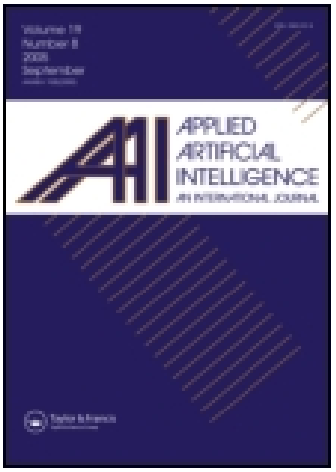

Applied Artificial Intelligence: An International Journal

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/uaai20

Real-Time Action Model Learning with Online Algorithm 3SG

Michal Čertický[a]

[a] Agent Technology Center, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

Published online: 08 Aug 2014.

To cite this article: Michal Čertický (2014) Real-Time Action Model Learning with Online Algorithm 3SG, Applied Artificial Intelligence: An International Journal, 28:7, 690-711, DOI: 10.1080/08839514.2014.927692

To link to this article: http://dx.doi.org/10.1080/08839514.2014.927692

Applied Artificial Intelligence, 28:690–711, 2014

ISSN: 0883-9514 print/1087-6545 online
DOI: 10.1080/08839514.2014.927692

Taylor & Francis
Taylor & Francis Group

REAL-TIME ACTION MODEL LEARNING WITH ONLINE ALGORITHM 3*SG*

Michal Čertický
Agent Technology Center, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

□ *An action model, as a logic-based representation of action's effects and preconditions, constitutes an essential requirement for planning and intelligent behavior. Writing these models by hand, especially in complex domains, is often a time-consuming and error-prone task. An alternative approach is to let the agents learn action models from their own observations. We introduce a novel action learning algorithm called* 3SG *(Simultaneous Specification, Simplification, and Generalization), analyze and prove some of its properties, and present the first experimental results (using real-world robots of the* SyRoTek *platform and simulated agents in action computer game* Unreal Tournament 2004*). Unlike the majority of available alternatives,* 3SG *produces probabilistic action models with conditional effects and deals with action failures, sensoric noise, and incomplete observations. The main difference, however, is that* 3SG *is an online algorithm, which means it is rather fast (polynomial in the size of the input) but potentially less precise.*

INTRODUCTION

Knowledge about domain dynamics, describing how certain actions affect the world, is essential for planning and intelligent goal-oriented behavior of both living and artificial agents. Such knowledge is, in artificial systems, referred to as an *action model*.

An action model can be understood as a double $\langle D, P \rangle$, where D is a representation of *domain dynamics* (*effects* and *preconditions* of every possible action) in any logic-based language, and P is a probability function defined over the elements of D. As a typical example of language used to represent D, we should mention the Stanford Research Institute Problem Solver (STRIPS) formalism (Fikes and Nilsson 1971) or more recent action language Planning Domain Definition Language (PDDL; McDermott et al. 1998).

Address correspondence to Michal Čertický, Stefanikova 40, Kosice, 04001, Slovakia. E-mail: certicky@agents.fel.cvut.cz

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/uaai

Typically, action models are handwritten by programmers or domain experts. In many situations however, we would like to be able to induce such models automatically, because writing them by hand is often a difficult, time-consuming, and error-prone task (especially in complex environments). In addition to that, every time we are confronted with new information, we need to do (often problematic) knowledge revisions and modifications.

By allowing our agent to learn action models automatically, we not only evade these problems, but also create some degree of environmental independence (we can deploy this agent into various environments, and let it learn local causal dependencies and consequences of its actions).

The process of automatic construction and subsequent improvement of action models, based on sensory observations, is called *action learning*.

We begin this article with a short overview of current action learning methods in "Background and Methods." After that, we introduce our novel technique, an online algorithm called 3*SG*, and follow it up by proof of its complexity and correctness in "Properties I: Correctness and Complexity." Then we present our pilot experiments in "Experiments." We used our algorithm for real-time action learning in two domains: an action computer game, *Unreal Tournament 2004* (Gemrot et al. 2009), and a real-world robotic platform, *SyRoTek* (Kulich et al. 2013). The article is concluded by a short collection of theorems that illustrate additional properties of 3*SG* in "Properties II: Learned Effects" and a draft of our ongoing and future research.

BACKGROUND AND METHODS

Recent action learning methods take various approaches and employ a wide variety of tools from different areas of artificial intelligence. First of all, we should mention the simultaneous localization and mapping algorithm called *SLAF* (Amir and Chang 2008), because the ideas behind it are related to our solution. *SLAF* uses agents' observations to construct a long propositional formula over time and subsequently interprets it using a satisfiability (SAT) solver. Another technique, in which learning is converted into a satisfiability problem (weighted Max-SAT in this case) and SAT solvers are used, is implemented in an Action-Relation Modeling System (*ARMS*) (Yang et al. 2007). As an example of a fully declarative solution, we should mention two mutually similar approaches based on logic programming paradigm ASP (Balduccini 2007) and its reactive extension (Certicky 2012). There are also several approaches that are not directly logic-based; for example, the action learning with a perceptron algorithm (Mourão, Petrick, and Steedman 2010) or the multi level greedy search over the space of possible action models (Zettlemoyer, Pasula, and Kaelblin 2005).

The advantage of the 3*SG* algorithm over these techniques is not only the fact that it produces *probabilistic* action models with *conditional effects* and deals with *action failures, sensorie noise*, and *incomplete information* (mentioned

methods have different subsets of these properties), but mainly that it is an *online algorithm*. This makes it fast enough to be used in real time. However, it also means that the precision of learned models cannot, in general, be guaranteed.

THE 3*SG* ALGORITHM

Preliminaries

To explain how the algorithm works, we will use a simple "toy domain" called Blocks World as a running example. This domain was discussed extensively (among others) in Nilsson (1982), Gupta and Nau (1992), Slaney and Thibaux (2001) and Russell and Norvig (2003).

Let us focus only on a very basic instance of the Blocks World domain, consisting of 3 blocks, *A, B, C*, which can be stacked into towers on the table. An agent can manipulate this domain by moving blocks from one position to another. There is only one kind of action, which is called *move,* and takes three parameters (B, P_1, P_2). The *move*(B, P_1, P_2) action moves a block B from position P_2 to position P_2 (P_1 and P_2 being either another block, or the table).

