Learning Goal Hierarchies from Structured Observations and Expert Annotations

Tolga Könik and John Laird

Artificial Intelligence Lab., University of Michigan
1101 Beal Avenue, MI 48109, USA
{konik, laird}@umich.edu

Abstract. We describe a framework for generating agent programs that model expert task performance in complex dynamic domains, using expert behavior observations and goal annotations as the primary source. We map the problem of learning an agent program on to multiple learning problems that can be represented in a "supervised concept learning" setting. The acquired procedural knowledge is partitioned into a hierarchy of goals and it is represented with first order rules. Using an inductive logic programming (ILP) learning component allows us to use structured goal annotations, structured background knowledge and structured behavior observations. We have developed an efficient mechanism for storing and retrieving structured behavior data. We have tested our system using artificially created examples and behavior observation traces generated by AI agents. We evaluate the learned rules by comparing them to hand-coded rules.

1 Introduction

Developing autonomous agents that behave "intelligently" in complex environments (i.e. large, dynamic, nondeterministic, and with unobservable states) usually presumes costly agent-programmer effort of acquiring knowledge from experts and encoding it into an executable representation. Machine learning can help automate this process. In this paper, we present a framework for automatically creating an agent program using the data obtained by observing experts performing tasks as the primary input. The ultimate goal of this line of research is to reduce the cost and expertise required to build artificial agents.

Learning from expert observations to replicate behavior is often called *behavioral cloning*. Most behavioral cloning research to date has focused on learning sub-cognitive skills in controlling a dynamic system such as pole balancing [10], controlling a simulated aircraft [11, 16], or operating a crane [20]. In contrast, our focus is capturing deliberate high-level reasoning.

Behavioral cloning was originally formulated as a direct mapping from states to control actions, which produces a reactive agent. Later, using goals was proposed to improved robustness of the learned agents. Camacho's system [2] induced controllers in terms goal parameter so that the execution system can use the same controllers under varying initial conditions and goal settings. It did not however learn how to set the goal parameters. Isaac and Sammut [5] present a two step approach where first a

R. Camacho, R. King, A. Srinivasan (Eds.): ILP 2004, LNAI 3194, pp. 198–215, 2004.

mapping from states to goal parameters is learned, then control actions are learned in terms of these goals. Suc and Bratko [19] describe induction of qualitative constraints that model trajectories the expert is trying to follow to achieve goals. These constraints are used in choosing control actions.

In the goal-directed behavioral cloning research mentioned above, goals are predefined parameters of a dynamic system. For example, the learning-to-fly domain has goal parameters such as target turn-rate. In contrast, we want to capture a hierarchy of high-level goals. For example in a building navigation domain, an expert may have a goal of choosing which door it should go through to get to a room containing a particular item. Unlike the above approaches, we don't assume that the system has pre-existing definitions for the goals. Instead in our framework, the meaning of goals are implicitly learned by learning when the experts select them together with the decisions that become relevant once the goals are selected. To facilitate this, we require that the experts annotate the observation traces with the names and parameters of goals (i.e. *select-door*(d_1)). This requirement is feasible in our setting because high-level goals typically change infrequently and the experts are likely to be conscious of them.

Our work is strongly influenced by van Lent's [22] learning by observation framework. His system, KnoMic, also represents and learns an explicit hierarchy of high-level goals. KnoMic uses an attribute-value based representation that would run into difficulties when structured properties of the environment are relevant, for example if there are multiple objects of the same kind (i.e. two enemy planes in a tactical air combat domain), structured domain knowledge (i.e. a building map in a navigation domain), or inferred knowledge (i.e. shortest-path towards a target room) is essential in choosing and executing the right strategy.

Our framework proposes a natural solution for the above limitations by framing the learning problem in the first order setting of Inductive Logic Programming (ILP) while maintaining most of the core features of KnoMic. Unlike KnoMic, our framework allows parametric and structured goal annotations, structured background knowledge, and structured sensors. In addition, KnoMic uses a simple single-pass learning algorithm that cannot deal with noise and assumes that the expert exhibits correct and consistent behavior at all times, while our framework uses an ILP algorithm that is robust in the presence of noise. To be able to use ILP algorithms in domains with large numbers of facts, we have developed an efficient mechanism to store and access structured behavior data.

We use the general agent architecture Soar [8] as the target language for the acquired knowledge. Soar uses a symbolic rule based representation that simplifies the interaction with the ILP learning component. Although Soar influences how knowledge is represented in our framework, we introduce the framework independent of Soar to make our learning assumptions more explicit and to have results that are transferable to other architectures.

The paper is organized as follows. Next, we describe our learning by observation framework. In section 3, we present experimental results. In section 4, we discuss related work. Finally, we conclude with remarks about future directions in section 5.

