Inductive Verification and Validation of Multi-Agent Systems

Nico Jacobs, Kurt Driessens, Luc De Raedt,
Dept. of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{nico, kurtd, lucdr}@cs.kuleuven.ac.be

Abstract

Most verification and validation (V&V) methods employ deductive reasoning to verify whether an implementation is in agreement with a specification. We present a novel approach to verification and validation of complex systems that is based on inductive reasoning. Induction allows to derive general rules from specific behaviours of the software (e.g. the inputs and outputs). Using inductive logic programming, partial *declarative* specifications of the software can be induced. These rules can be readily interpreted by the designers or users of the software, and can in turn result in changes to the software. The approach outlined was tested in the domain of multi-agent systems, more in particular, the RoboCup domain.

1 Introduction

Most approaches to V&V of knowledge based systems employ deductive reasoning techniques in order to check whether a knowledge based system is conform with some (possibly partial) specifications. There is however some inherent limitation of the deductive approach. Indeed, the deductive approach relies on the ability to specify the intended behaviour of the knowledge base. For complex systems, such as multi-agent systems operating in complex environments, this is a very hard task; one might even argue that for certain applications (such as RoboCup) designing the specifications of the overall multi-agent system is almost as hard as implementing the individual agents.

Inductive V&V methods take a different approach. Rather than starting from the specification and testing whether it is consistent with an implementation, inductive reasoning methods start from an implementation, or more precisely, from examples of the behaviour of the implementation, and produce a (partial) specification. Provided that the specification is declarative, it can be interpreted by the human expert. This machine generated specification is likely to give the expert new insights into the behaviour of the complex system he wants to verify. If the induced behaviour is conform with the expert's wishes, this will (partly) validate the system. Otherwise, if the expert is not satisfied with the induced specification, he or she may want to modify the knowledge based system and repeat the verification or validation process.

This paper addresses the use of inductive reasoning for knowledge base V&V. The employed techniques come from the domain of inductive logic programming, as these methods produce declarative specifications in the form of logic programs. Furthermore, the sketched techniques are tested in a multi-agent setting, i.e. that of RoboCup. In such complex multi-agent systems, it is very hard to see the impact of changes in one agent on the overall behaviour of the environment.

This paper is organised as follows: in section 2, we introduce inductive learning through inductive logic programming; in section 4 we show how this technique can be used in verification; in section 5 we give an example of a multi-agent system and how we used our technique to verify this, after which we conclude. For the remainder of this article we assume the reader is familiar with Prolog.

2 Inductive Logic Programming

Inductive logic programming [14] lies at the intersection of machine learning and computational logic. It combines inductive machine learning with the representations of computational logic. Computational logic (a subset of first order logic) is a more powerful representation language than the classical attribute-value representation used in machine learning. This representational power is necessary for V&V of contemporary knowledge based systems, because such knowledge based systems are in turn written in expressive programming languages or expert system shells. Another advantage of inductive logic programming is that it enables the use of background knowledge (in the form of Prolog programs) in the induction process.

An ILP system takes as input examples and background knowledge and produces hypotheses as output. There are two common used ILP settings which differ in the representation of these data: learning from entailment ([8] compares different settings) and learning from interpretation [11]. In this paper we will use the second setting. In learning from interpretations, an example or observation can be viewed as a small relational database, consisting of a number of facts that describe the specific properties of the example. In the rest of the paper, we will refer to such an example as a model. Such a model may contain multiple facts about multiple relations. This contrasts with the attribute value representations where an example always corresponds to a single tuple for a single relation.

The background knowledge takes the form of a Prolog program. Using this Prolog program, it is possible to derive addition properties (through the use of Prolog queries) about the examples. If for instance we are working in a domain where family-data is processed, possible background knowledge would be:

parent(X,Y) ← mother(X,Y).
parent(X,Y) ← father(X,Y).
grandmother(X,Y) ← mother(X,Z), parent(Z,Y).

There are also two forms of induction considered here: predictive and descriptive induction. Predictive induction starts from a set of classified examples, a background theory, and the aim is to induce a theory that will classify all the examples in the appropriate class. On the other hand, descriptive induction starts from a set of unclassified examples, and aims at finding a set of regularities that hold for the examples. In this paper, we will use the Tilde system [3] for predictive induction, and the Claudien system [10] for descriptive induction.