Throughout this article, we will use some intuitions borrowed from work on planning and logic programming (Baral 2003; Eiter et al. 2000): An *atom* is the form $p(t_1,\ldots,t_n)$, where p is a predicate symbol, and $t_1,\ldots,t_n$ are either *constants* or *variables*. If each of the t_is is a constant, we say that the atom is *ground*. *Fluent literal* (often called simply *fluent)* is either a ground atom, or a ground atom preceded by the negation symbol "¬". We say that fluents appear either in *positive* or in *negative* form. Fluents f and $\neg f$ are considered mutually complementary, and the following notation is used: $\overline{f} = \neg f$ and $\overline{\neg f} = f$. We will denote the set of all the fluents of our domain by $\mathcal{F}$. Every fluent is used to describe a certain property or our domain, which can change over time. Specific configurations of a domain, called *world states,* can therefore be specified by certain sets of fluents.

Figure 1 depicts two different world states of our example domain. For instance, the world state (a) is formalized as the following set of fluents: $s = \{on(a,table), on(b,c), on(c,table), \neg on(a,b), \neg on(a,c), \neg on(b,table), \neg on(b,a), \neg on(c,a),$

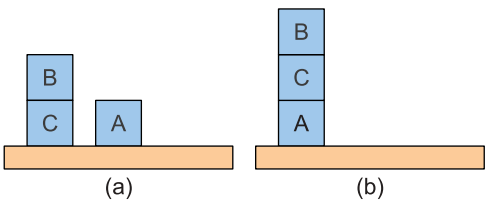

FIGURE 1 Two different world states of the Blocks World domain.

$\neg on(c,b), \neg blocked(a), \neg blocked(b), \neg blocked(c), \neg blocked(table)\}$. Note that this set is complete and consistent in the sense that out of each pair of mutually complementary fluents, it contains exactly one.

Definition 1. (World State). *We say that the set of fluents $s \subset \mathcal{F}$ is a world state, iff the following conditions hold:*

- $\forall f \in \mathcal{F} : f \in s \vee \overline{f} \in s.$
- $\forall f \in \mathcal{F} : f \in s \Rightarrow \overline{f} \notin s.$

We will denote the set of all the possible world states by $\mathcal{S}$. Note that the size of $\mathcal{S}$ can be expressed as $|\mathcal{S}| = 2^{|\mathcal{F}|/2}$.

Observations are also expressed by sets of fluents. However, unlike world states, they do not need to be complete or consistent. Any nonempty subset $o \subseteq \mathcal{F}$ can be considered an observation. In fact, more complex domains are typically only partially observable.

The last notion we need to define is action. *Action* (sometimes called "*action instance*") is the form $a(p_1, \ldots, p_n)$, where a is an action name and $p_1, \ldots, p_n$ are constant parameters. The set of all the possible actions is denoted by $\mathcal{A}$. An example of action from our Blocks World domain is "*move*(*b, c, a*)" or "*move*(*b, c, table*)."

Representation of Domain Dynamics

Now that we have all the necessary formal definitions, we can start explaining the 3*SG* algorithm. A considerable part of our solution lies in the selection of an appropriate representation structure for our domain dynamics *D*. It needs to be as compact as possible to allow for real-time learning in the presence of frequent and elaborate observations.

Classic representation structure, used, for example, in Amir and Chang (2008) and Amir and Russell (2003), and similar to structure used in Eiter et al. (2010), is called *transition relation: TR* $\subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Each of its elements expresses the relation between two world states and one action. Transition relation has the highest expressive power, but it is also very robust in terms of space complexity. A slightly more compact structure is its modification, which we call the *effect relation: ER* $\subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{F}$. Its space complexity tends to be lower,[1] because its elements explicitly represent only the relation between one world state, action, and a single fluent.

The representation structure used by algorithm 3*SG* is ideologically based on these two, and we call it the *effect formula*. However, the effect formula (EF) is not a relation—it is merely a finite set[2] of propositional atoms of two types:

1. a^f means, *"action* **a** *causes fluent* **f***"*
   Example: move(b,c,a)$^{on(b,a)}$
2. a^f_c means "**c** must hold in order for **a** to cause **f**"
   (in other words, **c** is a condition of a^f)
   Example: move $(b, c, a)^{on(b,a)}_{\neg blocked(a)}$

The advantage of *EF* is its compactness, which makes it ideal for real-time learning. A disadvantage is that, unlike transition relation, it cannot represent disjunctive effect conditions. Throughout the rest of this article, we will be working with action models in the $\langle EF, P\rangle$ form.

Note 1. Learning effects with disjunctive conditions is a considerably harder problem, and the majority of action learning methods (including ours) don't solve it. Therefore, we simply do not need the ability to represent such conditions, and the expressive power of EF will be sufficient for us.

Probability Function and the Input

The input of the 3*SG* algorithm is a triple (o, a, o'), where o' is the most recent observation, o is the observation from a previous time step, and a is the action executed between them. This triple is also called an *example*. Every such example can (but does not have to) be considered positive or negative with respect to some element of *EF*.

Numbers of positive and negative examples with respect to an element $i \in EF$ $(\text{pos}(i)$ and $\text{neg}(i))$ are based on the semantics of *EF* and are computed differently for both types of atoms. First, let us denote the set of those fluents that changed their value within our example by $\Delta(o, o')$ (in other words $\Delta(o, o') = \{f | \overline{f} \in o \wedge f \in o'\}$). Then,

- $\text{pos}(a^f) = |\{(o, a, o') | f \in \Delta(o, o')\}|$.
  For instance, an example from our Blocks World domain would be considered positive with respect to $move(a, c, b)^{on(a,b)}$ if $\neg on(a, b) \in o$ and $on(a, b) \in o'$. In other words, $on(a, b)$ started to hold just after we moved a to b.
- $\text{neg}(a^f) = |\{(o, a, o') | \overline{f} \in o'\}|$.
  Another example would be negative with respect to $move(a, c, b)^{on(a,b)}$ if $\neg on(a, b) \in o'$. After the action *move,* block a was not on b.
- $\text{pos}(a^f_c) = |\{(o, a, o') | c \in o \wedge f \in \Delta(o, o')\}|$
  An example would be positive with respect to $move(a, c, b)^{on(a,b)}_{\neg blocked(b)}$ if it was also positive with respect to $move(a, c, b)^{on(a,b)}$ and at the same

time $\neg blocked(b) \in o$. Expected effect $on(a,b)$ happened while b was not blocked.

- $\text{neg}\left(a_c^f\right) = \left|\left\{\left(o, a, o'\right) | \overline{c} \in o \wedge f \in \Delta\left(o, o'\right)\right\}\right|$
  Similarly, an example is positive with respect to $move(a, c, b)^{on(a,b)}_{\neg blocked(b)}$ if it is also positive with respect to $move(a, c, b)^{on(a,b)}$ but at the same time is $blocked(b) \in o$. Expected effect $on(a,b)$ happened even though b was blocked.

