Mach Learn (2006) 64:263–287
DOI 10.1007/s10994-006-7734-8

Learning goal hierarchies from structured observations and expert annotations

Tolga Könik · John E. Laird

Received: 12 April 2005 / Revised: 3 December 2005 / Accepted: 11 February 2006 / Published online: 17 May 2006

Abstract We describe a *relational learning by observation* framework that automatically creates cognitive agent programs that model expert task performance in complex dynamic domains. Our framework uses observed behavior and goal annotations of an expert as the primary input, interprets them in the context of background knowledge, and returns an agent program that behaves similar to the expert. We map the problem of creating 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 represented with first order rules. Using an inductive logic programming (ILP) learning component allows our framework to naturally combine structured behavior observations, parametric and hierarchical goal annotations, and complex background knowledge. To deal with the large domains we consider, we have developed an efficient mechanism for storing and retrieving structured behavior data. We have tested our approach using artificially created examples and behavior observation traces generated by AI agents. We evaluate the learned rules by comparing them to hand-coded rules.

Keywords Relational learning by observation · Relational learning · Inductive logic programming (ILP) · Behavioral cloning · Cognitive agent architectures

Editor: Rui Camacho

T. Könik (✉)
Computational Learning Laboratory, Center for the Study of Language and Information, Stanford University, Stanford, CA 94305, USA
e-mail: konik@stanford.edu

J. E. Laird
Artificial Intelligence Laboratory, Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109, USA
e-mail: laird@umich.edu

1. Introduction

Developing cognitive 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. In this paper, we explore the use of machine learning techniques to automate this process. We present a learning by observation framework that automatically creates agent programs 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 cognitive agents.

Learning to replicate behavior from only expert observations is sometimes 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 (Michie et al., 1990), controlling a simulated aircraft (Sammut et al., 1992; Michie and Camacho, 1992), or operating a crane (Urbančič and Bratko, 1994). 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 (1998) induced controllers that had goal parameters so that the execution system can use the same controllers under varying goal settings. It did not however learn how to set the goal parameters. Bain and Sammut (1999) discuss a two step approach where a goal model, a mapping from states to goal parameters, and an effect model, a mapping from control actions to changes in the state, are separately learned. The execution system selects control actions that will achieve the goal values by interpreting the effect rules. Isaac and Sammut (2003) also present a two step approach where an anticipatory level sets goal values and PID controllers at the lower level produce control actions to reduce the error between the goal and state values. Šuc and Bratko (2000) describe induction of constraints that model qualitative trajectories which the expert is trying to follow to achieve goals. These constraints are used to guide the choice of the control actions.

As in the goal-directed behavioral cloning research described above, representing goals explicitly is an important component of our approach for obtaining complex and flexible agents, but the goals in our framework are used in a quite different way. In the systems described above, the goals are desired values for some predefined parameters of a dynamic system. For example, the learning-to-fly domain has goal parameters such as target turn-rate. In contrast, the goals in our framework correspond to durative high-level internal states of the expert indicating that a particular kind of behavior is desired. These goals maybe related to a final state the expert wants to achieve, such as a `go-to-room(`$\mathtt{r}_1$`)` goal in a building navigation domain; they may be about maintaining a condition such a `maintain-altitude` goal in an airplane control domain; or they may simply represent the desire to exhibit a complex behavior as in `fly-in-a-circle` goal. Unlike the above approaches, we don't assume pre−existing definitions for the goals. In contrast, the meaning of each goal in our framework is discovered by learning under which circumstances the expert selects it as well as learning the behaviors that become relevant once it is selected. The goals are hierarchically organized so that the goals at the higher levels of the hierarchy correspond to more complex behavior.

One of the key challenges of learning by observation is that the expert's mental reasoning is not directly available to the learner. To tackle this difficulty, we use additional information sources such as background knowledge about the task and annotations of the observed behavior that specify the expert's goals. This additional input helps our learning system to

model the reasoning of the expert. Our system first learns how the experts select goals based on observed situations, background knowledge, and their active goals. Then it learns to select actions that exhibit behavior consistent with the selected goals.

van Lent's (2000) learning by observation framework also learns hierarchies of durative goals but its attribute-value based representation limits its ability to model the expert's reasoning. It would run into difficulties when structured properties of the environment are relevant, for example if it has to make decisions involving multiple objects (i.e. two enemy planes in a tactical air combat domain), if complex knowledge about the task (i.e. a building map in a navigation domain) is important in choosing the right strategy, or if the decisions of the expert must involve inference beyond the directly observed features of the external world (i.e. choosing a door towards a room that is not directly observed). In addition, KnoMic uses a simple single-pass specific-to-general learning approach that is not feasible with structured behavior data in the complex domains we consider.

Our *relational learning by observation* framework proposes a natural solution for the above limitations by using a first order language to uniformly represent information from multiple sources. This allows the use of structured behavior observations represented as temporally changing relational structures, parametric and hierarchical goal annotations, and complex background knowledge. On the other hand, both the goal annotations and the background knowledge are optional, but in their absence our framework will be reduced to behavioral cloning and the complexity of knowledge it can capture will be more limited.

We reduce the "behave like an expert" learning problem, to a set of supervised learning problems that can be framed in an Inductive Logic Programming (ILP) setting, where first order rules can be learned from structured data. To be able to use ILP algorithms in the large domains we consider, we devised an efficient mechanism to store and access structured behavior data.

We use the general agent architecture Soar (Laird et al., 1987) as the execution system of our target agent program. 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 agent systems.

The paper is organized as follows. In Section 2, we analyze the context in which we investigate learning by observation. In Section 3, we describe our learning by observation framework. In Section 4, we present experimental results. In Section 5, we discuss related work. Finally, we conclude with remarks about future directions in Section 6.

2. Design decisions

In this section, we list constraints we pose on the space of learning by observation systems we explore. We hope that this will help to make our major design decisions and assumptions more explicit (Fig. 1).

Available information sources

We assume that our system can use observation traces of experts' task-performance behavior consisting of the situations the experts encounter and the actions they execute (**A1**), observation traces of previously learned agents' task-performance behavior (**A2**), and annotations

Available Information Sources
A1. Observation traces of experts' task-performance behavior
A2. Observation traces of agent programs' task-performance behavior
A3. Goal annotations of the behavior observation traces
A4. Factual and common sense background knowledge

Representation of Captured Knowledge
A5. Durative actions and goals
A6. Hierarchical goals
A7. Object-valued and constant-valued goal and action parameters
A8. Rule based symbolic architecture
A9. Reactive agent architecture

Properties of the Learning Algorithm
A10. First order rules in hypothesis space
A11. Using complex background knowledge in learning
A12. Good generalizing learning algorithm
A13. Robust learning algorithm

Learning Bias and Strategies
A14. Learning multiple concepts to represent goals
A15. Testing sensory conditions in hypothesis
A16. Testing domain knowledge in hypothesis
A17. Testing active high-level goal conditions in hypothesis
A18. Testing goal completed beliefs

Fig. 1 Our assumptions about the systems we investigate. A1–A4 describe the assumed problem, while A5–A18 are additional constraints we pose on the solution space

indicating which goals are being pursued throughout an observed behavior (**A3**). The goal annotations contain only the names of the goals and their parameters (i.e. `goto-door(`d_1`)`) and do not describe their meanings. Instead, their meanings are learned through observation. The behavior observations are interpreted in the context of hand-coded background knowledge such as factual knowledge about the environment (i.e. map of a building) and task relevant common sense knowledge (i.e. rooms are connected through doors) (**A4**).