Tilde induces logical decision trees from classified examples and background theory. For example consider this background knowledge:

```
replaceable(gear).  replaceable(wheel).  replaceable(chain).
not_replaceable(engine).  not_replaceable(control_unit).
```

and a number of models describing worn parts and the resulting action:

```
begin(model(1)).    begin(model(2)).     begin(model(3)).   ...
sendback.           fix.                 keep.
worn(gear).         worn(gear).          end(model(3)).
worn(engine).       end(model(2)).
end(model(1)).
```

Tilde will return this classification tree (there were 15 models in this example):

```
worn(A) ?
+--yes: not_replaceable(A) ?
|       +--yes: sendback
|       +--no:  fix
+--no:  keep
```

Claudien induces clausal regularities from examples and background theory. E.g. consider the single example consisting of the following facts and empty background theory :

```
human(an). human(paul). female(an). male(paul).
```

Induced theory is :

human(X) $\leftarrow$ female(X).
human(X) $\leftarrow$ male(X).
false $\leftarrow$ male(X) $\wedge$ female(X).
male(X) $\vee$ female(X) $\leftarrow$ human(X).

Notice that this very simple example shows the power of inductive reasoning. From a set of specific facts, a general theory containing variables is induced. It is not the case that the induced theory deductively follows from the given examples.
Details of the Tilde and Claudien system can be found in [3] [10].

3 ILP for Verification and Validation

Given an inductive logic programming system, one can now verify or validate a knowledge based or multi-agent system as follows. One starts constructing examples (and possibly background knowledge) of the behaviour of the system to be verified. E.g. in a knowledge based system for diagnosis, one could start by generating examples of the inputs (symptoms) and outputs (diagnosis) of the system. Alternatively, in a multi-agent system one could take a snapshot of the environment at various points in time. These snapshots could then be checked individually and also the relation between the state an agent is in and the action he takes could be investigated.

Once examples and background knowledge are available one must then formulate verification or validation as a predictive or descriptive inductive task. E.g. in the multi-agent system, if the aim is to verify the properties of the states of the overall system, this can be formulated as a descriptive learning task. One then starts from examples and induces their properties. On the other hand, if the aim is to learn the relation among the states and the actions of the agent, a predictive approach can be taken.

After the formulation of the problem, it is time to run the inductive logic programming engines. The results of the induction process can then be interpreted by the human verifiers or validators. If the results are in agreement with the wishes of the human experts, the knowledge based or multi-agent system can be considered (partly) verified or validated. Otherwise, the human expert will get insight into the situations where his expectations differ from the actual behaviour of the system. In such cases, revision is necessary. Revision may be carried out manually or it could also be carried out automatically using knowledge revision systems (see e.g. Craw's Krust system [6], or De Raedt's Clint [7]). After revision, the validation and verification process can be repeated and this until the human expert is satisfied by the results of the induction engines.

4 Multi-agent systems

4.1 Complexity of Multi Agent Systems

Recently a lot of attention has been devoted to intelligent agents and a number of promising agent applications exists in domains such as information filtering and electronic commerce (see for instance [1] for an overview of the current state of the art in agent applications). Although the concept of an intelligent agent is not yet clearly defined there are diverse aspects reoccuring in numerous agents. We briefly mention some of these aspects and study their consequences for the V&V process.

A number of 'agent features' such as reactiveness (react to a changing environment) and pro-activeness (take the initiative to perform some action) will probably become standard practice in future software agent development. These features are intended to result in a more robust software product but at the same time make it more complex to implement such an agent. As a result of reactiveness an agent can change it's behaviour based on observations. Some agents even learn and incrementally adapt their behaviour. Pro-activeness has as a consequence that the functional view of software (transform a certain input into a desired output) does not hold for agents. Agents can act even if there was no (explicit) input; agents can act 'spontaneously'.

An other aspect in which agents differ from regular software applications is that they often have incomplete and/or incorrect information concerning the environment in which they operate. This is due to the fact that agents most of the time operate in complex and dynamic environments like the Internet or environments with real world interaction. Because we are never certain of the exact result of an action, we can never get an 100% accurate system. With agents, one should cope with systems that perform well most of the time.

Finally agents also act and by this influence their environment. Take for instance an agent who schedules the factory work flow. If at a certain point the agent notices that certain decisions were bad it can't roll back the environment; it has to live up to the effects of the actions it did or proposed. An agent continuously produces possible side-effects which irreversibly change the agent's environment.

A number of agent applications is based on multiple agents cooperating to achieve a common goal. In this case, the changes made to the environment are not a result of the behaviour of a single agent, but of the interaction of the agents with each other and the world they act upon. As a consequence, the emergent behaviour of the system is hard to understand. Furthermore, the system is very hard to specify (or verify) at the overall level. The global behaviour is not the sum of the local behaviours of the agents. Also, agent implementations are designed and implemented at this local level. It is then hoped that this will work as desired at the global level. This is e.g. the case in RoboCup. Further complications arise because the agent designer may not exercise total control over the environment, e.g. in Robocup one only controls one team. Because of this situation, a global V & V approach can benefit from using induction.