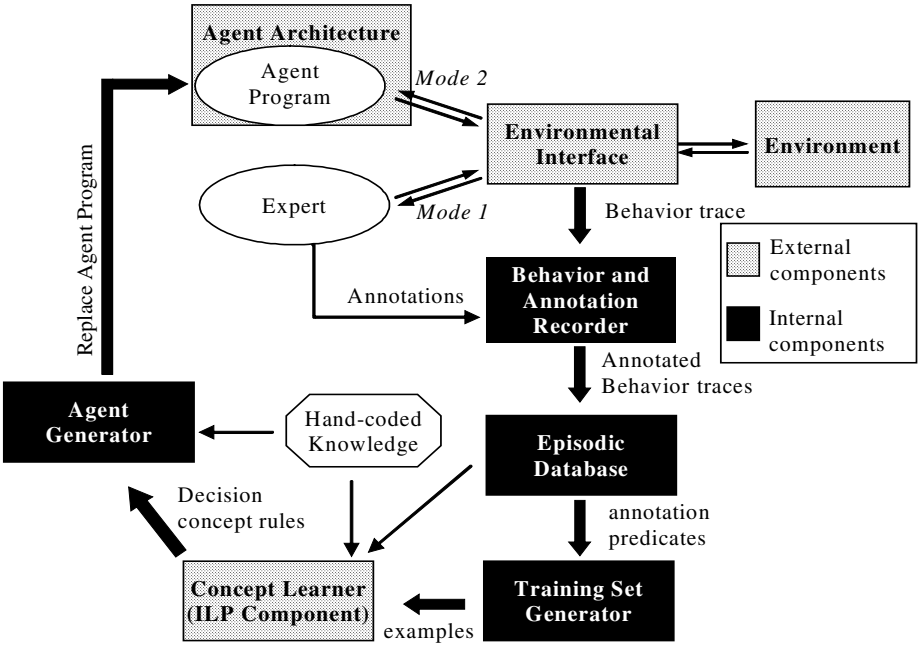

Fig. 1. General overview of our learning by observation framework. In mode 1, the expert generates annotated behavior. In mode 2, an agent program executes behavior and the expert accepts or rejects its annotations

2 Our Learning by Observation Framework

The execution cycle of our framework has two operation modes (Fig. 1). In the first mode, the expert interacts with the environment. In the second mode, the approximately correct agent created during previous learning interacts with the environment. Both of these interactions are recorded to a *behavior trace* structure. In the first mode, the expert annotates the behavior trace with the goals he/she has been pursuing. In the second mode, the agent proposes similar annotations and the expert accepts or rejects them. In both modes, the *annotated behavior traces* are inserted into an *episodic database* that efficiently stores and retrieves the observed situations and the expert annotations. The training set generator component maps the problem of "obtaining an agent program" to multiple problems that can be represented in a "supervised concept-learning" setting. These *decision concepts* are used in the generated agent program. For each decision concept, the training set generator returns positive and negative examples, using the information stored in the episodic database. The concept learner component uses an ILP algorithm that learns rules representing the decision concepts, using the examples in the training set and background knowledge obtained by accessing the episodic database and hand-coded domain theory. The agent generator component converts the decision concept rules to an executable agent program. At each cycle, a new agent program is learned from scratch

but since more behavior traces are accumulated, a more accurate agent program is expected to be learned, which can in turn generate new traces when the learned agent program interacts with the environment in the second execution mode. At any time during the agent performance (mode 2), the expert can intervene and take control (mode 1) to generate traces, for example if the agent is doing very poorly at a goal. This may help the learning focus on parts of the task where the agent program is lacking knowledge most.

We have partially implemented this framework to conduct the experiments reported in section 3.2. Our program works in the first mode of the execution cycle, and instead of human expert generated behavior, we use behavior of hand-coded Soar agents. At this stage of the research, cloning artificial agents is a cost-effective way to evaluate our framework - it greatly simplifies data collection and it does not require us to build domain specific components to track expert behavior and annotations. Instead, we built a general interface that can extract annotations and behavior from Soar agents on any environment Soar has been connected to.

2.1 Target Agent Architecture and Environments

We use Soar [8] as our target architecture. A long-term motivation is that Soar is one of the few candidates of unified cognitive architectures [14] and has been successful as the basis for developing knowledge-rich agents for complex environments [6, 9, 24] One practical reason for this choice is that there exist interfaces between Soar and these environments that can be reused in our system. Moreover, the hand-coded agents required significant human effort and they can form a basis of comparison for the agents we create automatically.

In this paper we will use examples from “Haunt 2 game” [9], which is a 3-D first person perspective adventure game built using the Unreal game engine. This environment has a large, structured state space, real time decisions, continues space, external agents and events.

2.2 Representation of the Environment and Task Performance Knowledge

In complex domains, an agent (expert/agent program) may receive vast amounts of raw sensory data and the low level motor interaction the agent has to control may be extremely complicated. Since we focus more on higher level reasoning of a cognitive agent than low-level control, we assume that the agents interact with the environment using an interface that converts the raw data to a *symbolic environmental representation* (*SER*). While the expert makes his decisions using a visualization of the raw data, the agent program will make decisions with corresponding symbolic data. Moreover, both the expert and the agent program execute only symbolic actions provided by SER, which is responsible for implementing these actions in the environment at the control level.

At any given moment, SER maintains a set of facts that symbolically represent the state of the environment as perceived from the expert’s perspective. Soar agents represent their beliefs about the external world and internal state using a directed graph of binary predicates. Adapting that style, we will assume that the environment representation maintained by SER contains predicates of the form $p(a, b)$ where p is a

relation between the objects in the environment denoted by *a* and *b* in SER. In the Haunt domain, a "snapshot" of this time varying representation may be as depicted in Fig. 2. The sensors are represented with a binary predicate where the first argument is a special symbol (i.e. agent) and the second argument is the sensed value. The sensors can be *constant-valued* such as the x-coordinate(agent, 35) or energy-level(agent, high) as well as *object-valued* such as current-room(agent, r_1). The object valued sensors can be used to represent structured relations among perceived objects. For example, when a book on top of a desk enters the visual display of the expert, it is SER's responsibility to build corresponding relations and to bind the sensors to these relations. SER also has the responsibility of associating the directly sensed features of the environment with the hand-coded factual knowledge. For example in Fig. 2, we not only see that the expert is in the room r_1, but we also know that he/she can go towards a room r_3 by following a path that goes through door d_1. During the learning phase both the observed dynamical features and the hand-coded factual knowledge are used in a uniform way.