The background knowledge is not necessarily obtained from the expert being modeled, and does not need to correspond to the expert's understanding and reasoning about the world. The task of the learning system is not to exactly replicate the internal reasoning process of the expert, but to create a model that exhibits similar behavior. Of course, the parts of the background knowledge that resembles the expert's internal knowledge may be more useful during this process. Therefore, while the goal annotations of the behavior observations must be obtained from the expert that exhibits them, a separate knowledge engineer can encode the background knowledge. Although encoding commonsense background knowledge may be difficult, it may be shared in learning multiple tasks. Moreover, existing general commonsense theories can be used if they are relevant in a domain. (i.e. a qualitative spatial theory in a building navigation domain). Factual background knowledge should be specific to an environment (i.e. the objects in a room), but that is typically easier to encode.

2.1. Representation of captured knowledge

We assume that the target agent program will have durative goals, which they achieve by maintaining durative actions continuously in a real-time environment (**A5**). In planning literature the term goal is often used as a predefined condition that holds in desired end-states. The goals in our framework have a more general meaning. They represent durative internal states of an agent indicating that particular kinds of behaviors are appropriate. They can represent intensions to achieve a particular condition, like in the planning sense, but they may be also about maintaining a process, such as watching a television show. We assume that the goals are structured in a hierarchy (**A6**) in order to represent complex behavior while keeping the learning tasks more manageable. The goals and actions may contain both constant valued and object valued parameters (**A7**) (i.e. `fly-at-altitude (4000)` in a flight simulator domain or `go-to-door(`$\mathtt{d}_1$`)` in a building navigation domain). The parameters of a goal improve the generality of the captured knowledge. The object-valued goal annotations also help the learning system by providing structured information about the internal reasoning of the expert. For example when the expert annotates a behavior with the goal `go-to-room(`$\mathtt{r}_1$`)`, by choosing the room object r_1, the expert points to information related to that room, such as where on the map that room is or which items it contains. At the end of learning, the captured knowledge is transformed to a program that can run in a rule based (**A8**) reactive agent architecture (**A9**).

2.2. Learning algorithm

Since the domains we consider are rich in structure, the learning algorithm we use should be able to generate hypotheses that can test structured conditions in a first order language (**A10**). Therefore, we frame our problem in a general ILP setting, without necessarily committing to a particular ILP algorithm. Since experts may use implicit knowledge that cannot be sensed from the environment directly, our learning algorithm should be able to use background knowledge that encodes factual or common sense knowledge (**A11**). Being able to deal with complex background knowledge is one of the major advantages of ILP algorithms over traditional feature-attribute based machine learning techniques. Moreover, due to variables in relations, ILP algorithms are suitable to abstract the details of the specific examples (**A12**). For example, to learn the skills to move towards a door, our system would not need to learn separate concepts specific to each door, requiring separate set of examples. Instead, the general knowledge of how to move towards any door will be learned using all of these examples. One implication is that we might require fewer expert behavior traces, probably the most costly resource in our problem. Finally, we need learning algorithms that can deal with noise because the background theory used in learning will probably model the expert's reasoning only approximately (**A13**). Moreover, it is not realistic to assume that the expert's behavior is always consistent with his/her goal annotations.

2.3. Learning strategies and bias

We represent how the expert maintains the activity of a goal using multiple learned concepts (**A14**). For example a goal may be represented by two concepts, one indicating when the goal is selected and the other indicating when the goal is terminated. Alternatively, a concept may represent the conditions that hold as long as a goal is active. Although these two schemes could be converted to each other if the knowledge representation is rich enough, one of them

might represent a particular goal in a more compact way, which could be learned easier and with higher confidence.

The conditions in the learned rules will need to refer to current sensor values so that the agent's behavior can be conditional on changes in the environment (**A15**). To represent internal knowledge and reasoning of the expert, the conditions in the hypothesis may refer to relations that are defined in the background knowledge (**A16**), as well as active higher level goals. For example when the `open(Door)` goal is learned in the context of a previously selected goal `drive(Car)`, which door should be opened depends on which car is being driven; that is, it must be a door of that particular car (**A17**).

During performance, the expert may acquire beliefs based on his observations that may persist even after the reasons for acquiring them are not observable. In general, it is difficult to learn decision processes that are based on such beliefs because they will not be visible to our system. Like Knomic (van Lent and Laird, 2001), we use a very limited form of beliefs about the completed goals. Such beliefs can help the agent to maintain information about its progress towards its goals. We assume that the agents can create internal belief structures about the goals they have completed and use these facts in subsequent decision making (**A18**).

3. Relational learning by observation framework

Figure 2 depicts our *relational learning by observation* framework. Its execution cycle has two modes. In the first mode, the expert generates behavior by selecting actions using a behavior interface, leading to the creation of an initial, approximately correct agent program.

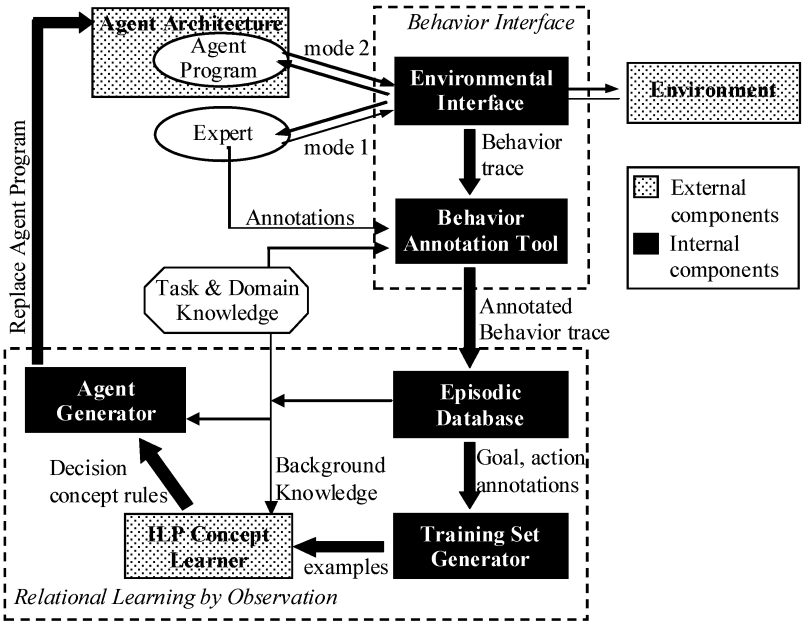

Fig. 2 Our relational learning by observation framework. In mode 1, the expert generates behavior. In mode 2, the expert gives feedback on the behavior generated by a previously learned agent program

In the second mode, a previously learned agent program generates behavior while the expert is giving feedback, leading to the creation of an improved agent program.