Actual numbers of positive and negative examples will be important to us because we need them to define the probability function P for individual elements of EF.

$$\forall i \in EF:$$

$$P(i) = \begin{cases} 0 & \text{if } \text{pos}(i) + \text{neg}(i) < \text{min}Ex \\ \frac{\text{pos}(i)}{\text{pos}(i)+\text{neg}(i)} & \text{if } \text{pos}(i) + \text{neg}(i) \geq \text{min}Ex \end{cases},$$

where minEx is a constant parameter with the following meaning: If the sum of positive and negative examples with respect to some element equals k, we say that this element is *supported* by k examples. We want every element of EF to be supported by at least minEx examples (min$Ex \in \mathbb{N}$), before we assign it a nonzero probability.

We also say that action model $\langle EF, P\rangle$ *covers* an example $\left(o, a, o'\right)$, iff $\forall f \in \Delta\left(o, o'\right) : a^f \in EF$. Similarly, an action model is *in conflict* with example $\left(o, a, o'\right)$, iff $\exists a^f \in EF : \left(\forall a_c^f \in EF : c \in o\right) \wedge \left(\overline{f} \in o'\right)$. Finally, iff $\exists a^f \in EF : \left(\forall a_c^f \in EF : c \in o\right) \wedge \left(f \in o'\right)$, we say that EF *predicts* fluent f in example $\left(o, a, o'\right)$.

Algorithm

Now that we have explained our representation structure, input, and probability function, we can proceed to the algorithm itself (we will be describing the pseudocode from Figure 2).

The acronym *"3SG"* stands for *"Simultaneous Specification, Simplification, and Generalization"* and it says a lot about the algorithm's structure.[3] 3SG is composed of three corresponding parts:

1. By **generalization** we understand the addition of a^f type atoms into EF. That is because their addition enlarges the set of examples covered by EF. Generalization is, in our algorithm, taken care of by the first *Foreach* cycle (lines 0–10), which adds a hypothesis (atom a^f) about a causing f,

```
INPUT:  Newest example (o,a,o'), where a is executed action, and o,o' are observations.
        Current action model ⟨EF,P⟩, which we will modify (possibly empty).
OUTPUT: Modified action model ⟨EF,P⟩.

00 Foreach f ∈ Δ(o,o') do {
01     // generalization of EF
02     If a^f ∉ EF then Add a^f to EF.
03     Else {
04         Modify P(a^f) by incrementing pos(a^f).
05         Foreach c ∈ o do {
06             if a^f_c ∈ EF then Modify P(a^f_c) by incrementing pos(a^f_c).
07             if a^f_c̄ ∈ EF then Modify P(a^f_c̄) by incrementing neg(a^f_c̄).
08         }
09     }
10 }
11
12 Foreach f such that f̄ ∈ o' do {
13     If a^f ∈ EF {
14         Modify P(a^f) by incrementing neg(a^f).
15         // specification of EF
16         Foreach c such that c̄ ∈ o do: if a^f_c ∉ EF then Add a^f_c to EF.
17     }
18 }
19
20 // simplification of EF
21 Foreach a^f_c ∈ EF older than memoryLength do {
22     If P(a^f_c) < minP then Delete a^f_c from EF.
23 }
24 Foreach a^f ∈ EF older than memoryLength do {
25     If ( P(a^f) < minP and there is no a^f_c in EF) or ( pos(a^f) + neg(a^f) < minEx ) {
26         Delete a^f from EF.
27     }
28 }
```

FIGURE 2 Pseudocode of 3*SG* algorithm.

after we observe that $\overline{f}$ changed to f. If such an element is already in *EF*, then the algorithm modifies its probability, along with all of its conditions. *Imagine for example, that our EF was empty when we observed the world state (a) from Figure 1, followed by the action* $a = move(b, c, a)$. *In other words,* o *is* $\{on(b, c), \neg on(b, a), \neg blocked(a), blocked(c), \ldots\}$, *and the following observation* o' *is* $\{\neg on(b, c), on(b, a), blocked(a), \neg blocked(c), \ldots\}$. *Fluents that changed their value are therefore* $\Delta = \{\neg on(b, c), on(b, a), blocked(a), \neg blocked(c)\}$. *We start with the first of them and add the "*$move(b, c, a)^{\neg on(b,c)}$*" hypothesis into EF.*

2. **Specification** is, however, an addition of a^f_c type atoms, which is taken care of by the second *Foreach* cycle (lines 12–18). We call it specification because new a^f_c elements restrict the set of examples covered by *EF* by imposing new conditions on the applicability of previously learnt a^f

effects. This cycle iterates over all of our a^f type atoms (with respect to which our current example is negative), lowers their probability, and generates several hypotheses about effect a^f failing because of the existence of some condition a^f_c such that c does not hold right now.

Now imagine that we are in the world state (b), and we try to execute the same action move(b,c,a). Because this is not possible (a is blocked), nothing will happen $\left(o' = o = \{blocked\,(a)\,, on\,(b, c)\,, \neg on\,(c, table)\,, \ldots\}\right)$. *Therefore, this is a negative example with respect to our previous hypothesis "*$move\,(b, c, a)^{\neg on(b,c)}$*." To explain this, we assume that our hypothesis has some conditions that are not met in this particular example. Therefore, we take everything that does not hold in o and use it to create a condition:*

$$move\,(b, c, a)^{\neg on(b,c)}_{\neg blocked(a)}\,,\ move\,(b, c, a)^{\neg on(b,c)}_{\neg on(b,c)}\,,\ move\,(b, c, a)^{\neg on(b,c)}_{on(c,table)}\ldots$$

We see that we have added the correct condition ("a must not be blocked"), and a couple of wrong ones. They will, however, be eventually deleted in the "Simplification" part.

3. **Simplification** of *EF* is, simply put, *forgetting* of those elements that have not been validated enough (their probability is too low) during some limited time period (specified by a constant *memory Length).* This is taken care of by the last two cycles (lines 21–28).

PROPERTIES I: CORRECTNESS AND COMPLEXITY

Before we can discuss the properties of 3*SG*, we need to understand that it is an *online* algorithm. In the context of action learning, this means that, instead of the whole history of observations, we always get only one (newest) example on the input. This example is used for irreversible modification of our action model. The drawback of online algorithms, compared with their offline counterparts, is that their final result depends on the order of received inputs, which makes it potentially less precise. Online algorithms process considerably smaller inputs, which makes them significantly faster and, therefore, more suitable for real-time learning.

Dependency on the ordering of inputs leads us to the question of the algorithm's correctness. In general, we declare an algorithm correct if it returns a correct result in finite time for any input. The definition of *correct result* will, in our case, have to be quite loose for the following reasons:

1. Online algorithms such as 3*SG* do not have access to a specific future or previous inputs.
2. Even if they did, we could not guarantee the consistency of new input with the older ones because of sensoric noise and possible action failures.