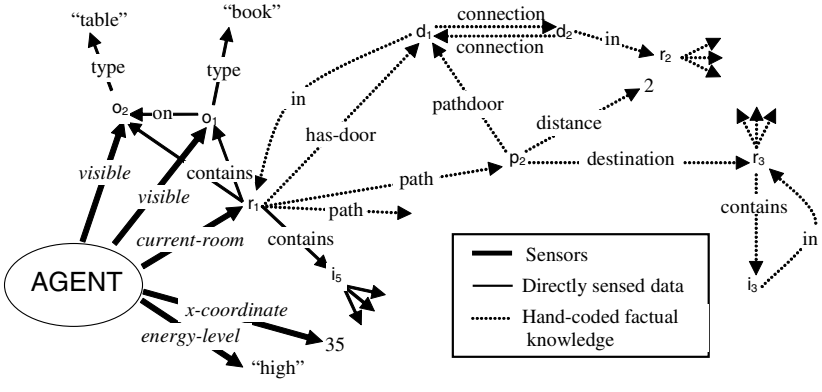

Fig. 2. A snapshot of the data maintained in the symbolic environmental representation (SER) in Haunt domain. SER dynamically updates directly sensed relations and associates factual background knowledge with the sensed objects

We assume that the performance knowledge of the target agent program is decomposed into a hierarchy of operators that represent the goals that the agents pursue and the actions that they take to achieve their goals (Fig. 3). With this assumption, we decompose the "learning an agent program" problem to multiple "learning to maintain an operator" problems. The suboperators correspond to strategies that the agent can use as part of achieving the goal of the parent operator. The agent has to continuously maintain the activity of these operators based on current sensors and internal knowledge. When the agent selects an operator, it must also instantiate the parameters. It then executes the operator by selecting and executing suboperators. The real execution on the environment occurs when *actions*, the lowest level operators, are selected. The names of the selected actions and their parameters are sent to the SER, which applies them in the environment. The actions are continuously applied on the environment as long as the agent keeps them active. We assume that there may be at most one operator active at each level of the

hierarchy. This simplifies the learning task because the learner associates the observed behavior only with the active operators and each operator is learned in the context of a single parent operator.

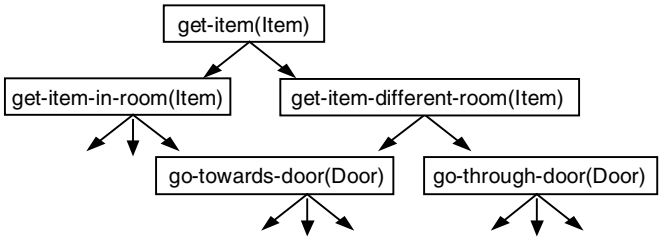

Fig. 3. An Operator Hierarchy in a Building Navigation Domain

For example, if an agent decides to get an item i_1 in a different room by selecting the operator get-item-different-room(Item) with the instantiation Item=i_1, to achieve the task, it could select the suboperator go-towards-door(Door), where Door should be instantiated with the door object on the shortest path from current room to the room where i_1 is in. The real execution occurs with primitive SER actions such as go(forward) or turn(left).

In this representation, information about how the operators are selected implies information about how the operators are executed because execution of an operator at one level is realized by selection of the suboperators at the lower level. Among other more complex possibilities, suboperators may represent alternative or sequential ways of reaching a goal, depending on the learned knowledge of how to maintain the activity of the operators. For example in Fig. 3, get-item-different-room and get-item-in-room are two alternative strategies that may be used to reach the parent goal get-item. Which one of them is preferred depends upon whether the target item is in the current room or not. On the other hand, the operators go-towards-door and go-through-door should be executed sequentially in a loop to achieve their high-level goal. Each time the agent enters a room that does not contain the target item, the agent selects a door and moves towards it (go-towards-door), until the agent is close enough to go through the door to enter a new room (go-through-door). If the item is not in the new room, the agent reselects a door and goes towards it (go-towards-door). If the agent enters a room containing the item, the operator get-item-different-room is immediately retracted with all of its suboperators and get-item-in-room is selected.

The initial knowledge that the system has about the operators consists of their names, the hierarchical relation among them and the scope of their parameters. The final agent obtained as the result of learning should have the capability of maintaining the activity of the operators (i.e. selecting them with correct parameters, stopping them when they achieve their goal, abandoning them in preference of other operators, etc.) and executing them (managing the suboperators).

2.3 Behavior and Annotation Recorder

While the expert or the agent program is performing a task, symbolic state of the environment is recorded into a structure called a *behavior trace*. The symbolic

representation that the SER maintains is sampled in small intervals, at consecutively enumerated time points s_i called *situations*. We assume that the domain dependent sampling frequency is sufficiently high so that no significant changes occur between two consecutive situations. We say that the *observed situation predicate* $p(s_i, a, b)$ holds if and only if $p(a, b)$ was in SER at the situation s_i.

If the environment contains static facts (i.e. rooms, doors, etc...) that do not change over different situations, that information can be added to the beginning of the behavior trace manually, even if the expert does not perceive them directly. This corresponds to the assumption that the expert already knows about these features and the learning system will use this information as background knowledge as it creates the model of the expert. If $p(x, y)$ is such a static fact, we say that the *assumed situation predicate* $p(s_i, x, y)$ is true for any s_i.

In the first execution mode, the expert annotates the situations in his/her behavior with the names of the operators and parameters that he/she selects from the operator hierarchy (i.e. Fig. 3). A valid selection that satisfies the semantics of the operator hierarchy must form a connected path of operators starting from the root of the operator hierarchy. Since the actions are executed using SER directly, action annotations can be recorded automatically without any expert effort. In the second execution mode, the expert inspects the annotated behavior traces proposed by the agent program and verifies or rejects the annotations.

We assume that the expert annotates a set of consecutive situations at a time. For a set of consecutive situations R and an operator $op(x)$ where x is an instantiated operator parameter vector, if the expert annotates the situations in R with $op(x)$, we say *accepted-annotation*$(R, op(x))$ where R is called the *annotation region*. Similarly, we say *rejected-annotation*$(R, op(x))$, if the expert has rejected the agent program's annotation of R with $op(x)$.

