\ \forall i.b.(pickup \quad b \quad g \quad i) \rightarrow (gripper\text{-}dry \quad g \quad i) \wedge (holding\text{-}block \quad b \quad i) \\ \qquad \wedge \neg(holding\text{-}block \quad b \quad i+1) \\ \forall i.b.(pickup \quad b \quad g \quad i) \rightarrow \neg(gripper\text{-}dry \quad g \quad i) \wedge \neg(holding\text{-}block \quad b \quad i) \\ \qquad \wedge (holding\text{-}block \quad b \quad i+1) \\ \forall i.b.(pickup \quad b \quad g \quad i) \rightarrow \neg(gripper\text{-}dry \quad g \quad i) \wedge (holding\text{-}block \quad b \quad i) \\ \qquad \wedge \neg(holding\text{-}block \quad b \quad i+1) \\ \cdots\cdots \end{cases} \tag{6}$$

4.3. Learn Weights of Candidate Formulas

According to reference [23], Markov Logic Networks L consists of a set of pairs (F_i,ω_i), where F_i is a formula in first-order logic and ω_i is a real number. With a finite set of constants $C=\{c_1,c_2,\cdots,c_n\}$, it defines a Markov network $M_{L,C}$ as following steps: (1) $M_{L,C}$ contains one binary node for each possible grounding of each predicate appearing in L.The value of the node is 1, if the grounded predicate is true, and 0 otherwise; (2) $M_{L,C}$ contains one feature for each possible grounding of each formula F_i in L. The value of this feature is 1 if the ground formula is true, and 0 otherwise. The weight of the feature is ω_i associated with F_i in L.

We apply Alchemy system [24] to learn weights of candidate formulas, by using weighted optimized pseudo log-likelihood. For each atomic formula, if it appears in DBs, then it corresponds to $x_i=1$, otherwise 0. As mentioned in step 2, we can obtain the candidate formulas of preconditions and effects of actions by (1), (4), then learn weights of all the candidate formulas by MLNs. For example, the weights of candidate formulas in (5) and (6) are shown as (7) and (8)

$$\begin{cases} \forall i,b,g,(pickup\ \ b\ \ g\ \ i) \to (gripper\ \ g\ \ i), when\ w = 0.3 \\ \forall i,b,g,(pickup\ \ b\ \ g\ \ i) \to (block\ \ b\ \ i), when\ w = 0.4 \\ \forall i,b,g,(pickup\ \ b\ \ g\ \ i) \to (gripper\text{-}dry\ \ g\ \ i), when\ w = -0.4 \\ \forall i,b,g,(pickup\ \ b\ \ g\ \ i) \to (holding\text{-}block\ \ b\ \ i), when\ w = -0.2 \end{cases} \tag{7}$$

$$\begin{cases} \forall i.b.(pickup\ \ b\ \ g\ \ i) \to (gripper\text{-}dry\ \ g\ \ i) \wedge \neg(holding\text{-}block\ \ b\ \ i) \\ \qquad \wedge (holding\text{-}block\ \ b\ \ i+1), where\ w = 0.77 \\ \forall i.b.(pickup\ \ b\ \ g\ \ i) \to (gripper\text{-}dry\ \ g\ \ i) \wedge (holding\text{-}block\ \ b\ \ i) \\ \qquad \wedge \neg(holding\text{-}block\ \ b\ \ i+1), where\ w = 0.12 \\ \forall i.b.(pickup\ \ b\ \ g\ \ i) \to \neg(gripper\text{-}dry\ \ g\ \ i) \wedge \neg(holding\text{-}block\ \ b\ \ i) \\ \qquad \wedge (holding\text{-}block\ \ b\ \ i+1), where\ w = 0.44 \\ \forall i.b.(pickup\ \ b\ \ g\ \ i) \to \neg(gripper\text{-}dry\ \ g\ \ i) \wedge (holding\text{-}block\ \ b\ \ i) \\ \qquad \wedge \neg(holding\text{-}block\ \ b\ \ i+1), where\ w = 0.47 \end{cases} \tag{8}$$

4.4. Obtain Action Model

In the candidate formulas of preconditions, we choose those formulas with weights bigger than some threshold as a set and convert the set into the preconditions of action model. Similarly, we can choose some candidate formulas of conditional effects and calculate their corresponding probabilities. Finally, we can obtain action model with probabilistic conditional effects. Weight of a formula in MLNs reflects the level of truth, which means the higher weight, the more formulas with true value after instantiation. At the beginning, we need to decide a threshold of the weights. For example, we set the threshold to be 0, then we can choose all the formulas with weights bigger than 0 in (7) as

$$\begin{cases} \forall i,b,g,(pickup\ \ b\ \ g\ \ i) \to (gripper\ \ g\ \ i) \\ \forall i,b,g,(pickup\ \ b\ \ g\ \ i) \to (block\ \ b\ \ i) \end{cases} \tag{9}$$