As a correct result, we will, therefore, consider such an action model that covers new examples and at the same time holds that if some new conflict with an old example occurred, it has to be caused either by *eliminating conflict* with the new example, or by *covering* it.

Theorem 1. (Correctness). *3SG algorithm without forgetting* $\left(\textit{memory Length} = \infty\right)$ *is correct according to above-mentioned definition.*

Proof. In order to declare correctness, we must prove the following three statements:

i. *3SG algorithm will terminate for any input.* This is a direct result of the algorithm's structure—first two cycles (lines 0–18 in Figure 2) always iterate over a finite set of observations (either o, or o'), whereas they don't modify this set at all. The remaining two cycles (lines 21–28) then iterate over the *EF* set, although they can only delete elements from it (they don't add anything new). Note that this condition would be satisfied even if the forgetting was allowed.

ii. *Resulting action model always covers new example.* According to the aforementioned definition of covering, for every $f \in \Delta\left(o, o'\right)$, we need to have an a^f atom in *EF*. This is taken care of by the first part of the algorithm, which adds a^f into *EF* (line number 2). If the forgetting is forbidden, no other part of the algorithm can delete this atom.

iii. *If a new conflict with an older example occurred, it is caused either by eliminating the conflict with the new example, or by covering it.* This condition is a trivial result of the fact that (without forgetting), the *EF* is modified only by addition of a^f on line 2 (in order to cover the new example), or by addition of new conditions a_c^f on line 16 if the new example is negative with respect to the corresponding a^f. According to the definition of conflict (explained previously), addition of such a^f conditions is such that $\overline{c} \in o$ removes the conflict with our example (because $\overline{c}$ and c cannot be in o at the same time).

Theorem 2. (Complexity). *Let n denote the cardinality of input EF and m the cardinality of the larger of two input observations o, o′. Worst-case time complexity of the 3SG algorithm is a polynomial in the size of the input. More precisely, if the EF is implemented by a B-Tree, the complexity is* $\mathcal{O}\left(\left(m^2 + n^2\right) \log n\right)$.

Proof. First of all, we will calculate the complexity of all four *Foreach* cycles within our algorithm. Once again, we will be referring to the pseudocode from Figure 2. Operations that are not mentioned in this proof run in constant time and can be ignored.

i. In the first cycle, we iterate over the set $\Delta(o, o')$ (line 0), which has m elements in the worst case. In every iteration, we need to find out if $a^f \in EF$ (lines 2 and 3). Finding an element in a B-tree has a time complexity of $\mathcal{O}(\log n)$ (Bayer and McCreight 1972). This *if*-condition gives us two possible outcomes, although computationally more complex is the case where $a^f \in EF$. There we have a nested a *Foreach* loop (line 5), which again iterates over, at most, m elements and for each of them it needs to check the existence of 2 different elements in EF (lines 6 and 7). This check gives us the complexity of $2 \cdot \mathcal{O}(\log n)$. Overall complexity of the first cycle is, therefore, in the worst case: $m \cdot m \cdot (2 \cdot \mathcal{O}(\log n)) = \mathcal{O}(m^2 \cdot \log n)$.

ii. In the second cycle, we iterate over the set of at most m elements (line 12) and in every iteration we check for the existence of a certain element in EF (line 13) with the complexity of $\mathcal{O}(\log n)$. The nested loop on line 16 then, again, at most m times checks for the existence of an element in EF and adds it if it's not there. Because the complexity of insertion into a B-Tree is also $\mathcal{O}(\log n)$ (Bayer and McCreight 1972), this whole cycle gives us the worst-case complexity of:

$$m \cdot ((\log n) + m \cdot 2(\log n)) = \mathcal{O}(m^2 \cdot \log n).$$

iii. In the third cycle, our algorithm deletes (forgets) some of the old atoms from EF. In order to know which atoms to delete, we need to test all n of them and, in the worst case, delete all of them (line 22). Because the deletion from B-Tree runs also in $\mathcal{O}(\log n)$ (Bayer and McCreight 1972), this cycle has an overall complexity of $\mathcal{O}(n \cdot \log n)$.

iv. The last cycle iterates once again over all n elements of EF (line 24), but here we need to check the nonexistence of relevant a_c^f atoms (line 25) for potentially all of them. This requires another iteration over all the n elements of EF. Together with the deletion of a^f we get: $n \cdot n \cdot \mathcal{O}(\log n) = \mathcal{O}(n^2 \cdot \log n)$.

Now, if we combine the complexities of all four cycles (running one after another), we get the overall complexity of the 3*SG* algorithm:

$$\begin{aligned}
&\mathcal{O}(m^2 \cdot \log n) + \mathcal{O}(m^2 \cdot \log n) + \mathcal{O}(n \cdot \log n) + \mathcal{O}(n^2 \cdot \log n) \\
&= \mathcal{O}((m^2 + m^2 + n + n^2) \log n) \\
&= \mathcal{O}((m^2 + n^2) \log n)
\end{aligned}$$

Note 2. *Implementing the* EF *as a B-Tree is not entirely optimal for our problem. Notice that in last cycle we needed to iterate over the whole* EF *set in order to find out what needs to be deleted. If we had our* EF *structured better, we could accelerate this process. The first thing that comes to mind is remembering the pointers between* a^f *atoms and all of their* a_c^f *conditions.*

EXPERIMENTS

We have conducted two experiments in different domains. The first domain was quite demanding in terms of computational complexity and we were able to examine the performance of 3*SG* in complex environments with many actions per minute. The second experiment was conducted in a real-world environment with a considerable amount of sensoric noise and a number of action failures.

Unreal Tournament 2004

The complexity of our algorithm (Theorem 2) suggests that the *size* of our *action model representation* will play an important role in the overall performance. Therefore, in the first part of this experiment we will measure the size of *EF*. In addition to that, we will be interested in the actual running speed of 3*SG* in practice, and of course also the quality of induced models.

As a domain for the first experiment, we have chosen the action FPS[4] computer game *Unreal Tournament 2004* (see Figure 3), which was accessed via the Java framework Pogamut (Gemrot et al. 2009). During the game, our agent was learning the effects and preconditions of all the observed actions

FIGURE 3 In-game screenshot from the UT2004 experiment. In the right part of the screen, we can see a part of learned action model (converted to PDDL for better readability).

such as *shoot, move, changeWeapon, takeItem,* and so on. These were executed either by our agent or by other agents (opponents).