2.4 Episodic Database

In practice, it is inefficient to store the list of all predicates that hold at each situation explicitly, especially in domains where sampling frequencies are high and there is much sensory input. The *episodic database* efficiently stores and retrieves the information contained in structured behavior traces and expert annotations. In each execution cycle, the training set generator accesses the episodic database while creating positive and negative examples of the decision concepts to be learned. Similarly, the ILP component accesses it to check whether particular situation predicates in the background knowledge hold in the behavior trace. Although the examples are generated only once for each concept, the background situation predicates may be accessed many times during learning. Typically, ILP systems consider many hypotheses before they return a final hypothesis as the result of learning and each time a different hypothesis is considered, the validity of background situation predicates that occur in the hypothesis must be tested. To make learning practical in large domains, it is crucial that the episodic database is an efficient structure.

We assume that for each situation predicate p, the arguments are classified as input or output types. Many ILP systems already require a similar specification for

background predicates.[1] The episodic database receives situation predicate queries of the form $p(s, x, y)$ where s is an instantiated situation, x is an instantiated vector of input variables, y is a vector of not instantiated output variables. The result of the query is y vectors that satisfy the query.

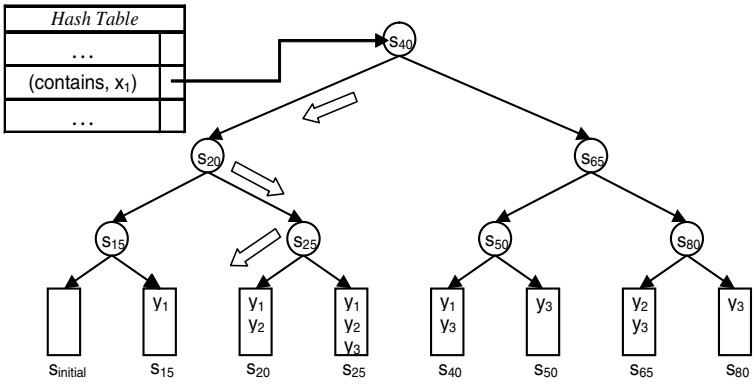

Fig. 4. Search for the query contains(s_{23}, x_1, Y) in the episodic database

In episodic database, each situation predicate is stored using multiple binary trees (Fig. 4). The leaves store the output values explicitly wherever they change and the nodes store the situations where these changes occur. More formally, for each pair (p, x), where p is a situation predicate and x is an input vector, the episodic database explicitly stores the output values Y_s, the set of all y vectors satisfying $p(s, x, y)$ for each situation s, where Y_s has changed compared to previous situation. Moreover, for each (p, x), it contains a binary search tree, where the nodes are these change situations and the leaves are the Y_s vectors. For example in Fig. 4, we have the index structure that represents the predicate contains(+Situation, +Room, -Item). This particular tree shows that room x_1 does not contain any objects in the initial situation. At situation s_{15}, the item y_1 appears in the room x_1. No changes occur, until the situation s_{20} when a new item y_2 is added to the room and so on. For example, to answer the query contains(s_{23}, x_1, Item), first the correct index tree associated with the pair (contains, x_1) is located using a hash table, then by a binary search, the last change before s_{23} is located. In this case, the last change occurs at s_{20} and Item will be instantiated with y_1 and y_2.

In our system, hand-coded static background knowledge is an important special case that is handled very easily by the episodic database. These predicates are added to the behavior trace once and then are never changed. The episodic database stores them very efficiently because their index trees will be reduced to single nodes. The expert annotation predicates are also stored in episodic database by using the operator name as input variable, and the operator arguments as output variables.

The episodic database stores the behavior traces efficiently, unless there are multi-valued predicates (multiple output instantiations at a situation) that change frequently

[1] Arguments that are declared constants are treated as input in episodic database representation.

or background predicates that have multiple mode definitions (input/output variable specifications) each requiring a separate set of index trees. In the domains we applied our system to, the first problem is negligible and the second problem does not occur.

Struyf, Ramon and Blockeel [18] describe a general formalization for compactly representing ILP background knowledge in domains that have redundancy between examples, which corresponds to consecutive situations in our case. Their system would represent our situation predicates by storing a list of predicate changes between each pair of consecutive situations. In that representation, to test a particular situation predicate, the behavior trace would have to be traced forward from the initial node, completely generating all facts in all situations until the queried situation is reached. For an ILP system that tests each rule over multiple examples, our approach would be more time efficient in domains having many facts at each situation because we don't need to generate complete states and we don't have to trace all situations. Instead, the episodic database makes binary tree searches only for the predicates that occur in the rule to be tested. In our learning by observation system, the gain from the episodic database is even more dramatic because the examples of the learned concepts are sparsely distributed over situation history.

2.5 Decision Concepts and Generating Examples

In section 2.2, we discuss how the problem of "learning an agent program" is decomposed into multiple "learning to maintain the activity of an operator" problems. In this section, we further decompose it into multiple "decision concept learning" problems that can be framed in an ILP setting.

A decision concept of an operator *op* is a mapping from the internal state and external observations of an agent to a "decision suggestion" about the activity of *op*. We currently define four decision concepts: selection-condition (when the operator should be selected if it is not currently selected), overriding-selection-condition (when the operator should be selected even if another operator is selected), maintenance-condition (what must be true for the operator to be maintained during its application), and termination-condition (when the operator has completed and should be terminated). For each decision concept, we have to define how their examples should be constructed from the observation traces and how they are used during execution. In general, for a concept of kind *con* and an operator $op(x)$, we get a decision concept $con(s, op(x))$ where s is a situation and x a parameter vector of *op*. For example selection-condition(S, go-to-door(Door)) would describe under which situation S the selection of go-to-door(Door) is advised and with what value Door should be instantiated.