The behavior generated in both modes is recorded as a *behavior trace* structure, which contains the selected actions and a symbolic representation of the observed situations. 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 goal annotations and the expert marks them as accepted or rejected depending whether they are consistent with previously accepted goals and behavior. The resulting *annotated behavior traces* are returned to a relational learning by observation component that interprets them in the context of task and domain knowledge to induce an agent program. The newly created agent program can generate further traces, until the expert is satisfied with the performance of the agent program and the training is terminated. At each cycle, a new agent program is learned from scratch but since more behavior traces have been accumulated, a more accurate agent program is expected to be learned. At any time during the agent program's performance (mode 2), the expert can intervene and take control (mode 1) to generate traces, such as when the agent program is performing poorly. This can help focus the learning on those parts of the task where the agent program is most lacking sufficient knowledge.

The annotated behavior traces, which are the primary input of the relational learning by observation component, consist of (1) *situations*, temporally changing relational structures symbolically representing the environment from the perspective of the agent exhibiting the behavior, (2) *action annotations*, the names and parameters of selected actions that generate the actual behavior, and (3) *goal annotations* that mark situations with the names and parameters of goals motivating the selected actions. Both the actions and goal annotations are durative and may persist over consecutive situations. For example the expert may annotate a list of situations as "In this time interval, I am pursuing the goal `go-to-room(`$\mathtt{r}_1$`)` and to achieve that, I am pursuing the action `move(forward)`". Moreover the action and goal annotations are marked as "accepted" or "rejected". While the expert generated actions and goals are marked as accepted, the markers on the agent program generated actions and goals are determined by the expert feedback. The output of the relational learning by observation is an agent program that takes the perceived situations as input and returns a list of active goals and actions at each situation.

The annotated behavior trace generated in each cycle is inserted into an *episodic database* that efficiently stores and retrieves the observed behavior and the expert annotations. The training set generator component maps the problem of "obtaining an agent program" to multiple problems of learning *decision concepts* that can be represented in a "supervised concept-learning" setting. The decision concepts include learning when the goal and actions should be selected, when they should be terminated, or when they should be interrupted. For each decision concept, the training set generator creates positive and negative examples using the annotated behavior traces stored in the episodic database. The concept learner component uses an ILP algorithm that learns rules for each decision concept, using the examples in the training set and as background knowledge the hand-coded task & domain theory and the annotated behavior traces. The agent generator component converts the decision concept rules into an executable agent program for a particular agent architecture.

We have partially implemented this framework to conduct the experiments reported in Section 4. Although we have implemented the relational learning by observation component to use behavior annotations with both correct and incorrect behavior, the rest of our implementation works in the first mode of the execution cycle where only correct behavior instances are used. Since we don't want to assume an initial agent program, it is crucial to

demonstrate some learning with correct behavior examples only. Therefore, in this paper we mainly focus on the first execution mode of our framework. Our experiments indicate that the first mode may be sufficient to capture behavior performance knowledge.

In our experiments, instead of human expert generated behavior, we use behavior of hand-coded Soar agents. 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. We built a general interface that can extract behavior and goal annotations from Soar agents on any environment Soar has been connected to. Of course learning from Soar agents does not guarantee learning from human experts. We plan to address this issue in future research. Könik et al. (2005) describes a preliminary implementation of a *learning from diagrammatic behavior specifications* system, where a relational learning by observation component uses behavior scenarios interactively specified by a human expert and an agent program using a graphical interface.

3.1. Target agent architecture and environments

We use the general agent architecture Soar (Laird et al., 1987) as the target performance system for our learning framework and Soar influences how the knowledge in our system is represented. Nevertheless, the knowledge we extract is independent of a particular architecture and can be converted to be used in other execution systems. Using a general architecture such as Soar allows us to benefit from architectural mechanisms during execution, while keeping our framework domain independent. A general architecture also provides a framework where learning by observation may be integrated with other learning strategies in future. A long-term motivation is that Soar is one of the few candidates of unified cognitive architectures (Newell, 1990) and has been successful as the basis for developing knowledge-rich agents for complex environments (Magerko et al., 2004; Wray et al., 2004; Jones et al., 1999). 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" (Magerko et al., 2004), which is a 3-D first person perspective adventure game built using the Unreal game engine. This environment has a large, structured state space consisting of objects that are interrelated in a time-varying fashion, have unobservable features, require real time decisions, exist in continuous space, and include external agents and events.

3.2. Representation of the environment: Situations

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 the high-level reasoning of a cognitive agent than low-level control, we assume that the agents interact with the environment using a behavior interface that converts the raw data to a *symbolic environmental representation* (*SER*). While the expert makes his/her 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 the behavior interface, which is responsible for implementing these actions in the environment at the control level.

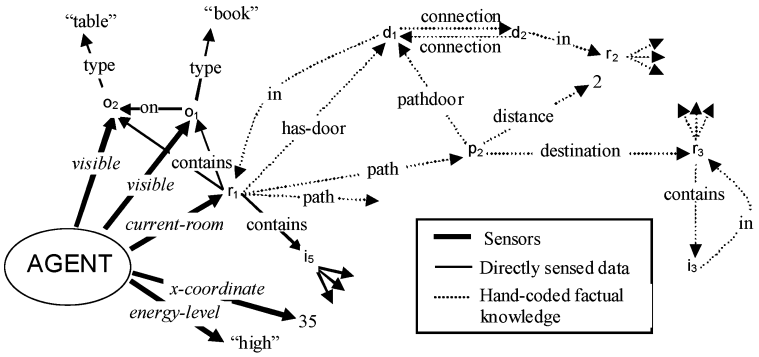

Fig. 3 A snapshot of the data maintained in the symbolic environmental representation (SER) in Haunt 2 domain. SER dynamically updates directly sensed relations and associates factual background knowledge with the sensed objects

At any given moment, SER contains 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 ground literals 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, an example "snapshot" of this time varying representation is depicted in Fig. 3. 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, r1)`. 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, the behavior interface builds the corresponding objects in SER and binds an "on" relation between them. The behavior interface also has the responsibility of associating the directly sensed features of the environment with background knowledge specific to particular objects in the environment. For example in Fig. 3, we not only see that the agent is in the room $\mathtt{r}_1$, but we also know that it can enter the room $\mathtt{r}_2$ by going through door $\mathtt{d}_1$. During the learning phase both the observed dynamical features and specific background knowledge are used in a uniform way.

3.3. Task performance knowledge: Goals and actions

We assume that the task-performance knowledge of the target agent program is decomposed into a hierarchy of operators that represent both the goals that the agent pursues and the actions that it takes to achieve these goals (Fig. 4). The primitive operators at the leaves represent actions that the agent can execute on the environment and the remaining operators represent the goals of the agent. With the operator hierarchy assumption, we decompose the "learning an agent program" problem to multiple "learning to maintain the activity of an operator" problems. The suboperators correspond to strategies that the agent can use as part of pursuing 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 its parameters. It then executes the operator by

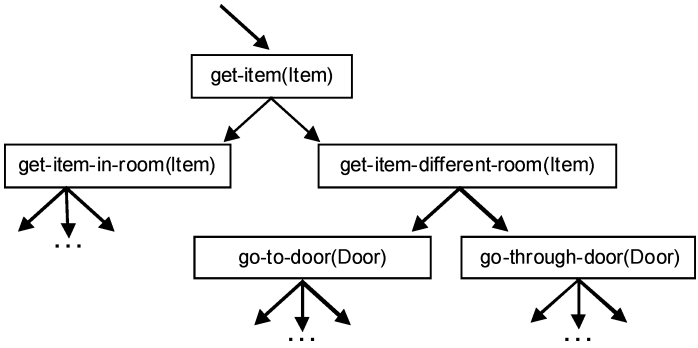

Fig. 4 An operator hierarchy in a building navigation domain

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 behavior interface, 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, except the lowest level operators representing the actions, which can be executed in parallel. This assumption 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. The operators at any level can be retracted in response to the perceived events, in which case all of its suboperators are also retracted and another operator at the same level is selected.