5 Experiments in RoboCup

5.1 RoboCup

We now perform experiments with the sketched approach to V & V in order to demonstrate the promise of the technique. As a test-bed we choose RoboCup [13], which is a standard problem for many fields of AI and Robotics research of which multi agent systems is only one. The challenge consists of various competitions such as the real robot league, the simulator league and the special skills competition. Most of the multi agent research is done in the simulator league.

In the simulator league, the object of the challenge is to build eleven software agents that together form a soccer playing team. The agents' environment is defined by the soccer-server supplied by the RoboCup competition. It creates the world in which the agents act and artificially supplies the researchers with real world problems such as uncertainties, noise, an obstructive environment — consisting of the simulated field and the opposing team —, limited communication and a high degree of dynamics, all in a controlled environment. The various uncertainties — e.g. a well kicked ball can still not be guaranteed to reach it's destiny, but can be intercepted by another player or simply deviate from it's course — result in a complex environment with complex, hard to verify behaviour of the agents. This forms an ideal test-bed for our ideas.

For the experiments, a reactive agent was used. Each time new sensory information was processed, the agent evaluated the believed state of the world and returned the appropriate action. There was no internal reference to prior actions of the agent or to long term plans. Two threads were used in the agent. One to process the sensory information and one to analyse the believed information and select the appropriate actions.

5.2 Modelling the Information

The first tests were run to study the behaviour of a single agent. We supplied the agents with the possibility to log their actions and the momentary state of the world as perceived by them. This way we were able to study the behaviour of the agents starting from their beliefs about the world. Because all the agents of one team were identical except for their location on the playing field, the log files were joined to form the knowledge base used in the experiments. A sample description of one state from the log-files looks as follows :

```
begin(model(e647)).
   player(my,1,-43.91466,5.173167,3352).
   player(my,2,-30.020395,7.7821097,3352).
   ...
   player(other,10,14.235199,15.192206,2748).
   player(other,11,0.0,0.0,0).
   ball(-33.730022,10.014952,3352).
   mynumber(5).
   bucket(1).
   rctime(3352).
   moveto(-33.730022,10.014952).
   actiontime(3352).
end(model(e647)).
```

The different predicates have the following meaning :

$player(T, N, X, Y, C)$ the agent has last seen the player with number N from team T at location (X, Y) at time C.

$ball(X, Y, C)$ the agent has last seen the ball at location (X, Y) at time C.

$mynumber(N)$ this state was written by the agent with number N. It thus corresponds to the observation of agent N.

$bucket(N)$ the bucket used for bringing the agent back to his home position. The bucket-value was lowered every input/output cycle and forced the agent to his home-location and reset when it reached zero.

$rctime(C)$ the time the state was written.

$actiontime(C)$ the time the action listed was choosen.

$moveto(X, Y), shoottogoal, passto(X, Y), turn(X), none$ the action the agent performed.

The $rctime(C)$ predicate was used to judge the age of the information in the model as well as to be able to decide how recent the action mentioned in the model or the information about the other players or the ball is. This was done by comparing the $rctime(C)$ with $actiontime(C)$ or the time specified in the $ball/3$ and $player/3$ predicates. Logging was done at regular time intervals instead of each time an action was performed, to be able to use the inactivity of the agent, or his passed actions as information also.

Some of the actions that were used while logging were already higher level actions than the ones that are sent to the soccer-server. However these actions, such as *shoottogoal* for example, were trivial to implement.

To make the results of the tests easier to interpret an even higher level of abstraction was introduced in the background knowledge used during the experiments. Actions that are known to have a special meaning were renamed. For instance a soccer-player that was looking for the ball always used the $turn(85)$ command, so this command was renamed to *search_ball*. An other example of information defined in the background knowledge is the following rule :

```
action(movetoball):- validtime, moveto(X1,Y1), ball(X2,Y2),
                      distance(X1,Y1,X2,Y2,Dist), Dist =< 5 .
```

in which the *moveto*(X, Y) command was merged with other information in the model to give it more meaning. For instance, *moveto*$(-33.730022, 10.014952)$ and *ball*$(-33.730022, 10.014952, 3352)$ in the model shown above, would be merged into *movetoball* by this rule. Often a little deviation was permitted to take the dynamics and noise of the environment into account. The actions used to classify the behaviour of the agent were : *search_ball*, *watch_ball*, *moveto*, *movetoball*, *moveback*, *shoottogoal*, *passto*, *passtobuddy* and *none*. Some of these actions were not used in the implementation of the agent but were included anyway for verification purposes. For instance, although — according to specifications — an agent should always "move to the ball" or "move back", the possible classification "moveto" was included in the experiments anyway.