The agent called the 3*SG* algorithm once after every observed action, which was approximately 7.432 times every second. The average size of his observations (o and o') was 70 fluents, which means that he processed approximately 140 fluents on every call. The following results were gathered during a 48.73-minute-long "deathmatch" game.

Size of the Action Model

During this experiment, the agent has observed and processed 21,733 examples, which corresponded to 7632 time steps (multiple actions could be observed in a single time step). The size of *EF* determines the size of the entire action model, because the probability function P is defined implicitly by a single equation. Our constants were set to the following values: $\min P = 0.9$, $\min Ex = 3$, $memoryLength = 50$. This means that we were deleting (forgetting) all those elements from *EF* that were not supported by at least 3 examples, or if their probability was lower than 90% after 50 time steps (approximately 19.54 seconds). The chart in Figure 4 depicts the size of *EF* during the whole 48.73 minute game. We can see that, thanks to forgetting, the size of our action model stays more or less on the constant level. If the forgetting was disabled, this size would grow uncontrollably.

Running Speed

In order to measure the actual running speed, we have created a log of consequent examples observed by an agent during the game, and only after that did we repeatedly call 3*SG* with each of these recorded examples on input. The algorithm was implemented in Python and our action model was stored in the MySQL database. The experiment was conducted under a Linux OS, on a system with Intel(R) Core(TM) i5 3.33GHz processor and

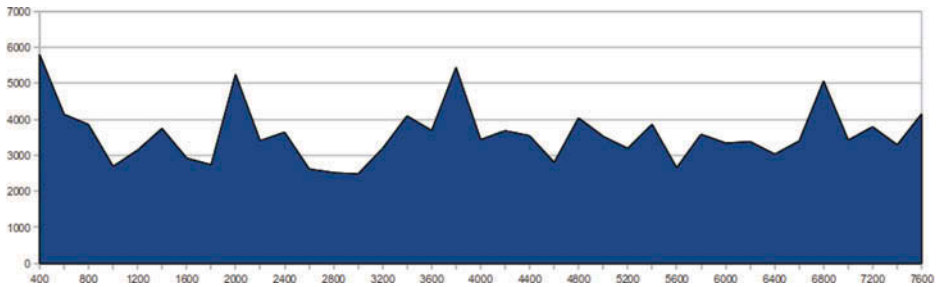

FIGURE 4 Development of *EF* size (*y*-axis) over time.

4GB RAM. Processing all 21,733 examples recorded during the 48.73-minute game took only 12 minutes and 10.138 seconds. This means that the 3*SG* algorithm can be used for action learning in real time, in domains at least as complex as that of the *UT2004* game.

Quality of Induced Models & Learning Curve

As we mentioned before, the action model serves as a representation of domain dynamics. In other words, it should accurately describe the changes happening within our domain, and consequently, we should be able to use this model to predict those changes. To evaluate the quality of an induced action model, we can therefore try to analyze its ability to predict actual changes happening in the domain during our experiments.

This is typically done by computing the *precision* and *recall* of the model, and their weighted harmonic mean called the *F-measure* (Rijsbergen 1979; Lewis and Catlett 1994; Makhoul et al. 1999).

In our case, the precision of the action model is defined as the ratio of predicted fluent changes that were consistent with the observations. Recall, on the other hand, is the ratio of observed fluent changes that we were able to predict. In other words:

- Let $A(f)$ be the number of observed examples $(o,\ a,\ o')$, where $f \in \Delta(o, o') \wedge EF$ predicts f (see subsection 3.1).
- Let $B(f)$ be the number of observed examples, where $f \in \Delta(o, o')$, but EF does not predict f.
- And finally, let $C(f)$ be the number of observed examples, where EF predicts f, but $\overline{f} \in o'$.

The precision and recall metrics with respect to a certain fluent f are then defined as

$$P(f) = \frac{A(f)}{A(f)+C(f)}, \quad \text{and} \quad R(f) = \frac{A(f)}{A(f)+B(f)}.$$

Now, if we compute the arithmetic mean of precision and recall with respect to every $f \in \mathcal{F}$ and denote them by P and R, the F-measure of our action model EF can be defined as:

$$F_\beta = (1+\beta) \cdot \frac{P \cdot R}{\beta^2 \cdot P + R}.$$

The nonnegative weight constant β here allows us to assign higher significance either to precision, or to recall. In our case, we considered the precision to be slightly more important, so we assigned β the value of 0.5.[5]

Notice that this performance metric combines both precision and recall in such a way that it strongly penalizes the models with either of them being

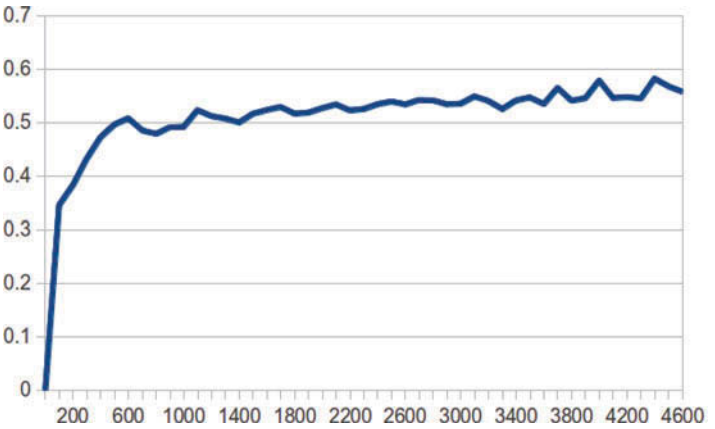

FIGURE 5 Development of the F-measure (*y*-axis) over time.

too low. This ensures that models need to be both accurate and descriptive enough in order to be considered good.

Now that we are able to evaluate the quality of the action model, we can take a closer look at the learning process. During the quality analysis, our 21,733 examples were divided into two disjoint sets: the *training set* and the *test set*; the later contained approximately 15% of all the examples (chosen randomly, using uniform distribution). In the learning process, we used only the examples from the training set. After every time step, we computed the F-measure of our model based on its ability to predict the changes occurring in the test set.

The learning curve in Figure 5 depicts the development of the quality of our action model (expressed by the F-measure) during our experiment. Notice the occasional slight decreases of the F-measure value. They are caused by the online nature of the 3*SG* algorithm. However, it is apparent that the quality is improving in the long term. Also note the rapid quality increase during the first minutes of the experiment. See Figure 6 for a more detailed picture of the first 500 time steps. It looks as though the learning process is fastest during the first 80 time steps (less than a minute), but continues during the whole experiment.