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. The suboperators may be executed in complex orderings to achieve the goal of the parent operator, depending on the observed situations and internal knowledge. The real execution occurs with lowest level operators representing SER actions such as `go(forward)` or `turn(left)`.

For example in Fig. 4, assume that the agent decides to get an item $\mathtt{i}_1$ by selecting `get-item(Item)` and instantiating $\mathtt{Item} = \mathtt{i}_1$. If the agent is not in the same room with $\mathtt{i}_1$, it selects the suboperator `get-item-different-room(`$\mathtt{i}_1$`)`. Then the agent executes this operator using its suboperators `go-to-door` and `go-through-door`. The operator `go-to-door` is used to select and approach a door leading towards the item $\mathtt{i}_1$. When the agent is close to that door, `go-to-door` is replaced with `go-through-door`, which moves the agent to the next room. Once in the next room, the agent selects `go-to-door` again but this time with a new door instantiation. This process continues until agent is in the same room with $\mathtt{i}_1$ and `get-item-different-room` is retracted with all of its suboperators and replaced with `get-item-in-room`.

The initial knowledge that the system has about the operator annotations are the names of the operators, the values of their arguments, pointers to their parent operator annotations, and markers indicating whether they are accepted or rejected annotations. 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).

3.4. Annotated behavior trace

While the expert or the agent program is performing a task, the symbolic state of the environment is recorded into a structure called a *behavior trace*. The symbolic representation that 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 facts 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 their parameter instantiations. A valid selection at a situation that satisfies the semantics of the operator hierarchy must form a connected path of operators starting from the root of the hierarchy (i.e. `get-item, get-item-different-room, go-to-door`,... in Fig. 4). The action annotations can be recorded by SER automatically without requiring any expert effort. Since the annotations are durative, the expert doesn't have to annotate each situation, instead, he/she can annotate only the situations where the annotations change. In the second execution mode, the expert inspects the annotated behavior traces proposed by the agent program and accepts or rejects the annotations.

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 $op(x)$.

3.5. 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.

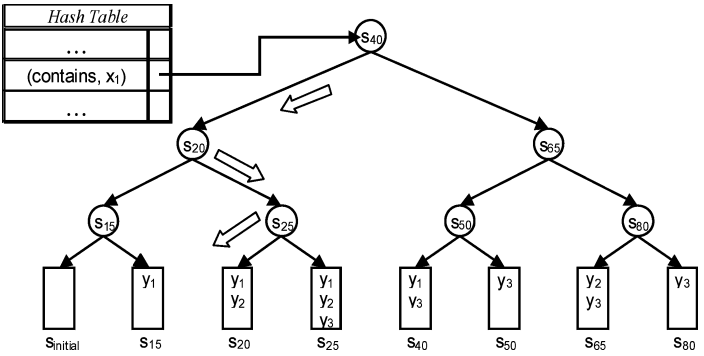

Fig. 5 Search for the query `contains(s23, x1, Y)` in the episodic database, which succeeds with $\mathtt{Y} = \mathtt{y}_1$, and $\mathtt{Y} = \mathtt{y}_2$

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 instantiated or not instantiated output variables. The result is a list of variable bindings for uninstantiated variables satisfying the query.

In episodic database, each situation predicate is stored using multiple binary trees (Fig. 5) that are indexed by the name of the predicate and input vector instantiation. The leaves store the output values explicitly for each situation they change and the nodes form a binary search tree to access these leaves efficiently. More formally, for each pair (p, x), where p is a situation predicate and x is an instantiated 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. 5, we have part of the structure that represents the observed instances of `contains(+Situation, +Room,−Item)`. This particular tree shows that room x_1 does not contain any objects in the initial situation. At situation $\mathtt{s}_{15}$, the item $\mathtt{y}_1$ appears in the room $\mathtt{x}_1$. No changes occur until the situation $\mathtt{s}_{20}$ when a new item $\mathtt{y}_2$ is added to the room and so on. For example, to answer the query `contains(s23, x1, Item)`, first the correct index tree associated with the pair `(contains, x1)` is located using a hash table, then by a binary search, the last change before $\mathtt{s}_{23}$ is located. In this case, the last change occurs at $\mathtt{s}_{20}$ and `Item` will be instantiated with $\mathtt{y}_1$ and $\mathtt{y}_2$.

In our system, 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 are reduced to single nodes. The expert annotation predicates are also stored in episodic database by using the operator name as an 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 or there are predicates that have multiple mode definitions (input/output variable specifications) each

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

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 in the binary situation predicate representation used by Soar.

Struyf et al. (2003) describe a general formalization for compactly representing ILP background knowledge in domains that have redundancy between examples, such as in the case of consecutive situations in our case. Their system would represent our situation predicates by storing a list of added and deleted predicates 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 situation, 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 the behavior trace.

3.6. Decision concepts and generating examples

In Section 3.2, 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 a supervised ILP learning setting. This decomposition allows the system to learn each decision concept independently (such as when to select an operator and when to terminate an operator). This approach is taken because the decision concepts are often highly disjunctive—that is there are many reasons for selecting an operator and many independent reasons for terminating an operator. After learning, the system can dynamically combine decision concepts learned from different examples. If the decision concepts were not learned independently, the system would have to explicitly learn every combination of decision concepts, requiring many more examples and leading to less robust performance.

A decision concept of an operator *op* is a mapping from the internal state and external perception of an agent to a "decision suggestion" about the activity of *op*. Figure 6 depicts 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 continued during its application), and `termination-condition` (when the operator has completed and should be terminated). For each decision concept, we must define how their examples should be constructed from the behavior 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 if `selection-condition(S, go-to-door(Door))` holds for a situation $\mathtt{S} = \mathtt{s}_0$ and door object $\mathtt{Door}=\mathtt{d}_0$, it represents advice indicating that the agent should select the operator $\mathtt{go\text{-}to\text{-}door(d}_0\mathtt{)}$ at situation $\mathtt{s}_0$. The goal of learning is to learn a first order concept of form:

$$con\,(s, op\,(x)) \leftarrow P\,(s, x) \qquad (1)$$

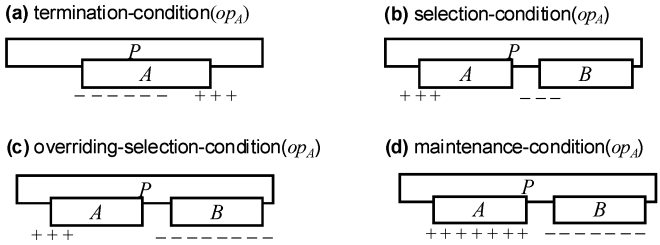

Fig. 6 The positive and negative example regions of different concepts. The horizontal dimension corresponds to the change in situations over time. A, B, and P are the accepted annotation regions of the operators op_A, op_B, and their parent operator $parent(op_A)$. Positive and negative example regions are marked with "+" and "−" symbols

where P is a condition that can match the situation predicates in the episodic database, or hand-coded background predicates. The positive and negative examples of decision concepts are ground terms of the form $con(s_0, op(x_0))$. The training set generator constructs these examples using the accepted and rejected annotations provided by the expert and the ILP component uses them in learning hypotheses of the form (1).