The training set generator constructs the positive and negative examples of decision concepts, using the expert annotation information stored in episodic database. For a decision concept *con* and expert annotation $op(x_0)$, where x_0 is an instantiated parameter vector, a positive (negative) example is a ground term $con(s, op(x_0))$, where s is an element of a set of situations called *positive (negative) example region* of $op(x_0)$. Fig. 5 depicts the positive and negative example regions of an operator op_A, for different kind of decision concepts. The horizontal direction represents time (situations) in the behavior trace and the boxes represent the accepted annotation regions *P*, *A*, and *B* of three operators $parent(op_A)$, op_A, and op_B such that $parent(op_A)$ is the parent operator of op_A, and op_B is an arbitrary selected operator that

shares the same parent with op_A. op_B may be the same kind of operator with op_A, but it should have a different parameter instantiation. The positive example region of the selection condition of op_A is where the expert has started pursuing op_A and its negative example region is where another operator is selected (Fig. 5.b). As an example, if we have op_A = go-towards-door(d_1), A=s_{20}-s_{30}, and B=s_{50}-s_{60}, we could have the positive example selection-condition(s_{20}, go-towards-door(d_1)) and the negative example selection-condition(s_{50}, go-towards-door(d_1)).

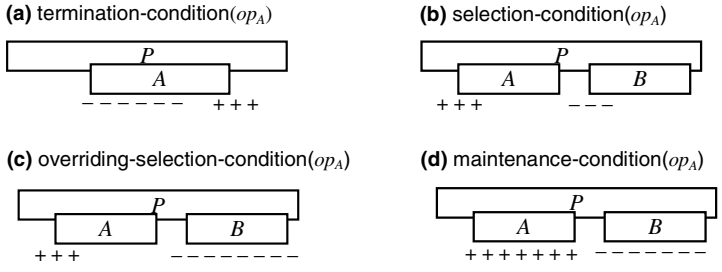

Fig. 5. The positive and negative example regions of different concepts. A, B, and P are the annotation regions of the operators op_A, op_B, and their parent operator $parent(op_A)$

In general, the examples of decision concepts of an operator op_A are selected only from situations where there is the right context to consider a decision about it. Since the operator hierarchy dictates that $parent(op_A)$ must be active at any situation where op_A is active, all decision concept examples of op_A are obtained only at situations where $parent(op_A)$ is active. Similarly during the execution, the decision concepts of op_A are considered only at situations where $parent(op_A)$ is active.

Different concepts will have different, possibly conflicting suggestions on how the operators should be selected. For example, a situation where termination-condition(op_A) holds suggests that the agent has to terminate op_A, if op_A is active and that op_A should not be selected if it not active. selection-condition(op_A) would be useful to decide whether op_A should be selected, if a previous operator op_B is already terminated (i.e. because of termination-condition(op_B)), it would not be very useful while op_B is still active because such situations are not considered as examples for selection-condition(op_A). On the other hand, overriding-selection-condition(A) could indicate terminating op_B and selecting op_A, even during the situations where op_B is active. Neither selection-condition(op_A) nor overriding-selection-condition(op_A) makes a suggestion while op_A is active, because their examples are not collected in such regions. Finally, like overriding-selection-condition(op_A), maintenance-condition(op_A) suggests that op_A should start even if another operator is still active. Unlike the other selection conditions, absence of maintenance-condition(op_A) suggests that op_A should not be started at situations where it is not active, and that op_A should be terminated, if it is active.

If our goal were programming an agent manually, having only a subset of these concepts could be sufficient. For example, the rules in Soar version 7 are closer to termination/selection conditions while the rules of Soar version 8 are closer to maintenance conditions. Nevertheless, given a representation language, a particular operator may be more compactly represented using a subset of concepts, making it easier to learn inductively.

In general, different decision concepts of an operator may have conflicting suggestions. There are several possibilities for dealing with this problem. One can commit to particular priority between the decision concepts. For example KnoMic [21] learns only selection and termination conditions. In execution, KnoMic assumes that termination conditions have higher priority. Another alternative is to have a dynamic conflict resolution strategy. For example, a second learning step could be used to learn weights for each concept such that the learned weight vector best explains the behavior traces. In this paper, we don't further explore conflict resolution strategies but we concentrate on learning individual decision concepts.

2.6 Learning Concepts

The learning component uses an ILP algorithm, currently inverse entailment [13], to learn a theory that represents decision concepts using the examples received from the training set generator, the situation predicates stored in the episodic database, and hand-coded domain knowledge.

```
selection-condition(S, go-to-door(Door) )   ←
          active-operator(S, get-item(Item) ),
          current-room(S, agent, Room1 ),
          has-door(S, Room1, Door ),
          path(S, Room1, Path),
          pathdoor(S, Path, Door),
          destination(S, Path, Room2),
          contains(S, Room2, Item).
```

Fig. 6. A desired hypothesis for the selection condition of go-to-door operator

Soar stores its binary predicates as a directed graph (Fig. 2), and regular Soar programs take advantage of this property by using only rules that instantiate the first arguments of these predicates before testing them. Fortunately, this structural constraint can be very naturally represented in inverse entailment (and many other ILP algorithms) using mode definitions and it significantly reduces the search space. Fig. 6 depicts a correct rule that is learned during the experiment reported in section 3.1. It reads as: "At any situation S with an active high-level operator get-item(Item), the operator go-to-door(Door) should be selected if Door can be instantiated with the door on the shortest path from the current room to the room where Item is in."

The learning system models the selection decision of go-to-door by checking the high-level goals and retrieving relevant information (active-operator retrieves information about the desired item), by using structured sensors (i.e. current-room), and domain knowledge (i.e. has-door, path)

During evaluation of a hypothesis, the situation predicates, such as current-room or contains, call background predicates that query the episodic database structure. active-operator is a special hand-coded background predicate that generates the parameters of the active parent operator[2] by calling the accepted-annotation predicates stored in the episodic database.

[2] The actual syntax of this predicate is slightly more complex to comply with the restrictions of the ILP algorithm used.

The operator hierarchy simplifies the search for a hypothesis in two ways. First, the decisions about an operator are learned in the context of the parent operator. The conditions for maintaining a parent operator are implicit conditions of the child operator; they don't need to be learned and as a result the conditions get simpler and easier to learn. At a level of the hierarchy, learning only the distinctions of selecting between sibling operators may be sufficient. Second, object-valued parameters of a parent operator can provide access to the more relevant parts of the background knowledge (i.e. active-operator), in effect simplifying the learning task. For example in Fig. 6, the conditions for selecting the correct door could be very complex and indirect if the parent operator did not have the Item parameter that guides the search (i.e. the towards the room that contains the item).