Resulting Models

Let us now take a closer look at our resulting action model. To enhance the readability, we have translated the *EF* into the planning language PDDL. This translation itself is quite straightforward and uninteresting. It is important because during the translation, all the atoms from *EF* whose probabilities were lower than min$P$ were ignored. Also, to simplify our PDDL representation even further, we

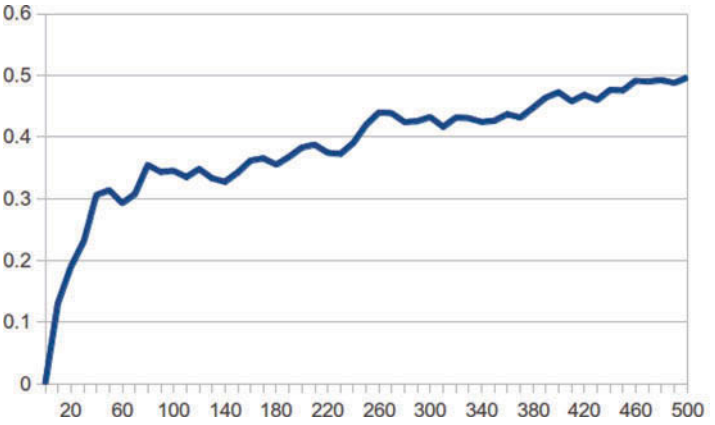

FIGURE 6 Development of the F-measure (*y*-axis) during first 500 time steps.

- merged the effect groups, where it was possible, by introduction of variables and
- converted this condition into a precondition of the whole action if all the effects of an action had one common condition.

This way we acquired a simplified, human-readable PDDL representation of our action model.

The first example depicted in Figure 7 is the action called *changeWeapon*(*A, B, C*). This has been learned exactly according to our intuitions: *Agent A can change the equipped weapon from B to C only if* $B \neq C$. *When he does this, C will be equipped, and B will no longer be equipped.* The second action, *move*(*A, B*), contains more learned effects (to keep things brief, we present four of them). The first two effects are not surprising: after the *move*(*A, B*) action, *agent A will be on position B and will not be on any other adjacent position Z.* The last two effects are less obvious, but they have been learned correctly based on observed examples. Specifically, the third effect of the *move* action states that *if agent A moves to a position called playerstart*25, *he will die.* The agent learned this, because almost every time it arrived there, something killed it. Similarly, it learned that moving to a position *playerstart*20 causes its health to get to maximum value. This was probably learned because there was a Medkit item at that position. An interesting fact is that the first four (intuitive) effects were learned very quickly (they were in *EF* during the first minute of gameplay); the less-obvious effects took our agent more time to learn (2.5 and 16 minutes, respectively).

Robotic Platform SyRoTek

Our second experiment was conducted using the real-world robots of the *SyRoTek* platform (Kulich et al. 2013). The *SyRoTek* system (see Figure 8)

```
(:action changeWeapon
         :parameters (?a ?b ?c)
         :precondition (and (differentWeapon ?b ?c) (differentWeapon ?c ?b))
         :effect (and
                  (equippedWeapon ?a ?c)
                  (not (equippedWeapon ?a ?b))
         )
)
(:action move
         :parameters (?a ?b)
         :effect (and
                  (onPosition ?a ?b)
                  (when (and (edge ?b ?z) (edge ?z ?b))
                                   (not (onPosition ?a ?z)))
                  (when (= ?b playerstart25) (dead ?a))
                  (when (= ?b playerstart20) (health ?a maximum))
                  … etc.
         )
```

FIGURE 7 Example of learned actions (converted to PDDL planning language).

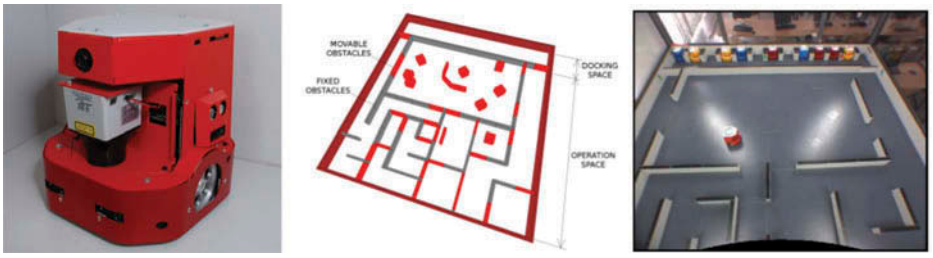

FIGURE 8 *SyRoTek* robot, map of the arena, and a photo taken during our experiment. The size of the arena is 350 x 380 cm.

allows its users to remotely control multiple mobile robots with a variety of sensors in the environment, with customizable obstacles.

During this experiment, we used only a single robot with a laser range-finder sensor and a camera-based localization system. All the other sensors were ignored. Also, the agent could execute only three kinds of actions: *move forward, turn left,* or *turn right*. It chose its actions randomly and always executed only one action at a time. Because the agent acted in a real-world environment, we experienced a considerable amount of sensoric noise and a number of action failures. However, this experiment was less computationally intensive.

Continuous data extracted from the sensors were discretized into fluents. The agent perceived exactly 56 fluents per observation and called the 3*SG* algorithm after every action—approximately once every 8 seconds. The experiment was terminated after 1250 examples (which took approximately 166 minutes).[6] Our constants were set to the following values: $\min P = 0.65$, $\min Ex = 3$, $memoryLength = 150$. In other words, we deleted an element of *EF* if it was not supported by at least three examples or if its probability was lower than 65 % after 150 time steps (approximately 20 minutes).

Quality of Induced Models & Learning Curve

Due to a relatively small number of examples and the length of time between individual actions, the algorithm's running speed and the size of *EF* were not an issue. More interesting was the actual development of quality of our learned model. 1250 examples were first divided into a training set consisting of 1000 examples and a test set of 250 randomly chosen examples (with uniform distribution).

As our quality metrics, we once again chose the F-measure (as defined in previously). The learning curve (Figure 9) is similar to that from the previous experiment. We can see the rapid quality increase during the first 80 time steps. There are also several slight decreases (caused by the online nature of the 3*SG* algorithm), but an overall long-term trend is increasing. An interesting observation is of the periods when the quality of our model was almost constant (for example, during time steps 550–690). During these periods, the agent was stuck between the obstacles (recall that its actions were random, so this could happen quite easily).