The examples of a concept are created from a set of situations S called *positive (negative) example regions* such that for each $s_0 \in S$, $con(s_0, op(x_0))$ is a positive (negative) example. Figure 6 depicts the positive and negative example regions of an operator op_A, for different kind of decision concepts. The horizontal dimension represents consecutive temporal 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. According to our basic hierarchy assumption, for a given operator op_A that is not the top[2] operator, there is a unique $parent(op_A)$. op_B may be the same kind of operator with op_A, but it should have a different parameter instantiation. For example, 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. 6(b)). If we have op_A=`go-to-door(`$\mathtt{d}_1$`)`, $A = \mathtt{s}_{20} - s_{30}$, and $B = \mathtt{s}_{50} - s_{60}$, we could have the positive example `selection−condition(`$\mathtt{s}_{20}$`, go-to-door(`$\mathtt{d}_1$`))` and the negative example `selection-condition(`$\mathtt{s}_{50}$`, go-to-door(`$\mathtt{d}_1$`))`.

In general, the examples of decision concepts of an operator op_A are selected only from situations where there is the appropriate context in which 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 suggestions about the activity of operators. 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 an op_B is still active because such situations are not considered as

[2] The top level goal is an exception and has no parent goal. We assume that no decision concept needs to be learned for it, because it is either determined externally or it is active throughout the lifetime of the agent.

examples for `selection-condition`(op_A). On the other hand, `overriding-selection-condition`(A) indicates terminating op_B and selecting op_A, even during the situations where op_B is active since its negative examples are collected throughout a region 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 since the positive examples of this concept are collected through all situations where op_A is active.

If our goal were programming an agent manually, having only a subset of these concepts for each operator would 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. However when learning, having a language with the flexibility to represent a broader range of operator concept is advantageous because 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 a particular priority between the decision concepts. For example KnoMic (van Lent and Laird, 2001) learns only concepts similar to our 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 do not explore conflict resolution strategies but instead concentrate on the question of whether the individual decision concepts can be learned using ILP.

The negative example regions in Figs. 6(b)–(d) implicitly assume that the selection of an operator op_A is undesirable when another operator op_B is selected. This assumption does not hold in domains where the desired agent behavior is non-deterministic, that is there may be situations where the agent should select randomly among equally good alternatives. In such domains, the negative examples of selection concepts should be treated weaker, as preferences rather than completely accurate examples.

The parametric nature of our operators causes another difficulty in collecting negative examples. The negative examples in Figs. 6(b)–(d) indicate when not to select an operator op_A, but our early experiments have shown that they may not give enough information about which operator parameter instantiations would be incorrect at a situation where op_A has to be selected. To deal with this problem, we describe an additional negative example generation scheme. For each positive example $con(s_0, op(x_1))$, we generate *heuristic negative examples* of the form $con(s_0, op(x_2))$ using the same situation s_0 but an operator parameter vector x_2, which was used in a different selection of the same operator (Fig. 7). Again, while modeling nondeterministic behavior, these negative examples should be treated as preferences; because a selection of $op(x_1)$ does not guarantee that another selection $op(x_2)$ would be wrong. During learning with these examples, we search for compact hypotheses that cover all positive examples while minimizing coverage of negative examples. This approach is very similar to the positive-only learning strategy described by Muggleton (1995) except that in their algorithm, the negative examples would be created by randomly choosing all variables in the examples, including the situations. Compared to Muggleton's approach, we

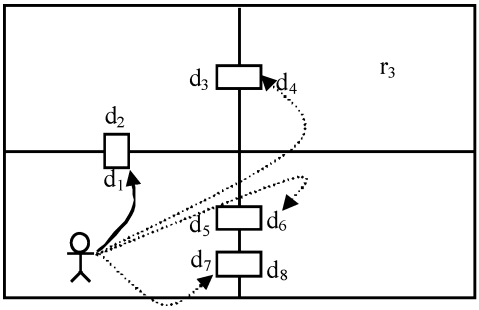

Fig. 7 Heuristic negative examples are generated by changing the operator parameters of a positive example to parameters observed in other situations for that operator

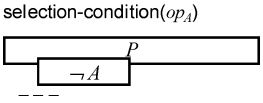

Fig. 8 Negative examples for selection-condition, extracted from a rejected annotation $\neg A$, where the expert rejects the agent program's annotation op_A

use special properties of our situation based representation, to obtain more relevant negative examples.

If data collected in the second execution cycle of our framework is available, where the expert evaluates the agent's generated behavior and annotations, we get an opportunity to extract more reliable negative examples for the selection condition (Fig. 8). The regions where the expert rejects the selection of an operator $op(x_1)$ at a situation s_0, provide reliable negative examples of selection-condition $(s_0, op(x_1))$. Nevertheless, to get to the second cycle, we need our framework to be capable of learning in the first cycle an approximate agent that can generate some behavior. Therefore, in the experiments in Section 4 we focus on learning from expert generated behavior.

3.7. Learning concepts

The learning component of our framework uses an ILP algorithm, currently inverse entailment (Muggleton, 1995), to learn a theory that represents decision concepts. The learning uses the examples generated by the training set generator and the background knowledge consisting of a hand-coded domain theory and the situation predicates stored in the episodic database.

This algorithm first generates a very specific rule, called the bottom clause, using a single positive example. In the next step, rules consisting of substructures of this bottom clause are generated in a heuristic search, where each one of these rules is tested for coverage of positive and negative examples.

In our case, for a randomly selected positive example $con(s_0, op(x_0))$, the list of all facts that hold in the situation s_0 are collected. These facts may come not only from the episodic database, but they can also be results of inferences made by the hand-coded background theory. Next, each fact in the list is generalized by replacing all objects with variables to obtain the bottom clause (Fig. 9). Note that we use Prolog syntax where the capitals are variables. We chose semantically meaningful variable names for presentation purposes although the actual algorithm generates random variable names.

```
selection-condition(S, go-to-door(Door)) ←
        active-operator(S, get-item(Item))        and
        current-room(S, agent, Room1)             and
        door(S, Room1, Door)                      and
        door(S, Room1, Door2)                     and
        connected-door(S, Door2, Door2)           and
        in-room(Door2, Room2)                     and
        contains(Room2, Door2)                    and
        ...
        contains(Room3, Item)                     and
        ...
```

Fig. 9 The most specific (bottom) clause of the learned hypothesis

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

Fig. 10 A desired hypothesis for the selection condition of `go-to-door` operator

Figure 10 depicts a correct hypothesis that is learned with this process in the experiment reported in Section 4.1. It reads as: "At any situation section`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".

Here, 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. `contains, path`).

The object valued parameters of the parent operator simplify the learning task, by providing access to the more relevant parts of the background knowledge. For example in Fig. 10, 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).

3.8. Agent generation for a particular agent architecture

At the end of each learning phase, the learned concepts must be compiled to an executable program for the target agent architecture. In general, the conditions at the if-part of the decision concepts must be "testable" by the agent program. The difficulty of this translation depends on the mechanisms provided by a particular architecture. Since Soar automatically provides mechanisms to maintain an operator hierarchy and since we represent the situations similar to how Soar represents them, the translation to Soar rules is not difficult. The only complication is that for each hand-coded background predicate, we must 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.