Other background knowledge included the playing area's of the soccer-agents and other high level predicates such as *haveball*, *ball_near_othergoal*, *ball_in_penaltyarea*, etc. Again, not all of these concepts were used when implementing the agent. This illustrates the power of using background knowledge. Using background knowledge, it is possible for the verifier to focus on high-level features instead of low-level ones.

5.3 Verifying Single Agents

The first tests were performed with Tilde, which allowed the behaviour of the agent to be classified by the different actions of the agent.

The knowledge base used was the union of the eleven log-files of the agents of an entire team. The agents used in the team all had the same behaviour, except for the area of the playing field. The area the agent acted in depended on the number of the agent and also was specified in the used background knowledge. The resulting knowledge base consisted of about 17000 models, collected during one test-game of ten minutes.

The first run of Tilde resulted in the following decision tree

```
seeball ?
+--yes: ball_in_my_area ?
|       +--yes: haveball ?
|       |       +--yes: ball_near_othergoal ?
|       |       |       +--yes: action(shoottogoal) [15 / 15]
|       |       |       +--no:  action(passtobuddy) [122 / 124]
|       |       +--no:  action(movetoball) [1007 / 1015]
|       +--no:  bucket_was_empty ?
|               +--yes: action(moveback) [342 / 347]
|               +--no:  action(watch_ball) [2541 / 3460]
+--no:  action(search_ball) [7770 / 7771]
```

Most of the classifications made by Tilde were very accurate for the domain. However, the prediction of the *action*(*watch_ball*) only reached an accuracy of 73,4%.

To get a better view on the behaviour of the agent in the given circumstances Claudien was used to describe the behaviour of the agent in case "*seeball*, *not*(*ball_in_my_area*), *not*(*bucket_was_empty*)."

Claudien found two rules that describe these circumstances. The first rule was the one Tilde used to predict the *watch_ball* action.

```
action(watch_ball) if  not(action(none)), seeball,
                       not(ball_in_my_area), not(bucket_was_empty).
```

Claudien discovered the rule had an accuracy of 73%.

The other rule that was found by Claudien was the following :

```
action(moveback) if  not(action(none)), seeball ,
                     not(ball_in_my_area), not(bucket_was_empty).
```

which reached an accuracy of 26%. It states that the agent would move back to his home location at times he was not supposed to. Being forced to go back to its home-location every time the bucket was emptied, this behaviour was a result of the bucket getting empty while the player was involved in the game and therefore not paying immediate attention to the contents of the bucket.

To gain more consistency in the agents behaviour, the bucket mechanism was removed and replaced by a new behaviour where the agent would move back when it noticed that it was to far from its home location. The new behaviour, after being logged and used in a Tilde-run resulted in the following tree :

```
seeball ?
+--yes: ball_in_my_area ?
|       +--yes: haveball ?
|       |       +--yes: ball_near_othergoal ?
|       |       |       +--yes: action(shoottogoal) [48 / 48]
|       |       |       +--no:  action(passtobuddy) [85 / 85]
|       |       +--no:  action(movetoball) [796 / 810]
|       +--no:  at_place ?
|               +--yes: action(watch_ball) [3826 / 3840]
|               +--no:  action(moveback) [384 / 394]
+--no:  action(search_ball) [7180 / 7318]
```

in which the *action*(*watch_ball*) was predicted with an accuracy of 99,6 %. The increase in consistency in the behaviour in the agent, improved its performance in the RoboCup environment. Because the agent did not move back to his home-location at moments it wasn't necessary he could spend more time tracking the movement of the ball and fellow agents.

5.4 Verifying Multiple Agents

In agent applications it is often important that not only all agents individually work properly, the agents also have to cooperate correctly. One important point in this is to check if the beliefs of the different agents more or less match. In the case of our RoboCup agents we want to know for instance if there is much difference between the position where player A sees player B and the position where player B thinks it is[1]. So first we used Claudien to find out how often agents have different believes about there positions, and how much their beliefs differ.

