Belief Update in the pGOLOG Framework

Henrik Grosskreutz and Gerhard Lakemeyer

Department of Computer Science V
Aachen University of Technology
D-52056 Aachen, Germany
{grosskreutz,gerhard}@cs.rwth-aachen.de

Abstract

High-level controllers that operate robots in dynamic, uncertain domains are concerned with at least two reasoning tasks dealing with the effects of noisy sensors and effectors: They have a) to project the effects of a candidate plan and b) to update their beliefs during on-line execution of a plan. In this paper, we show how the pGOLOG framework, which in its original form only accounted for the projection of high-level plans, can be extended to reason about the way the robot's beliefs evolve during the on-line execution of a plan. pGOLOG, an extension of the high-level programming language GOLOG, allows the specification of probabilistic beliefs about the state of the world and the representation of sensors and effectors which have uncertain, probabilistic outcomes. As an application of belief update, we introduce *belief-based programs*, GOLOG-style programs whose tests appeal to the agent's beliefs at execution time.

1 Introduction

High-level robot controllers that operate in dynamic, uncertain domains and have to cope with sensors and effectors that have uncertain, probabilistic outcomes are concerned with at least two distinct reasoning tasks. First, given a candidate plan, *probabilistic projection* allows the prediction of the effects of the plan. This task, which is a prerequisite to deliberate over different possible plans, lies at the heart of probabilistic planning [Kushmerick *et al.*, 1995; Draper *et al.*, 1994] and the theory of POMDPs [Kaelbling *et al.*, 1998]. Second, given a characterization of the robot's beliefs about the state of the world, a robot should be able to update its beliefs during the execution of a plan interacting with the robot's noisy sensors and effectors. This second task, to which we will refer to as *belief update*, following [Bacchus *et al.*, 1999], is necessary to allow probabilistic reasoning (in particular probabilistic projection) in non-initial situation.

In this paper, we show how the probabilistic high-level programming framework pGOLOG [Grosskreutz and Lakemeyer, 2000b], which in its original form only accounted for the probabilistic projection of high-level plans interacting with noisy sensors and effectors, can be extended to reason about the way the robot's beliefs evolve during the on-line execution of a plan.[1] To do so, we first explicitly model a layered robot control architecture where the robot's high-level controller does not directly affect the world by operating the robot's physical sensors and effectors, but instead is connected to a basic task-execution level which provides specialized *low-level processes* like navigation, object recognition or grasping objects. Having such a model has the advantage that there is a clear separation of the actions of the high-level controller from those of the low-level processes. In particular, while the action of activating a low-level process and its execution time are under the control of the high-level controller, neither are the effects of the activated process nor its completion time.

To model the effects of the low-level processes, we make use of probabilistic programs, where the different probabilistic branches of the programs correspond to different possible outcomes of the low-level processes.[2] Based on such a model of the possible effects of the low-level processes, we specify how the robot's beliefs about the state of the world evolve during the on-line execution of a plan, in particular how the beliefs change when the robot activates a low-level process that operates the robot's physical effectors or when a low-level process provides noisy information about the state of the world. Finally, we show how based on the robot's evolving belief state it becomes possible to execute so-called *belief-based programs*, GOLOG-style programs [Giacomo *et al.*, 2000] whose tests appeal to the agent's beliefs *at execution time*.

To get a better feel for what we are aiming at, let us consider the following *ship/reject*-example, adapted from [Draper *et al.*, 1994]: We are given a manufacturing robot with the goal of having a widget painted (*PA*) and processed (*PR*). Processing widgets is accomplished by rejecting parts that are flawed (FL) or shipping parts that are not flawed. Initially, the probability of being flawed is 0.3. *ship* and *reject* always make PR TRUE, however *ship* causes an execution error (ER) if FL holds, and *reject* causes ER to be TRUE if FL does not hold. The robot can activate a low-level process *paint*, which first under-coats the widget (UC) for 10 seconds, then takes 20 seconds to paint it. However, *paint* has a 5% probability to fail. There is also a low-level process *inspect* which can be used to determine whether or not the widget is flawed. However, *inspect* has a 10% probability to overlook a flaw and report *OK* instead of $\overline{OK}$ even though the widget is flawed; if the widget is not flawed, it always reports

[1] Here, we use the term on-line-execution in the sense of [de Giacomo and Levesque, 1999].

[2] We remark that modeling low-level processes as programs allows a very fine-grained characterization of the effects of the low-level processes at a level of detail involving many atomic actions, taking into account the temporal extent of the processes.

OK.