Therefore, predicates $(gripper\ \ g\ \ i),(block\ \ b\ \ i)$ are the preconditions of action $(pickup\ \ b\ \ i)$. Similarly, we choose those formulas under the same condition, with weights bigger than 0 in (8), and calculate their corresponding probabilities, then we can acquire the action model of $(pickup\ \ b\ \ i)$ with probabilistic and conditional effects as shown in Table 5.

Table 5. Acquired Action Model

Action schema	*pickup(?b ?g)*
Preconditions	*block(?b), gripper(?g)*
Probabilistic conditional	*<(0.87 (gripper-dry ?g) (and (holding-block ?b))), (0.13 (gripper-dry ?g) (and(not(holding-block ?b))))> <(0.48 (not (gripper-dry ?g)) (and (holding-block ?b))), (0.52 (not (gripper-dry ?g)) (and(not(holding-block ?b))))>*

5. Experiments and Evaluation

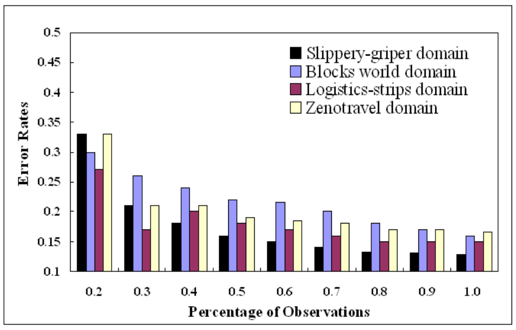
5.1. Settings

Firstly, we define the error rates of our algorithm as follows: (1) Error rates of preconditions: let the number of all the possible preconditions in action models be N_{pre} ,the set of preconditions of learned action models be T'_{pre} ,and the set of preconditions of correct action models be T''_{pre} .If a precondition belongs to T'_{pre} , not T''_{pre} , then the number of errors in preconditions denoted by E_{pre} ,adds one; similarly if a precondition belongs to T''_{pre} ,not T'_{pre} , E_{pre} adds one. Then the number of errors in preconditions can be expressed as $n_{pre} = \left| T'_{pre} \cup T''_{pre} - T'_{pre} \cap T''_{pre} \right|$ .Thus error rate of preconditions can be calculated as $P_{pre} = \frac{n_{pre}}{N_{pre}}$. (2) Error rates of effects: since the learned action models have probabilistic effects, then we calculate the error rates of effects in a different method.

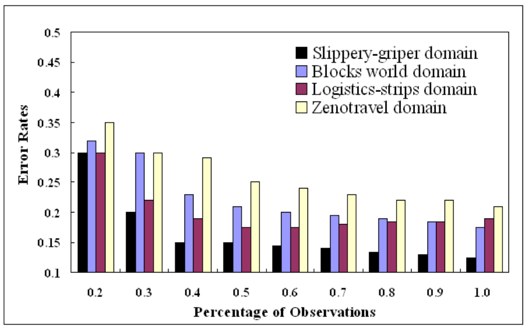
To evaluate the algorithm IPRA, we collected plan traces from the following planning domains: slippery-gripper, blocks-world, zenotravel, logistics-strips. These domains have the characteristics we need to evaluate in IPRA algorithm: all the four domains have uncertain effects. Using probabilistic planner Probabilistic-FF, we generated 20-100 planning traces from the three domains, as training data of learning action models with probabilistic effects. We consider the given action models in the above web-page as correct one, and then use the correct action models to evaluate the error rates of learned action models.

5.2. Accuracy and Observed Intermediate States

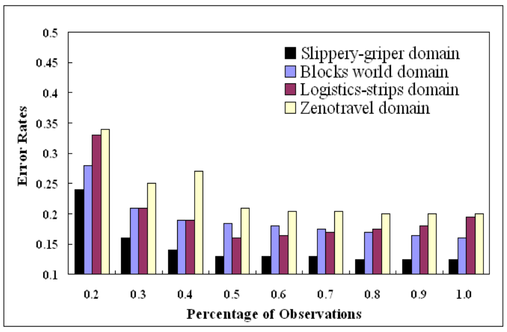
To simulate partial observation between two actions in a plan trace, from the plan traces, we randomly select observed states with specific percentage of observations 1/5, 1/4, 1/3, 1/2, 1. For each percentage value, *e.g.*, 1/3, we randomly select an observation within three consecutive states in a plan trace. We run the selection process three times. IPRA generates learned action models each time, and meanwhile error rates are calculated. Finally, we calculate an average error rate on the plan traces. The results of these tests are shown in Figure 1.