We have two mechanisms for encoding domain knowledge to be used in learning. In section 2.2, we described the assumed situation predicates that are added to the behavior trace as factual information the expert may be using. An alternative is to use a hand-coded theory written in Prolog. In our example in Fig. 6, the rule uses assumed knowledge about path structures between each pair of rooms (i.e. path, pathdoor, destination). An alternative would be that the agent infers that information dynamically during learning, for example using knowledge about the connectivity of neighbor rooms. For example we could have a shortest-path(+Situation, +Room1, +Room2, -Door) predicate which infers that Door in Room1 is on the shortest path from Room1 to Room2.

2.7 Agent Generation for a Particular Agent Architecture

At the end of each learning phase, the learned concepts should be compiled to an executable program in an agent architecture, in our case Soar. In general, the conditions at the if-part of the decision concepts should be "testable" by the agent program. The translation of observed situation predicates is trivial. On the other hand, for each hand-coded background predicate, we should have corresponding hand-coded implementations in the agent program. For example, while the active-operator is a prolog program that checks accepted-annotation predicate during learning, it should have an agent architecture specific implementation to be used in execution that checks and returns information about the active high-level operators.

3 Experiments

We have conducted two set of experiments to evaluate our approach. In the first experiment, we generated artificial examples for a selection condition concept in a building navigation problem. We used the inverse entailment implementation Progol [13] for that experiment. For the second experiment, we used behavior data generated by Soar agents with our learning by observation framework (Fig. 1), partially implemented in SWI-Prolog. Our program intercepts the symbolic interaction of Soar agents with the environment, stores the interactions in an episodic database, creates decision concept examples, and declarative bias (such as mode definitions), and calls the ILP engine Aleph [17] that we have embedded in our system. In these more recent experiments, we have used Aleph instead of Prolog because Aleph is more

customizable. It was easier to embed Aleph into our system because it also runs on SWI-Prolog. Using behavior of hand-coded Soar agents to create new "clone" agents allows us to easily experiment with and evaluate our framework. Since Soar agents already use hierarchical operators, it is easy to extract the required goal annotations from them. While the intercepted environmental interaction is used to create the behavior trace, the internal reasoning of the agents is only used to extract the goal annotations.

3.1 Learning from Artificially Created Data

In our first experiment the selection condition of go-to-door is learned in the context of get-item using artificially created examples. One possible correct hypothesis in this problem is depicted in Fig. 6. The goal is to learn to select a door such that it is on a path towards an item that the agent wants to get.

In this experiment, we have artificially created situations where go-to-door(Door) operator is selected. First, we generated random map structures consisting of rooms, doors, items, paths, and shortest path distances between rooms. Then, we have generated random situations by choosing different rooms for the current-room sensor, and different items as the parameter of the high-level get-item goal. Finally, we have generated positive examples of our target concept by choosing a situation and the parameter of the go-to-door operator, namely, the correct door objects that leads toward the target item.

Instead of using negative examples, we marked a varying number of positive examples with a "complete selection" tag, indicating that the expert returns all of the best parameter selections for that situation (i.e. there maybe multiple doors that are on a shortest path). We used declarative bias to eliminate hypotheses that satisfy Door variables that are not among the expert selection for these marked examples.

To cover qualitatively different cases, we have generated 6 maps using 2 possibilities for the number of items on the map (1 or 3 items) and 3 possibilities for connectivity of the rooms (0, 3, or 6 extra connections where 0 means a unique path between each pair of rooms.). In these examples, a varying number of examples are marked with the "complete selection" tag (0-5 positive examples are marked). For these 36 combinations, we have conducted 5 experiments each with 5 positive examples. We ran Progol with noise setting turned off, searching for the best hypothesis that cover all positives while satisfying declarative bias.

We measured the learned hypothesis in terms of over-generality, over-specifity and accuracy. For example if for a situation s the doors that satisfy the correct hypothesis are $\{d_1,d_2,d_3\}$ and the doors that satisfy learned hypothesis h are $\{d_1,d_2,d_4,d_5,d_6\}$, then we get: $accuracy(h, s) = 2/6$, $overgenerality(h, s) = 3/6$, and $overspecifity(h, s) = 1/6$.

To evaluate the learned hypothesis, we have created test sets consisting of 6 random maps each with 10 fully connected rooms, choosing from 3 possibilities for connectivity (0, 5, or 10 extra connections) and 2 possibilities for the number of items (1 or 3 items). We have intentionally used test maps larger than training maps to ensure that hypotheses that may be specific to the training map size are not measured as accurate during testing.

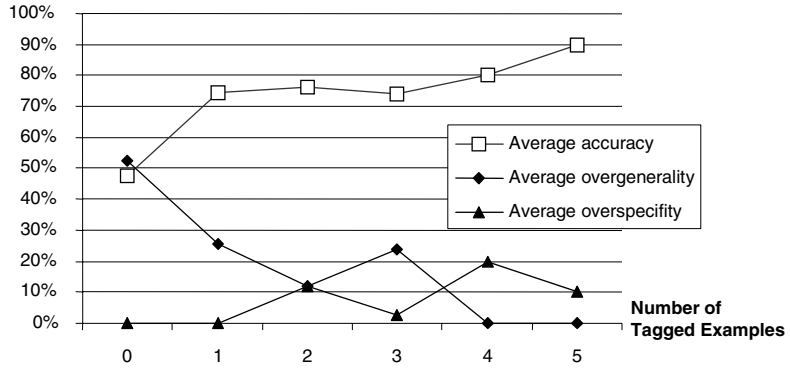

Fig. 7. Average accuracy, overspecifity and overgenerality of learned hypotheses on test data. Each data point is the average of 30 learned hypotheses

For each map, we have tested both the learned hypothesis and the correct hypothesis over all possible situations on these maps. (All possible combinations of current room and target item values.) For each of them, we have compared the output variables the hypotheses generate, namely the instantiations of the door variables. Fig. 7 shows the average *accuracy* is steadily increasing with the number of tagged examples.