4. Experiments

We have conducted three experiments to evaluate the learnability of the decision concepts using ILP. In the first two experiments, we generated artificial examples for an operator selection concept in a building navigation problem. In the third experiment, we used behavior data generated by a Soar agent. Using behavior of hand-coded Soar agents to create new "clone" agents allows us to easily experiment with and evaluate our framework, as well as to compare the learned knowledge to the knowledge used in the original agents. 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.

Our implementation of the first mode of our learning by observation framework (Fig. 2), 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 (Srinivasan, 2003) that we have embedded to our system. We used Aleph in a setting where it implements inverse entailment (Muggleton, 1995) with an A* heuristic similar to Progol (Muggleton, 1995), but we modified it to benefit from the efficient mechanisms of the episodic database. We use a cost function that maximizes the covered positive examples, while minimizing the covered negative examples and the minimum number of literals that must be added to the current hypothesis to generate the output variables.

4.1. Learning from artificially created examples

In our initial experiments we evaluated the learnability of a decision concept using examples generated from a hand-coded rule representing that concept. At the end of learning, we compared the learned concept to the original hand-coded concept.

As a test case, we chose the problem of learning the selection condition of `go-to-door(Door)` operator when called as a suboperator of `get-item(Item)`. A correct hypothesis in this problem is depicted in Fig. 10. The goal is to learn to select a door such that it is on a path towards a room that contains an item that the agent wants to get.

4.1.1. Learning from correct positive and negative examples

In our first experiment, we evaluated our system using artificially created correct positive and negative examples generated from a correct hand-coded concept. In this experiment, we artificially generated examples of that concept as well as situation and annotation predicates to be used in background of learning. For each run of the experiment, we generated random map structures consisting of rooms, doors, items, and relations between them such as connectivity between two doors in neighboring rooms, or a path structure between doors, and rooms (similar to Fig. 3). Then, we generated random situations by choosing different rooms for the `current-room` sensor, and different items as the parameter of the parent goal annotation `get-item`. Finally, by testing the correct hand-coded concept (Fig. 10) on random situations, we generated correct positive and negative examples of the target concept.

For each run of the experiment, we generated a single map consisting of 6 rooms, 6 items randomly placed in these rooms. Each map is created by first randomly connecting rooms until there is a unique path between all rooms, and then by adding 3 or 12 extra connections so that there are multiple ways to navigate between rooms and the learning problem is harder.

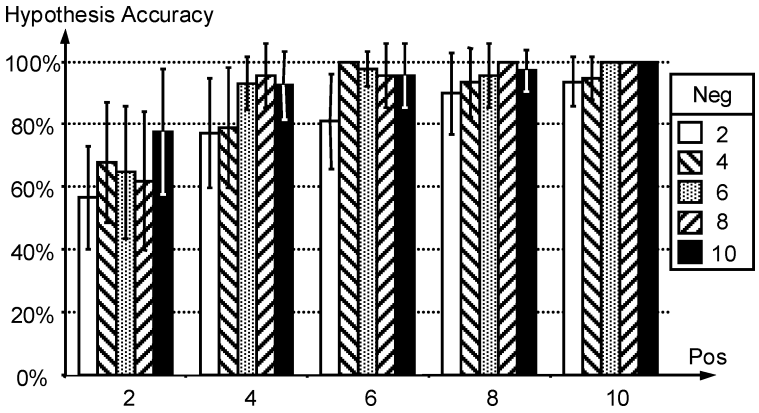

Fig. 11 Accuracy distribution of the learned hypotheses with correct positive and negative examples

(We want to show that we can learn arbitrary choices if both are equally good.) We considered 5 different sizes for positive and negative examples set sizes and for each of 25 combinations, we generated 20 training sets and ran the experiment 500 times in total. We completed each run when the learned hypothesis covered all positive and no negative examples.

We evaluated the learned concepts by comparing their predictions to the predictions of the hand-coded correct concept on 6 independently generated test maps. We measured the accuracy of the learned hypothesis (as compared to the correct hypothesis) at all possible situations on all test maps. For example, if in a situation (current-room, target item selection) the correct concept satisfy the doors $\{d_1, d_2\}$ indicating that either one of them are equally good selections, and if the learned hypothesis satisfy the doors $\{d_2, d_3\}$, we calculate the accuracy as 1/3, by dividing the size of the intersection of these sets to the size of their union. We calculated the accuracy of each learned hypothesis by taking the average of its accuracy over all situations on all test maps. Figure 11 depicts the distribution of the accuracy of the learned hypotheses over the number of examples used in the training. Each bar represent the average accuracy of the learned hypothesis over 20 experiments while the distance between the error markers represent one standard deviation of the accuracy. The accuracy may decrease due to insufficient number of negative or positive examples. We observe that the accuracy increases with the number of examples until finally the learned hypothesis is equivalent to the correct hypothesis in all 20 runs with sufficient number of examples.

4.1.2. Learning from only positive examples

In our second experiment, we evaluated our system on the same learning problem but this time we used only correct positive examples as the input. The learning system created negative examples heuristically from the positive examples using the scheme depicted in Fig. 7. This experiment provides a more realistic depiction of the first cycle of our framework where no negative examples are available.

Beside the negative example selection scheme, we used a similar methodology to the previous experiment to generate the data set and to evaluate the results, except this time, since the correct hypothesis may cover some heuristically generated negative examples and may not have a perfect coverage score, we restricted search to exclude hypotheses longer

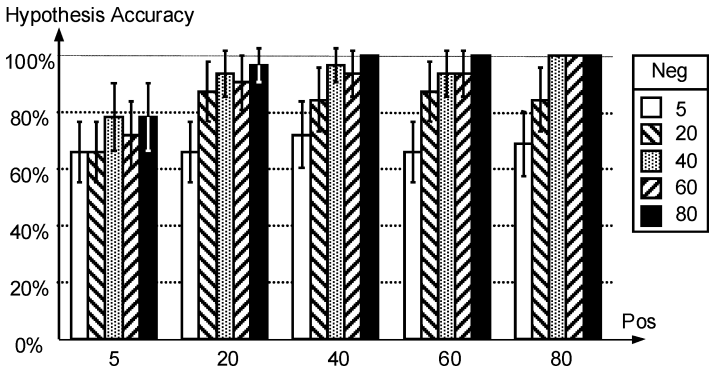

Fig. 12 Accuracy distribution of the hypothesis learned with correct positive and heuristic negative examples

than the correct concept and we terminated the search when the score of the considered hypothesis reaches the score of the correct concept on the training set. Since this experiment could potentially consider all possible hypotheses at that clause length and the durations can be long, we simplified the learning problem, by forcing the addition of `active-operator` literal to all rules; that is our system did not need to explicitly learn that checking the parent goal annotation is useful. Forcing the inclusion of this condition is consistent with hand-coded agents, where operator selections are usually conditional on the parent operator. Without this simplification, we got similar results on a few cases we tested but the experiments took considerably longer time.

We considered 5 different sizes for the positive and negative example sets and for each of 25 combinations we generated 15 random maps (3, 12, or 24 extra connections, 6 rooms, and 6 items) and ran the experiment 375 times in total.