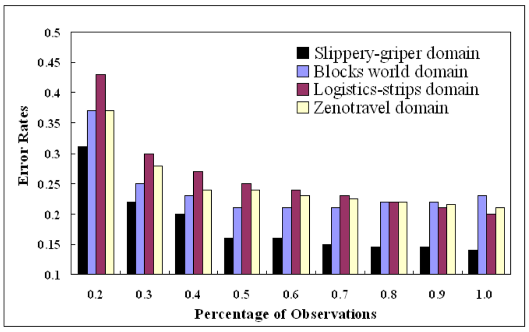
(a) Threhold t = 0.5



(b) Threhold t = 0.01



(c) Threhold t = 0.5



(d) Threhold t = 0.001

Figure 1. Error Rates of Learned Action Models in Different Domains

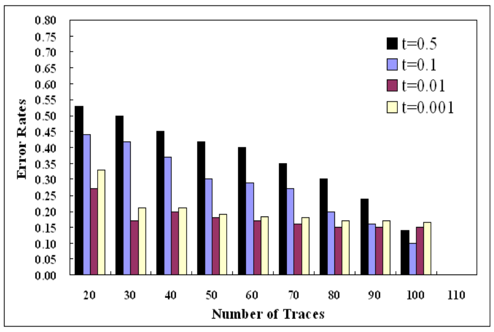
Figure 1 shows the performance of the IPRA algorithm with respect to different threshold values *t* used to select the candidate formulas, which are set to be 0.001, 0.01, 0.1 and 0.5, respectively. From the results, we find that error rate is sensitive to the choice of threshold. Generally, thresholds shall not be set to be extremely smaller or bigger. A bigger threshold will miss out some useful formulas; meanwhile a smaller threshold will cover some formulas with noise. From these experiments, it is shown that when the threshold is set to be 0.1, the mean average accuracy is optimal. Furthermore, the error bars representing the confidence intervals, show that our algorithm performance is stable.

The result also shows the relationship between the accuracy of learned model and percentage of observed intermediate states. In most cases, the more observations we have, the lower the error rate will be, which is consistent with our intuition. However, there are some cases, e.g., when threshold *t* is set to be 0.5, and there are only 1/4 of the states observed, the error rate is lower than the case when 1/3 of the states are given. These cases are not consistent with our intuition, but they are possible, since when more observations are obtained, the weights of their corresponding formulas go up and the weights of other formulas may go down in the whole learning process. Thus, if the threshold *t* is still set to be 0.5, some formulas which were chosen before are missed out, and the error rate will be higher. Thus, we conclude that in these cases, we need to reduce the value of the threshold correspondingly to make the error rate lower.

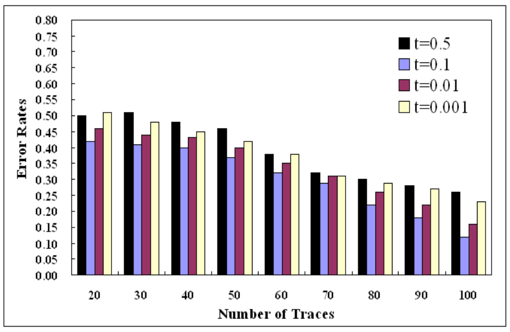
When threshold is set to 0.1, comparing the error rates of the four domains, it is obviously observed that the error rate of more complicated domain (with more predicates and actions) is generally higher than that of other domains, while with the increase of the number of plan traces, the error rate will decrease to about 10%. The reason is that in those complicated domains, a large number of predicates and actions will result in more candidate formulas of preconditions and conditional effects. In this case, if we don’t have enough number of plan traces, then the noise in the experimental result will be quite serious. Therefore, in those complicated plan domains, the number of plan traces should be at least 100.