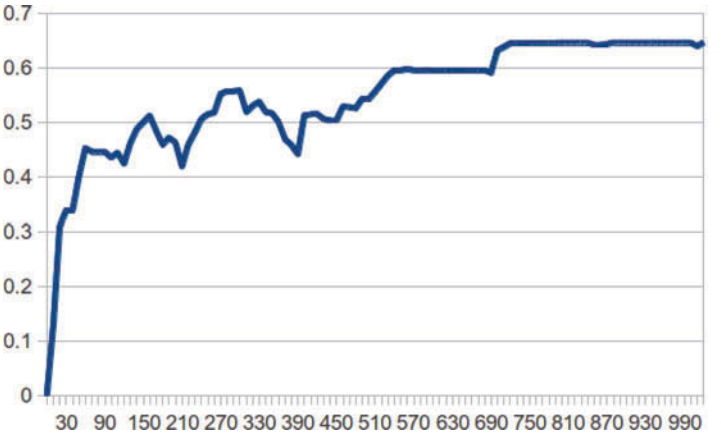

FIGURE 9 Development of the F-measure (*y*-axis) over time.

Resulting Models

Once again, we have translated our induced action model into PDDL for better readability.

What you see in Figure 10 is a fragment of the action model from the middle of the learning process. The first depicted action, *move*(*A*, *B*), simply tries to move the agent forward (from position *A* to *B*). We can see that our agent was able to learn its first two effects just as expected: *Moving from A to B causes us to be on position B, and no longer on A*. The third and fourth effects were temporary. They were added by the 3*SG* at a certain time step and disproved and deleted later. Notice that the fourth effect (*orientation*(*west*)) is a result of sensoric noise or an action failure (because the *move* action should not typically change the agent's orientation).

The second action, called *turnLeftFrom*(*A*), also had two intuitive and a couple of temporary effects. The first effect simply says that *after turning left from direction A, we will no longer be facing A*. The second effect means that *turning left from A causes us to face Y, if A is in a clockwise direction from Y*.

```
(:action move
        :parameters (?a ?b)
        :effect (and
                (position ?b)
                (not (position ?a))
                (blocked 27x41)
                (orientation west)
        )
)
(:action turnLeftFrom
        :parameters (?a)
        :effect (and
                (not (orientation ?a))
                (when (clockwise ?y ?a) (orientation ?y))
                (when (= ?a west) (front me 27x41))
                (when (= ?a west) (not (back me 28x38)))
                ... etc.
        )
)
```

FIGURE 10 Example of learned actions from the middle of the learning process (converted to PDDL planning language).

We should point out the fact that we used quite a long *memoryLength* (150 time steps is a relatively long period within our 1250-step experiment). This means that temporary effects are kept in the action model longer (they are deleted later, even if they have low probability).

PROPERTIES II: LEARNED EFFECTS

In this section we will take a closer look at some of the miscellaneous properties of the 3*SG* algorithm, which may be less important but can provide deeper insight into its behavior. The following collection of theorems will describe the conditions under which we learn an action's effects. Let us therefore start with a formal definition of this concept.

Definition 2. *Learned effect* will be understood as a double $E = \left\langle a^f, C = \{a^f_{c0}, a^f_{c1}, \ldots, a^f_{ci}\}\right\rangle$, where C is a set of conditions of a^f. We say that *we have learned an effect E* iff the following holds with respect to our action model $\langle EF, P\rangle$*:*

i. $a^f \in EF$.
ii. $\forall a^f_c \in C : \left(a^f_c \in EF\right) \wedge \left(P\left(a^f_c\right) \geq minP\right)$.
iii. $(C = \emptyset) \Rightarrow \left(P\left(a^f\right) \geq minP\right)$.

In other words, a^f and all its conditions must be contained in *EF*, and these conditions need to have a high enough[7] probability value *P*. If *E* is a non-conditional effect ($C = \emptyset$), we also require a^f itself to be probable enough.

Note 3. Note that if we have at least one probable condition a^f_c, we allow a^f to be improbable itself. That is because effect a^f can contain useful information even if it does not work most of the time, but we know under which conditions it applies. Imagine, for example, that a^f means *"eating food causes you to be sick."* In 99% of examples it does not happen. However, if in addition to that, we have a strong condition a^f_c meaning that *"eating food causes you to be sick only if the food is poisoned,"* then these two atoms together represent a useful piece of information.

For the purposes of the following theorems, we will establish a notation, where $\text{pos}(a^f)_c$ means the "*number of positive examples with respect to* a^f*, where fluent c was in observation o*" (and similarly for negative examples: $\text{neg}(a^f)_c$).

Theorem 3. *Consider a* 3*SG algorithm without forgetting* (memoryLength $= \infty$). *If we observe a sufficient number of positive, and no negative, examples with respect to*

a^f (precisely $(\text{memoryLength} = \infty)$*, then we can be sure that we have learned the nonconditional effect* $E = \langle a^f, \emptyset \rangle$*. We can also say that we have not learned any other conditional effect* $E' = \langle a^f,\ C \neq \emptyset \rangle$*.*

Theorem 4. *If we have observed at least one negative example with respect to a^f, then we need forgetting (memoryLength must be finite number) in order to learn the nonconditional effect* $E = \langle \mathrm{a^f}, \mathrm{C} = \emptyset \rangle$*.*

Theorem 5. Consider a *3*SG algorithm without forgetting (memoryLength $= \infty$*)*. If we observe a sufficient number of positive examples with respect to a^f (i.e., $pos\left(a^f\right) \geq minEx$), whereas the number of those where c was observed is sufficiently higher than those with $\bar{c}$ observed (more precisely $\forall a_c^f \in C : pos\left(a^f\right)_c \geq \min P.\left(pos\left(a^f\right)_c + pos\left(a^f\right)_{\bar{c}}\right)$), and, at the same time, every $a_c^f \in C$ is supported by at least minEx examples, and a^f has at least one negative example where $\bar{c}$ held, then we can safely say that we will learn the conditional effect $E = \langle a^f, C \neq \emptyset \rangle$.

Proof. Definition 2 enumerates three conditions that need to be met before we can say that we have *learned* the effect *E*. One after another, we will show how these conditions result from the premises of our theorem.

i. We know that $a^f \in EF$ because 3*SG* algorithm adds a^f to *EF* when confronted with first positive example (line 2 in Figure 2) and we assumed that $pos(a^f) \geq minEx$ and $minEx > 0$.
ii. In order for a_c^f to appear in *EF*, we need (according to line 16) the $\text{neg}\left(a^f\right)_{\bar{c}}$ to be greater than *0,* which is directly our assumption. Now consider that, according to semantics of *EF*, positive examples for a_c^f are exactly those examples that are positive with respect to a^f and $c \in o$. Similarly, negative examples for a_c^f are exactly those that are positive with respect to a^f, but with $\bar{c} \in o$. Our assumption "$\text{pos}\left(a^f\right)_c \geq \min P.\left(\text{pos}\left(a^f\right)_c + \text{pos}\left(a^f\right)_{\bar{c}}\right)$" can therefore be rewritten as "$\text{pos}\left(a_c^f\right) \geq \min P.\left(\text{pos}\left(a_c^f\right) + \text{neg}\left(a_c^f\right)\right)$". This can be further modified into "$\text{pos}\left(a_c^f\right) / \left(\text{pos}\left(a_c^f\right) + \text{neg}\left(a_c^f\right)\right) \geq \min P$," which is equivalent with the "$P\left(a_c^f\right) \geq \min P$" condition of Definition 2, because we also assume that a_c^f is sufficiently supported $\left(\text{pos}\left(a_c^f\right) + \text{neg}\left(a_c^f\right) \geq \min Ex\right)$.
iii. Finally, we will prove the last condition $\left((C = \emptyset) \Rightarrow \left(P\left(a^f\right) \geq \min P\right)\right)$ by contradiction. Let us assume its negation: $(C = \emptyset) \wedge \left(P\left(a^f\right) < \min P\right)$