3.2 Learning from Agent Program Generated Data

In this experiment, we have used the annotated behavior traces generated by a Soar agent in Haunt domain. All behavior data is created using a single level map consisting of 13 fully connected rooms that are marked with symbolic nodes to help the navigation of the agent. For each door, there are nodes on each side of the door.

The Soar agent controls a virtual character that has previously explored the level and built up an internal map of the rooms and the location of items in the level. In our experiment, we concentrated on the behavior it generates to retrieve items. The Soar agent randomly chooses an item, and selects the goal goto-room(Room) by instantiating Room with the room where the item is in. It then uses goto-node-in-room(Node) and goto-node-next-room(Node) operators to go towards Room. The agent selects goto-node-in-room operator to move to a node in front a door that leads towards Room. To go through the door, the agent chooses goto-node-next-room with a node on the other side of the door and moves towards it. These two operators are used in a loop until the agent is in the target room and the parent operator goto-room is retracted.

In this experiment, our goal is to learn the selection and termination concepts (Fig. 5 a, b) of goto-node-in-room in the context of a given goto-room operator. We have collected 3 minutes of behavior trace of the Soar agent (~30000 situations).

We have recorded several numerical sensors such as x-coordinate, distance to visible objects, and object valued sensors that monitor the last visited node, the nodes the agent can see, the nodes in front of the agent, the nearest visible node, the current room, and the previous room among others. The learning system used background

knowledge about the locations of nodes, rooms, doors, and their relation to each other. A typical situation contained over 2000 situation predicates and a typical bottom clause (generated to variable depth 4) has over 500 literals. 600 positive and 200 negatives examples are generated for each concept. We ran Aleph with its default inverse entailment strategy. The numerical sensors are only used in a limited way; only within conditions that test whether sensors are close to constant values.

Although this experiment returned the correct termination condition, we got an overgeneral theory for the selection condition that may select a random node in the current room. Probably this stems from the fact that the negative examples are generated at situations where the learned operator is not selected (Fig. 5.b). These situations do not provide sufficient information about which parameter selections would be incorrect at a situation where the learned operator is selected.

Based on this observation, we have conducted another experiment where the selection condition of goto-node-in-room is learned correctly using a slightly different approach. For each positive example $cond(s, op(x_1))$, we generated negative examples of the form $cond(s, op(x_2))$ using the same situation s but different operator parameters. In our case, x_2 would be a node that the expert has not selected in situation s. This approach resembles the positive-only learning strategy described by Muggleton [13] except that in our case, the negative examples are generated by choosing only the operator parameters randomly not the situations. We have selected these parameters randomly from the set of parameters observed in expert annotations. Using the positive examples in previous experiment and generating 20 random negatives for each, we get the correct rule in Fig. 8.

```
selection-cond( S, goto-node-in-room( TargetNode) ) ←
          active-operator(S, goto-room(TargetRoom) ),
          current-room(S, agent, CurrentRoom),
          path(S, CurrentRoom, Path),
          pathnode(S, Path, TargetNode),
          destination(S, Path, TargetRoom).
```

Fig. 8. Selection condition of goto-node-in-room operator induced using only positive examples

In this experiment, we have demonstrated that general correct concepts for selecting and terminating operators can be learned in structured domains using only correct expert behavior. Our experiment indicates that negative examples obtained at situations where an operator is not selected may not be sufficient in learning operators with parameters. Generating negative examples with random parameters may solve this problem.

4 Related Work

Khardon [7] studied learnability of action selection policies from observed behavior of a planning system and demonstrated results on small planning problems. His framework requires that goals are given to the learner in an explicit representation, while we try to inductively learn the goals.

To learn procedural agent knowledge, there are at least two alternatives to learning by observation. One approach is to learn how the agent actions change the perceived environment and then use that knowledge in a planning algorithm to execute behavior. TRAIL [1] combines expert observations and experimentations to learn STRIPS like teleoperators using ILP. OBSERVER [23] uses expert observations to learn planning operators in a rich representation (not framed in an ILP setting). Moyle [12] describes an ILP system that learns theories in event calculus, while Otero describes an ILP system that learn effects in situation calculus [15]. These systems could have difficulty if changes caused by the actions are difficult to observe, possibly because the actions cause delayed effects that are difficult to attribute to particular actions. In these cases, our approach of trying to replicate expert decisions, without necessarily understanding what changes they will cause, may be easier to learn.

Another alternative to learning by observation is to use reinforcement learning. Relational reinforcement learning [4] uses environmental feedback to first learn utility of actions in a particular state and then compiles them to an action selection policy. Recently, expert behavior traces have been combined with the traces obtained from experimentation on the environment [3]. Expert guidance helps their system to more quickly reach states that return feedback. In this system, the selections of the experts are not treated as positive examples and learning still uses only environmental feedback. In complex domains, our strategy of capturing the expert behavior may be easier than trying to justify actions in terms of future gains, especially when the reward is sparse. Moreover, replicating the problem solving style of an expert, even if he/she makes sub-optimum decisions, is an important requirement for some applications such as creating "believably human-like" artificial characters. Unlike learning by observation, none of the two approaches above are very suitable for that purpose because their decision evaluation criteria is not based on similarity to expert but success in the environment.