5.3. Plan Traces in Action-Model Learning

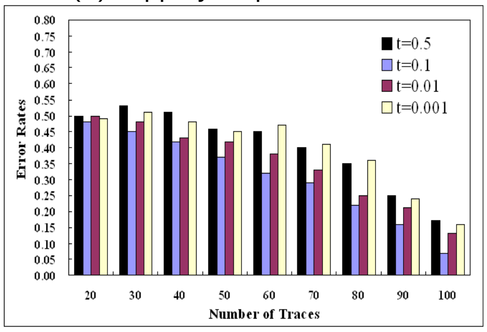
To see how error rate are affected by the number of plan traces, we used different number of plan traces as the training data to evaluate the performance. In experiments, we assume that each plan trace had 1/5 of fully observed intermediate states. These observed states were randomly selected. The process of generating state observations is repeated five times, where each time an error rate is generated under different selections. Figure 2 shows that error rates are affected by the number of given plan traces.



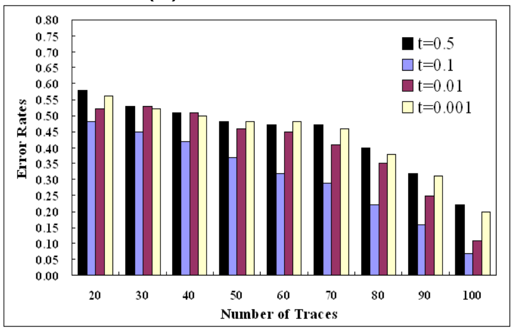
(A) Slippery-Griper Domain



(B) Blocks World Domain



(C) Logistics-Strips Domain



(D) Zenotravel Domain

Figure 2. Error Rates of Different Number of Plan Traces

Generally, error rate decreases when the number of plan traces increase. When the number of plan traces is smaller, the error rate is higher. When the number of plan traces increases to some extent, the error rate decreases rapidly, but eventually it goes down slowly. It means that the difference between learned action models and correct ones is obvious when information is limited, but the difference will decrease when enough information is available. It can be speculated that learned action models will be approximate to correct ones, when enough number of plan traces is available.

When threshold is set to 0.1, comparing the error rates in the four domains, it is obviously observed that the error rate of the more complex domain (with more predicates and actions) is generally higher than the others. With the increase of the number of plan traces, the error rate will decrease to lower than 10%. The reason is that in those complex domains, a large number of predicates and actions will result in more candidate formulas of preconditions and effects. In this case, if we have not enough number of plan traces, then the noise in the experimental result will be quite serious. Therefore, in those complicated plan domains, the number of plan traces will be more than 100.

6. Conclusion

In this paper, we proposed an automated planning and machine learning to translate software requirements into partial planning domain, formally described by PDDL language. Then we build up an action model learning algorithm to obtain complete planning domain and requirements specification. The proposed method can be used to acquire software requirement automatically. In future, we are planning to improve IPRA algorithm to apply it in the problem of system re-configuration at runtime.

References

[1] K. D. Mu, W. Liu and Z. Jin, "Managing Software Requirements Changes Based On Negotiation-Style Revision", Journal Of Computer Science And Technology, vol. 26, no. 5, **(2011)**, pp. 890-907.

[2] H.-G. Chen, J. J. Jiang and G. Klein, "Reducing Software Requirement Perception Gaps Through Coordination Mechanisms", Journal of Systems and Software, vol. 82, no. 4, **(2009)**, pp. 650-655.

[3] R. Fikes and J. N. Nils, "Strips: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence, vol. 2, no. 3, (2009), pp. 189-203.

[4] L. Castillo and L. Morales and A. Gonzalez-Ferrer, "Automatic Generation of Temporal Planning Domains for E-Learning Problems", Journal of Scheduling, vol. 13, no. 4, **(2010)**, pp. 347-362.

[5] H. Kautz, D. Mcallester and B. Selman, "Encoding Plans in Propositional Logic", Proceedings of the 5th International Conference of Principles of Knowledge Representation and Reasoning, **(1996)**, pp. 1084-1090.

[6] K.-I. Kawarabayashi and Y. Kobayashi, "A Linear Time Algorithm for the Induced Disjoint Paths Problem in Planar Graphs", Journal of Computer and System Sciences, vol. 78, no. 2, **(2012)**, pp. 670-680.

[7] G. Malik, H. Adele and K. Craig, "Pddl-The Planning Domain Definition Language", Http://Www.Informatik.Uni-Ulm.De/Ki/Edu/Vorlesungen/Gdki/Ws0203/Pddl.Pdf, 1998.

[8] D. Smith and D. Weld, "Confor M Ant Graphplan", Proceeding of 15th National Conference on Artificial Intelligence, **(1998)**, pp. 133-145.

[9] D. Weld, C. Anderson and D. Smith, "Extending Graph Plan to Handle Uncertainty and Sensing Actions", Proceedings of 15th National Conference on Artificial Intelligence, **(1998)**, pp. 103-110.