The fact that C is empty means (assuming we don't allow empty observations), that there was no negative example with respect to a^f (the algorithm would otherwise add a_c^f into C on line 16). Because we assume that $\text{pos}\left(a^f\right) \geq \min Ex$ and we don't have any negative examples, the probability $P\left(a^f\right)$ must equal 1. Therefore it cannot be lower than min$P$.

CONCLUSION AND PLANS

We have shown that the time complexity of the 3*SG* algorithm is polynomial in the size of the input and also that it is formally correct under certain conditions. More importantly, our first practical experiments with its application in *Unreal Tournament 2004* and robotic platform *SyRoTek* suggest that it can be effectively used to learn the effects and preconditions of actions in real time, even in quite elaborate domains with considerable sensoric noise and action failures.

Currently, we are preparing additional theoretical results, especially based on the competitive analysis (Borodin and El-Yaniv 1998) of the algorithm. In the near future, we plan to examine the possibilities of automatic/dynamic modification of memory length and other learning parameters. In parallel, we will investigate the influence of the agent's behavior on the speed of the learning process. We believe that "explorative" behavior, where the agent would proactively try to confirm or disprove the hypotheses, might positively influence the learning rate and improve the overall quality of learned action models.

NOTES

1. To be exact, because $|\mathcal{S}| = 2^{|\mathcal{F}|/2}$, we can say that the space complexity of *EF* is lower, if $|\mathcal{F}| > 4$.

2. Effect formula can be equivalently understood either as a set of atoms or their conjunction (thus *formula*). Its name was inspired by a structure used in Amir and Chang (2008).

3. The name "3*SG*" is inspired by algorithm *SLAF* from Amir and Chang (2008).

4. FPS stands for "first person shooter" game genre. Players of the FPS game perceive their three-dimensional surroundings from their own viewpoint, while moving around, collecting items, and engaging in the combat.

5. Most commonly used values for β are 0.5 if the precision is more important, 2 if the recall is more important, or 1 if they are considered even.

6. Unlike the previous experiment, our agent here could observe only one action at a time. This means that, in this case, the number of time steps is the same as the number of examples.

7. Minimal required probability value is specified by a constant min$P$ from interval (0,1). For example, 0.9.

REFERENCES

Amir, E., and A. Chang. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33(1):349–402.

Amir, E., and S. Russell. 2003. Logical filtering, In *IJCAI*, Vol. 3, 75–82. Lawrence Erlbaum.

Balduccini, M. 2007. Learning action descriptions with a-prolog: Action language C. In *Proceedings of the AAAI spring symposium: Logical formalizations of commonsense reasoning*, 13–18. AAAI Press.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*, (vol. 2). New York, NY: Cambridge University Press.

Bayer, R., and E. M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1(3):173–189.

Borodin, A., and R. El-Yaniv. 1998. *Online computation and competitive analysis*. New York, NY, USA: Cambridge University Press.

Certicky, M. 2012. Action learning with reactive answer set programming: Preliminary report. In *ICAS 2012, The eighth international conference on autonomic and autonomous systems*, 107–111. IARIA.

Eiter, T., E. Erdem, M. Fink, and J. Senko. 2010. Updating action domain descriptions. *Artificial Intelligence* 174(15):1172–1221.

Eiter, T., W. Faber, N. Leone, G. Pfeifer, and A. Polleres. 2000. Planning under incomplete knowledge. In *Proceedings of the First International Conference on Computational Logic*, 807–821. CL '00, London, UK, UK: Springer-Verlag.

Fikes, R. E., and N. J. Nilsson. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3–4):189–208.

Gemrot, J., R. Kadlec, M. Bída, O. Burkert, R. Píbil, J. Havlíček, L. Zemčák, J. Šimlovič, R. Vansa, M. Štolba, T. Plch, and C. Brom. 2009. Agents for games and simulations. chap. Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents, 1–15. Berlin, Heidelberg: Springer-Verlag.

Gupta, N., and D. S. Nau. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2–3):223–254.

Kulich, M., J. Chudoba, K. Košnar, T. Krajnik, J. Faigl, and L. Přeučil. 2013. SyRoTek—distance teaching of mobile robotics. *IEEE Transactions on Education* 56(1):18–23.

Lewis, D. D., and J. Catlett. 1994. Heterogeneous uncertainty sampling for supervised learning. In *Proceedings of the eleventh international conference on machine learning*, 148–156. San Francisco, CA: Morgan Kaufmann.

Makhoul, J., F. Kubala, R. Schwartz, and R. Weischedel. 1999. Performance measures for information extraction. In *Proceedings of the DARPA broadcast news workshop*, 249–252. Herndon, VA: DARPA.

McDermott, D., M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. 1998. PDDL - The planning domain definition language. *Annals of Physics* 54(CVC TR-98-003):26.

Mourão, K., R. P. A. Petrick, and M. Steedman. 2010. Learning action effects in partially observable domains. In *Proceedings of the 2010 conference on ECAI2010: 19th European conference on artificial intelligence*, 973–974. Amsterdam, The Netherlands: IOS Press.

Nilsson, N. J. 1982. *Principles of artificial intelligence*. Berlin: Springer.

Rijsbergen, C. J. Van. 1979. *Information retrieval* (2nd ed.). Newton, MA, USA: ButterworthHeinemann.

Russell, S. J., and P. Norvig. 2003. *Artificial intelligence: A modern approach* (2nd ed.). Upper Saddle River, NJ: Pearson Education.

Slaney, J., and S. Thibaux. 2001. Blocks world revisited. *Artificial Intelligence* 125(1–2):119–153.

Yang, Q., K. Wu, and Y. Jiang. 2007. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence* 171(2–3):107–143.

Zettlemoyer, L. S., H. M. Pasula, and L. P. Kaelblin. 2005. Learning planning rules in noisy stochastic worlds. In *AAAI*, 911–918. AAAI Press.