The mean and the standard deviation of the learned concept accuracy are depicted in Fig. 12. In this experiment, we used more examples than the previous one because the heuristic negative example generation scheme can generate some incorrect negative examples. Again, as the number of examples increase, the accuracy of the learned concepts increase steadily reaching 100% accuracy in all experiments. Note that, 100% accuracy indicates equivalence to the correct concept not perfect coverage of the training examples.

4.2. Learning from agent program generated behavior

In our final 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. There are nodes on both side the doors and at the corners of the rooms (Fig. 13).

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 `go-to-room(Room)` by instantiating `Room` with the `room` where the item is in. It then uses `go-to-node-in-room(Node)` and `go-to-node-next-room(Node)` operators to go towards `Room`. The agent selects `go-to-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 `go-to-node-next-room` with a node on the other side of the door and moves towards it.

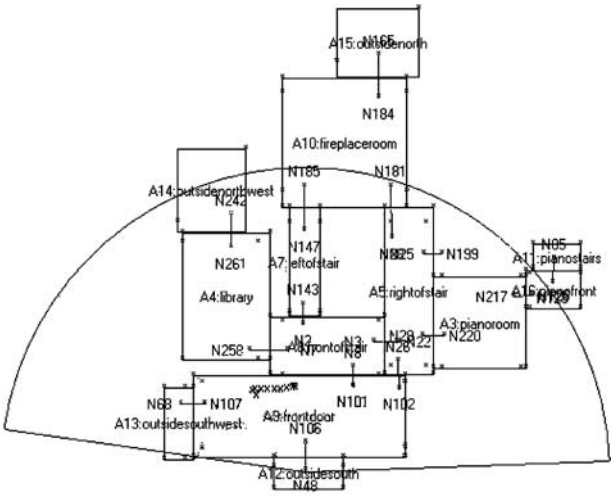

Fig. 13 A level map in Haunt domain

These two operators are used in a loop until the agent is in the target room and the parent operator `go-to-room` is retracted.

We have recorded several numerical sensors such as x-coordinate, object valued sensors that point to nearby objects, 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 predicates and a typical bottom clause (generated to clause length 6) has over 1000 literals.

In this experiment, our goal is to learn the selection and termination conditions of `go-to-node-in-room` in the context of a given `go-to-room` operator. We have collected 3 minutes of behavior trace of the Soar agent (∼30000 situations), where the agent has changed 57 rooms.

In this problem, the termination concept of `go-to-node-in-room` is very simple and is easily learned; it just checks that the agent is at the target node. We conducted systematic experiments to evaluate whether the selection conditions of `go-to-node-in-room` can be learned reliably using positive examples and heuristic negatives as depicted in Fig. 7. For 4 positive and 3 negative example set sizes, we ran the experiment 10 times. The distributions of learned hypotheses are depicted in Fig. 14. Since the continuous nature of our domain prohibits a complete accuracy analysis corresponding to what we conducted in the previous two experiments, we have qualitatively classified the learned hypothesis. Once again, when the number of examples is sufficiently high, the correct hypothesis (Fig. 15) is learned consistently.

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 a combination of positive examples and inaccurate heuristic negative examples may be sufficient to learn selection conditions, even in cases where the target behavior is not deterministic.

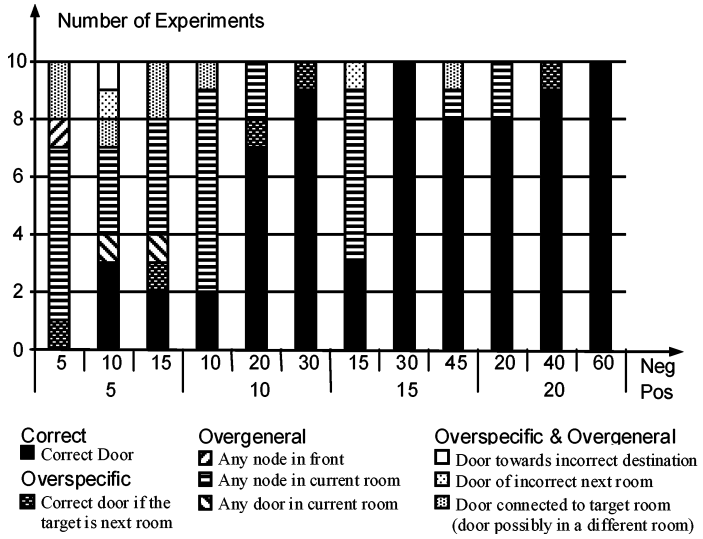

Fig. 14 The distribution of the learned concepts with real behavior trace data of a Soar agent

```
selection-condition(S, go-to-node-in-room(TargetNode) ) ←
          active-operator(S, go-to-room(TargetRoom) )    and
          current-room(S, agent, CurrentRoom )           and
          path(S, CurrentRoom, Path)                     and
          pathnode(S, Path, TargetNode)                  and
          destination(S, Path, TargetRoom)
```

Fig. 15 The correct hypothesis learned with real behavior trace data of a Soar agent

5. Related work

Khardon (1999) 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.

One alternative to our inductive approach to learning by observation is to use explanation-based learning (EBL), where learning is a result of explaining few examples using a deductive background knowledge theory (Mitchell et al., 1986; DeJong and Mooney, 1986; DeJong, 1993). For example, Segre's ARMS system (1993) used EBL on expert behavior traces to learn assembly plans for a simulated robot arm. Recently, learning by observation that utilizes an EBL-like technique was used with the general agent architecture ICARUS (Salomaki et al., 2005). This system represents the learned procedural knowledge in a skill hierarchy (similar to our operator hierarchy) built through learning. Although this system doesn't require expert goal annotations, it requires definitions of the goal conditions that hold when the task is terminated, a causal action model, and a hierarchical theory of concepts relevant for the task. Most EBL systems require background knowledge that can deductively explain expert behavior, whereas in our inductive approach background knowledge is optional and used to the extent it helps to find similarities between observed examples in the behavior.

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 (Benson and Nilsson, 1995) combines expert observations and experimentation to learn STRIPS like teleo-operators using ILP. OBSERVER (Wang, 1996) uses expert observations to learn planning operators in a rich representation (not framed in an ILP setting). Moyle (2003) describes an ILP system that learns theories in event calculus, while Otero describes an ILP system that learns effects in situation calculus (Otero, 2003). 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 (Džeroski et al., 2001) uses environmental reward to first learn utility of actions in a particular state and then compiles them to an action selection policy. Recently, Driessens and Džeroski (2002) combined expert behavior traces with the traces obtained from experimentation on the environment. Expert guidance helps their system to reach states that return reward more quickly. Unlike our framework, the actions the expert selects are not directly treated as correct decisions and the actions are not learned in the context of goals. Instead, the values of the actions are propagated back from the goal states that return reward. In complex domains with large state space and sparse reward, this strategy of justifying actions in terms of future rewards maybe less effective than learning by observation. 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 the expert but on the success in the environment.