[10] Y. Chen, "Constrained Partitioning In Penalty Formulations For Solving Temporal Planning Problems", Artificial Intelligence, vol. 170, no. 3, **(2006)**, pp. 187-231.

[11] L. Philipple, "Algorithms For Propagating Resource Constraints In Ai Planning And Scheduling: Existing Approaches And New Results", Artificial Intelligence, vol. 143, no. 2, **(2003)**, pp. 151-188.

[12] M. Omid, H. Steve and C. Anne, "On the Undecidability of Probabilistic Planning and Related Stochastic Optimization Problems", Artificial Intelligence, vol. 147, no. 2, **(2003)**, pp. 5-34.

[13] H. Younes, M. L. Littman and D. Weissman, "The First Probabilistic Track Of The International Planning Competition", Journal Of Artificial Intelligence Research, vol. 24, no. 3, **(2005)**, pp. 851-887.

[14] J. Zhou, M. Yin and W. Gu, "Research on Decreasing Observation Variables For Strong Planning Under Partial Observation", Journal of Software, vol. 20, no. 2, **(2009)**, pp. 290-304.

[15] S. Yan, M. Yin, G. Gu and X. Liu, "Research And Advances In Probabilistic Planning", Caai Transactions on Intelligence Systems, vol. 3, no. 1, **(2008)**, pp. 9-22.

[16] G. Yolanda, "Learning By Experimentation: Incremental Refinement of Incomplete Planning Domains", Proceeding of the Eleventh International Conference on Machine Learning, **(1994)**, pp. 87-95.

[17] O. Tim and R. C. Paul, "Searching For Planning Operators With Context-Dependent and Probabilistic Effects", Proceedings of the Thirteenth National Conference on Artificial Intelligence, **(1996)**, pp. 865-868.

[18] D. Matthew, O. Tim and R. C. Paul, "Learning Planning Operates In Real-World, Partially Observable Environments", Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, **(2000)**, pp. 246-253.

[19] W. Xuemei, "Learning by Observation And Practice: An Incremental Approach for Planning Operator Acquisition", Proceeding of the Twelfth International Conference on Machine Learning, **(1995)**, pp. 549-557.

[20] M. Staron and W. Meding, "Predicting Weekly Defect Inflow In Large Software Projects Based On Project Planning And Test Status", Information And Software Technology, vol. 50, no. 7-8, **(2008)**, pp. 782-796.

[21] I. Woungang, F. O. Akinladejo and D. W. White, "Coding-Error Based Defects in Enterprise Resource Planning Software: Prevention, Discovery, Elimination and Mitigation", Journal of Systems and Software, vol. 85, no. 7, **(2012)**, pp. 1682-1698.

[22] P. H. Michael and L. J. Charles, "Schema Learning: Experience-Based Construction of Predictive Action Models", Advances in Neural Information Processing Systems (Nips 2004), **(2004)**.

[23] J. W Thomas and L. L. Michael, "Efficient Learning of Action Schema and Web-Service Descriptions", In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008), **(2008)**, pp. 714-719.

[24] K. Stanley, S. Parag, R. Matthew and D. Pedro, "The Alchemy System for Relational AI", University Of Washington, Seattle, **(2005)**.

[25] A. De Roo, H. Sözer and M. Akşit, "Verification and Analysis of Domain-Specific Models of Physical Characteristics in Embedded Control Software", Information and Software Technology, vol. 54, no. 12, **(2012)**, pp. 1432-1453.

[26] N. Hoan Anh, N. Tung Thanh and N. H. Pham, "Clone Management for Evolving Software", IEEE Transactions on Software Engineering, vol. 38, no. 5, **(2012)**, pp. 1008-1026.

[27] M. Staron, "Critical Role of Measures in Decision Processes: Managerial And Technical Measures in the Context of Large Software Development Organizations", Information And Software Technology, vol. 54, no. 8, **(2012)**, pp. 887-899.

[28] S. Adolph, P. Kruchten and W. Hall, "Reconciling Perspectives: A Grounded Theory Of How People Manage The Process Of Software Development", Journal Of Systems And Software, vol. 85, no. 6, **(2012)**, pp. 1269-1286.

[29] N. Shahmehri, A. Mammar and E. Montes De Oca, "An Advanced Approach for Modeling and Detecting Software Vulnerabilities", Information and Software Technology, vol. 54, no. 9, **(2012)**, pp. 997-1013.

Author

Zhang Tienan was born in 1973. He received his master degree in Xiangtan University at 2008. Current he is a lecturer of Hunan Institute of Engineering. His research directions include software engineering, distributed computing, network integration.