To do these tests, we transformed the data-file so that one model contains the believes of multiple agents at the same moment in time. To accomlish this, an extra argument was added to the predicates which indicated the ownership of the belief. For instance, in the *vplayer*/5 predicate, the first argument refered to the owner of the belief. Claudien found multiple rules like the one below:

[1]it is impossible to know what the real position of a player is, so we can only compare the different believes the agents have

```
Dist < 2 if mynumber(A,Nr), vplayer(A,my,Nr,X1,Y1),
            vplayer(B,my,Nr,X2,Y2), mynumber(B,Nr2),
            vplayer(B,my,Nr2,X3,Y3), not(A=B),
            distance(X1,Y1,X2,Y2,Dist),
            distance(X2,Y2,X3,Y3,Dist2), Dist2 < 10.
```

This rule, which has an accuracy of 78% states that if two players are less then 10 units apart, the difference in the believes of the position of one of those two players is less then 2 units. From all the rules we could conclude useful information, for instance, we found out that our agents can best estimate a team-mate's position from distance 10. All rules found were 'acceptable' rules (e.g. for distances larger than 10, the error is positively correlated with the distance between the players), so we can conclude from the observed behaviour that the beliefs of the different agents do not differ much.

6 Related work

This work builds upon earlier ideas on combining V&V with inductive logic programming [9]. It is also related to other approaches applying machine learning with validation and verification. This includes the work of Susan Craw on her KRUST system for knowledge refinement [6], the work by Bergadano et al. [2] and the work by De Raedt et al. [12]. The approach taken in KRUST is complementary to ours. Rather than starting from examples of the actual behaviour of the system, KRUST starts from examples of the desired behaviour of the system. Whenever the two behaviours do not match, KRUST will automatically revise the knowledge based system. It is clear that the KRUST approach could also be applied within our methodology, at the point where the human discovers inconsistencies between the two behaviours. If the human then specifies examples of the intended behaviour, KRUST might help revising the original knowledge base. The approach of Bergadano et. al. and De Raedt et. al. uses inductive machine learning to automatically and systematically generate a test set of examples that can be used for verification or validation. Finally, our work is also related to the work by William Cohen [5] on recovering software specifications from examples of the input-output behaviour of the program.

7 Conclusions

We briefly sketched a novel approach to V&V that is based on inductive reasoning rather than deductive reasoning. We also reported a number of experiments in the domain of multi-agent systems (RoboCup) which prove the concept of the approach.

Further work on this topic could involve applying the verification and validation technique also to other multi-agent systems (such as e.g. DESIRE [4]), and also to extend the inductive method to other representations. For instance, it seems very well possible to apply inductive techniques in order to automatically construct decision tables starting from the knowledge base. Such decision tables are already popular in V&V, but they are typically made by the human expert (in collaboration with the machine), see e.g. [15].

Acknowledgements: the authors wish to thank Hendrik Blockeel and Luc Dehaspe for their help with the Tilde and Claudien system. Nico Jacobs is financed by a specialisation grant of the Flemish Institute for the promotion of scientific and technological research in the industry (IWT). Luc De Raedt is supported by the Fund for scientific research, Flanders. This work is supported by the European Community Esprit project no. 20237 (ILP 2).

References

[1] *Proceedings of the Second International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, 1997.

[2] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Transactions on Software Engineering and Methodology*, 5(2):119–145, 1996.

[3] H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.

[4] F. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Desire: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6:67–94, 1997. Special Issue on Formal Methods in Cooperative, Information Systems.

[5] W. Cohen. Recovering Software Specifications with ILP. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 142–148, 1994.

[6] S. Craw and D. Sleeman. Knowledge-based refinement of knowledge based systems. Technical Report 95/2, The Robert Gordon University, Aberdeen, UK, 1995.

[7] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[8] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.

[9] L. De Raedt. Using ILP for verification, validation and testing of knowledge based systems, 1997. invited talk at EUROVAV 1997.

[10] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

[11] L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[12] L. De Raedt, G. Sablon, and M. Bruynooghe. Using interactive concept learning for knowledge-base validation and verification. In *Validation, Verification and Test of Knowledge-based Systems*, pages 177–190, 1991.

[13] H. Kitano, M. Veloso, H. Matsubara, M. Tambe, S. Coradeschi, I. Noda, P. Stone, E. Osawa, and M. Asada. The robocup synthetic agent challenge 97. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 24–29. Morgan Kaufmann, 1997.

[14] S. Muggleton and C. D. Page. A learnability model for universal representations. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, pages 139–160, Sankt Augustin, Germany, 1994. GMD.

[15] J. Vanthienen, C. Mues, and C. Wets. Inter-tabular verification in an interactive environment. In *Proceedings of the '97 European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-97)*, pages 155–165, 1997.