5 Conclusions and Future Work

We have described a framework to learn procedural knowledge from structured behavior traces, structured goal annotations and complex background knowledge. We decomposed the learning an agent program problem to the problem of learning individual goals and actions by assuming that they are represented with operators that are arranged hierarchically. We operationalized learning to use these operators by defining decision concepts that can be learned in a supervised learning setting. We have described an episodic database formalism to compactly store structured behavior data. Episodic database was crucial in testing our system in a large domain. We have partially implemented the first cycle of our framework, where the learning system uses only correct behavior data. We have conducted two experiments to evaluate our approach. In the first experiment, we used a small data set of artificially created situations. Here, the target concept is successfully learned, but we required additional expert input in addition to the correct decisions. In the second experiment, we used a large data set generated from the behavior of a hand-coded agent in a complex domain. Learning selection conditions as defined in Fig. 5 generated overgeneral results, because the learning system did not have sufficient information to eliminate

incorrect parameter selections. When selection conditions are learned with a "positive examples only" strategy, this problem is overcome and a correct concept is learned.

Our first goal for future work is to implement the second execution cycle of our framework. We predict that the behavior data obtained this way will provide valuable examples and improve learning results. A formal evaluation of our episodic database formalism is also left for future work. We are currently extending this formalism so that it not only compactly represents behavior data, but also test rules more efficiently by testing the rules on a range of situations at once.

Acknowledgements. This work was partially supported by ONR contract N00014-03-10327.

References

1. Benson, S., Nilsson, N.: Inductive Learning of Reactive Action Models. In: Prieditis, A., Russell, S.: Machine Learning: Proceedings of the Twelfth International Conference. Morgan Kaufmann, San Fransisco, CA (1995) 47-54
2. Camacho, R.: Inducing Models of Human Control Skills. In: 10th European Conference on Machine Learning (ECML-1998). Chemnitz, Germany (1998)
3. Driessens, K., Dzeroski, S.: Integrating Experimentation and Guidance in Relational Reinforcement Learning. In: Sammut, C., Hoffmann, C. (eds.): Proceedings of the Nineteenth International Conference on Machine Learning (ICML-2002). Morgan Kaufmann Publishers (2002) 115-122
4. Dzeroski, S., Raedt, L. D., Driessens, K.: Relational Reinforcement Learning. Machine Learning **43** (2001) 5-52
5. Isaac, A., Sammut, C.: Goal-directed Learning to Fly. In: Fawcett, T., Mishra, N. (eds.): Proceedings of the Twentieth International Conference (ICML 2003). AAAI Press (2003)
6. Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P. G., Koss, F. V.: Automated Intelligent Pilots for Combat Flight Simulation. AI Magazine **20** (1999) 27-42
7. Khardon, R.: Learning to Take Actions. Machine Learning **35** (1999) 57-90
8. Laird, J. E., Newell, A., Rosenbloom, P. S.: Soar: An Architecture for General Intelligence. Artificial Intelligence **33** (1987) 1-64
9. Magerko, B., Laird, J. E., Assanie, M., Kerfoot, A., Stokes, D.: AI Characters and Directors for Interactive Computer Games. In: The 16th Innovative Applications of Artificial Intelligence (IAAI-2004). San Jose, CA (2004)
10. Michie, D., Bain, M., Hayes-Michie, J.: Cognitive Models from Subcognitive Skills. In: Grimble, M., McGhee, J., Mowforth, P. (eds.): Knowledge-Based Systems in Industrial Control. Peter Peregrinus, Stevenage (1990) 71-90
11. Michie, D., Camacho, R.: Building Symbolic Representations of Intuitive Real-Time Skills From Performance Data. In: Machine Intelligence 13 Workshop (MI-1992). Glasgow, U.K (1992)
12. Moyle, S.: Using Theory Completion to Learn a Robot Navigation Control Program. In: Matwin, S., Sammut, C. (eds.): Inductive Logic Programming, 12th International Conference, Revised Papers. Lecture Notes in Computer Science, Vol. 2583. (2003) 182-97
13. Muggleton, S.: Inverse Entailment and Progol. New Generation Computing **13** (1995) 245-286
14. Newell, A.: Unified Theories of Cognition. Harvard Univ. Press (1990)

15. Otero, R. P.: Induction of the Effects of Actions by Monotonic Methods. In: Horváth, T. (ed.): Inductive Logic Programming, 13th International Conference. Lecture Notes in Computer Science, Vol. 2835. Springer (2003) 299-310
16. Sammut, C., Hurst, S., Kedzier, D., Michie, D.: Learning to fly. In: Sleeman, D., Edwards, P. (eds.): Proceedings of the 9th International Conference on Machine Learning (ICML-1992). Morgan Kaufmann (1992) 385-393
17. Srinivasan, A.: The Aleph 5 Manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph (2003)
18. Struyf, J., Ramon, J., Blockeel, H.: Compact Representation of Knowledge Bases in ILP. In: Matwin, S., Sammut, C. (eds.): Inductive Logic Programming, 12th International Conference, Revised Papers. Lecture Notes in Computer Science, Vol. 2583. Springer, Germany (2003)
19. Suc, D., Bratko, I.: Problem decomposition for behavioural cloning. In: 11th European Conference on Machine Learning. Lecture Notes in Computer Science, Vol. 1810. Springer-Verlag, Germany (2000) 382-391
20. Urbancic, T., Bratko, I.: Reconstructing Human Skill with Machine Learning. In: Cohn, A. G. (ed.): Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94). John Wiley and Sons (1994) 498-502
21. van Lent, M., Laird, J.: Learning procedural knowledge through observation. In: Proceedings of the International Conference on Knowledge Capture (KCAP-2001). ACM Press, New York (2001) 179-186
22. van Lent, M.: Learning Task-Performance Knowledge through Observation. Ph.D. Thesis. Univ. of Michigan (2000)
23. Wang, X.: Learning Planning Operators by Observation and Practice. Ph.D. Thesis (Tech. Report CMU-CS-96-154). Computer Science Department, Carnegie Mellon University (1996)
24. Wray, R. E., Laird, J. E., Nuxoll, A., Stokes, D., Kerfoot, A.: Synthetic Adversaries for Urban Combat Training. In: Proceedings of Innovative Applications of Artificial Intelligence (IAAI-2004). AAAI Press, in press