6. Conclusions and future work

We have described a framework to learn procedural knowledge from structured behavior traces, hierarchical goal annotations parameterized with objects, and complex background knowledge. We decomposed the "learning an agent program" problem into 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, which was crucial in testing our system in a large domain. We have partially implemented the first mode of our framework, where the learning system uses only correct behavior data. We have conducted three experiments to evaluate the learnability of individual decision concepts. In the first experiment, we used a small data set of artificially created situations and demonstrated that learning converges to the correct hypothesis with few correct positive and negative examples. In the second experiment, we used a similar data set but this time instead of using correct negative examples, we generated heuristic negative examples from correct positive examples. Although the negative examples generated this way are inherently inaccurate, the learning again converged to the correct hypothesis with an increasing number of examples. In the third experiment, we used a large data set generated from the behavior of a hand-coded agent in a complex domain. We again used correct positive examples and inaccurate heuristic negative

examples and showed that the learning converges to the correct hypothesis with an increasing number of examples, even though only correct agent behavior is used.

The behavior dataset used in our last experiment provides a challenging new testbed for future ILP research. The search space is large, and feedback may be rare. Interested researchers may obtain this dataset by contacting the first author. In general, learning from structured data that changes over time is a rarely studied subject. Although we have used the episodic database to efficiently represent and access the input data, we treated the situations independently and used a generic ILP algorithm during learning. We believe that this kind of relational data has special properties that can be exploited to improve learning. For example, finding a generalization between two consecutive situations may be easier than finding a generalization between independent situations because in the former case, we know which objects correspond to each other.

A formal evaluation of our episodic database formalism is left for future work. We are currently extending this formalism so that it not only compactly represents behavior data, but also tests rules more efficiently by testing the rules on a range of situations at once.

We aim to implement the second execution cycle of our framework. We predict that the behavior data obtained this way will provide valuable examples and improve the effectiveness of learning. We also left the question of how the conflicting suggestions of different decision concepts are resolved for future research. We believe that a secondary learning step can address this issue.

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

References

Bain, M., & Sammut, C. (1999). A framework for behavioural cloning. In K. Furukawa, D. Michie, & S. Muggleton (Eds.), *Applied machine intelligence, machine intelligence,* 15, Oxford University Press.

Benson, S., & Nilsson, N. (1995). Inductive learning of reactive action models. In A. Prieditis, & S. Russell (Eds.), *Machine learning: Proceedings of the twelfth international conference* (pp. 47–54). San Francisco, CA: Morgan Kaufmann.

Camacho, R. (1998). Inducing models of human control skills. In *Machine Learning: ECML-98, 10th European Conference on Machine* (pp. 107–118). Germany: Springer.

DeJong, G. (ed.). (1993). *Investigating Explanation-Based Learning*. Boston: Kluwer.

DeJong, G., & Mooney, J. R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*(2), 145–176.

Driessens, K., & Dzeroski, S. (2002). Integrating experimentation and guidance in relational reinforcement learning. In C. Sammut, & C. Hoffmann (Eds.), *Proceedings of the nineteenth international conference on machine learning* (pp. 115–122). San Francisco, CA: Morgan Kaufmann.

Džeroski, S., Raedt, L. D., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*(1–2), 5–52.

Isaac, A., & Sammut, C. (2003). Goal-directed learning to fly. In T. Fawcett, & N. Mishra (Eds.), *Proceedings of the twentieth international conference*. Menlo Park, CA: AAAI Press.

Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P. G., & Koss, F. V. (1999). Automated intelligent pilots for combat flight simulation. *AI Magazine*, *20*(1), 27–42.

Khardon, R. (1999). Learning to take actions. *Machine Learning*, *35*(1), 57–90.

Könik, T., Pearson, D., & Laird, J. (2005). Learning through interactive behavior specifications. In G. D. W. Aha, & G. Tecuci (Eds.), *Mixed initiative problem-solving assistants: Papers from the 2005 fall symposium*, Tech. Report FS-05–07. Menlo Park, CA: AAAI Press.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: an architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Magerko, B., Laird, J. E., Assanie, M., Kerfoot, A., & Stokes, D. (2004). AI characters and directors for interactive computer games. In *Proceedings of the sixteenth innovative applications of artificial intelligence conference*. Menlo Park, CA: AAAI.

Michie, D., Bain, M., & Hayes-Michie, J. (1990). Cognitive models from subcognitive skills. In M. Grimble, J. McGhee, & P. Mowforth (Eds.), *Knowledge-based systems in industrial control*. Stevenage: Peter Peregrinus.

Michie, D., & Camacho, R. (1992). Building symbolic representations of intuitive real-time skills from performance data. In *Machine Intelligence 13 Workshop*.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: a unifying view. *Machine Learning*, *1*(1), 47–80.

Moyle, S. (2003). Using theory completion to learn a robot navigation control program. In S. Matwin, & C. Sammut (Eds.), *Inductive logic programming, 12th international conference, revised papers*, *Lecture notes in computer science, 2583* (pp. 182–197). Germany: Springer.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing*, *13*, 245–286.

Newell, A. (1990). *Unified theories of cognition*. Harvard Univ. Press.

Otero, R. P. (2003). Induction of the effects of actions by monotonic methods. In T. Horváth (Eds.), *Inductive logic programming, 13th international conference, lecture notes in computer science, 2835* (pp. 299–310). Germany: Springer.

Salomaki, B. Choi, D., Nejati, N., & Langley, P. (2005). Learning teleoreactive logic programs by observation. In G.D.W. Aha, & G. Tecuci (Eds.), *Mixed initiative problem-solving assistants: Papers from the 2005 fall symposium*, Tech. Report FS-05–07. Menlo Park, CA: AAAI.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. In D. Sleeman, & P. Edwards (Eds.), *Proceedings of the 9th international conference on machine learning* (pp. 385–393). Morgan Kaufmann.

Segre, A. M. (1993). ARMS: Acquiring robotic assembly plans. In G. DeJong (Ed.), *Investigating explanation-based learning*. Boston: Kluwer.

Srinivasan, A. (2003). The Aleph 5 Manual. http://web.comlab.ox.ac.uk/oucl/ research/areas/machlearn/ Aleph/.

Struyf, J., Ramon, J., & Blockeel, H. (2003). Compact representation of knowledge bases in ILP. In S. Matwin, & C. Sammut (Eds.), *Inductive logic programming, 12th international conference, revised papers, lecture notes in computer science, 2583*. Germany: Springer.

Šuc, D., & Bratko, I. (2000). Problem decomposition for behavioural cloning. In *11th European Conference on Machine Learning, Lecture Notes in Computer Science, 1810* (pp. 382–391). Germany: Springer.

Urbančič, T., & Bratko, I. (1994). Reconstructing human skill with machine learning. In A.G. Cohn, (Eds.), *Proceedings of the 11th european conference on artificial intelligence* (pp. 498–502). John Wiley and Sons.

van Lent, M., & Laird, J. (2001). Learning procedural knowledge through observation. In *Proceedings of the International Conference on Knowledge Capture* (pp. 179–186). New York: ACM Press.

van Lent, M. (2000). *Learning task-Performance knowledge through observation*, Ph.D. Thesis. Electrical Eng. and Computer Science Dept, University of Michigan.

Wang, X. (1996). Learning planning operators by observation and practice. Ph.D. Thesis (Tech. Report CMU-CS-96.154). Computer Science Department, Carnegie Mellon University.

Wray, R. E., Laird, J. E., Nuxoll, A., Stokes, D., & Kerfoot, A. (2004). Synthetic adversaries for urban combat training. In *Proceedings of the Sixteenth Innovative Applications of Artificial Intelligence Conference*. Menlo Park, CA: AAAI.