In this scenario, an example projection task is: how probable is it that the plan "first *inspect* the widget; thereafter, if *OK* holds then *ship* else *reject* it" will falsely ship a flawed widget. On the other hand, belief update is concerned with questions like: what is the probability that the widget is flawed if during on-line-execution the robot actually perceived *OK*.[3] The difference between the two tasks is that in the former case, the agent reasons about how the world might evolve, while in the latter case its beliefs change as a result of actual actions. We remark that besides updating its beliefs concerning the state of the world in terms of fluents like *PA* or *PR*, the robot also has to update its beliefs concerning the state of execution of the low-level processes; for example, 15 seconds after activation of the *paint* process the robot should not only be aware of the fact that the widget is under-coated by now, but also that the process is no longer in its initial state but only 15 seconds away from completion. Finally, a belief-based plan is a specification appealing to the robot's beliefs at execution time, like for example "as long as your (i.e. the robot's) confidence in whether the widget is flawed or not is below a threshold of 99%, (re-)inspect the widget. Thereafter, ship the widget if your belief in the widget being not flawed exceeds 99%, else reject it." Note that in this plan the activation of low-level processes is conditioned on the robot's beliefs at execution time.

The rest of this paper is organized as follows: after a brief review of the situation calculus and pGOLOG, we describe an overall robot control architecture for acting under uncertainty. Thereafter, we define successor state axioms that ensure that the robot's belief state evolves correctly during the course of action. Finally, we introduce belief-based programs and show how they can be used to solve the example problem. The paper ends with a discussion of related work and concluding remarks.

2 The Situation Calculus

We will only go over the situation calculus [McCarthy, 1963; Levesque *et al.*, 1998] briefly here: all terms in the language are of sort ordinary objects, actions, situations, or reals.[4] There is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation of s resulting from performing action a in s; relations and functions whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate resp. function symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form [Levesque *et al.*, 1998]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms (SSA), one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as a function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem.
- Domain closure and unique name axioms for the primitive actions.
- A collection of foundational, domain independent axioms. One of them defines how a situation s' can be reached from a situation s by a sequence of actions. Since we use it frequently, we define it here:[5]

  $$s \sqsubset do(a, s') \equiv s \sqsubseteq s'$$

  where $s \sqsubseteq s'$ stands for $(s \sqsubset s') \vee (s = s')$.

Adding a Timeline In its basic form, the situation calculus has no notion of time. In order to model that processes have temporal extent, we introduce a special unary functional fluent *start* of sort real. The understanding is that $start(s)$ denotes the time when situation s begins (we assume that $start(S_0)$ is 0). The fluent *start* changes its value only as a result of the special primitive action $tUpdate(t)$, with the intuition that, normally, every action is instantaneous, that is, the starting time of the situation after doing a in s is the same as the starting time of s. The only exception is $tUpdate(t)$. Whenever this action occurs, the starting time of the resulting situation is advanced up to t. The following axiom makes this precise.

$$Poss(a, s) \supset [start(do(a, s)) = t \equiv a = tUpdate(t) \vee t = start(s) \wedge \neg\exists t'.a = tUpdate(t')]$$

We will see in Section 4 how *tUpdate* actions are used to synchronize *start* with the actual time during on-line execution of robot plans. We remark that in [Grosskreutz and Lakemeyer, 2000a] a version of the temporal situation calculus is considered where it is possible to wait for conditions like a robot arriving at a certain location, which is modeled using continuous functions of time, an issue we ignore here for simplicity.

3 pGOLOG

As argued in [Grosskreutz and Lakemeyer, 2000b], robot "actions" such as *paint* or *inspect* are often best thought of as low-level processes with uncertain, probabilistic outcome which need to be described at a level of detail involving many atomic actions, rather than as primitive, atomic actions. To describe such processes, we proposed to model the processes as programs using the probabilistic language pGOLOG. The idea is to *model the noisy low-level processes as probabilistic programs*, where the different probabilistic branches of the programs correspond to different possible outcomes of the processes. Given a faithful characterization of the low-level processes in terms of pGOLOG programs, we can then reason about the effects of the activation of the processes through simulation of their corresponding pGOLOG models.

Besides constructs such as sequences, iterations and recursive procedures, pGOLOG provides a probabilistic branching instruction: $prob(p, \sigma_1, \sigma_2)$. Its intended meaning is to execute program σ_1 with probability p, and σ_2 with probability $1 - p$. In addition to the constructs already present in

[3] The respective probabilities are 0.3*0.1=3% and 3/73=4.1%.

[4] While the reals are not normally part of the situation calculus, we need them to represent probabilities. For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations.

[5] We use the convention that all free variables are implicitly universally quantified.

[Grosskreutz and Lakemeyer, 2000b], we introduce the parallel construct *withPol*(σ_1, σ_2) adapted from [Grosskreutz and Lakemeyer, 2000a]. The intuition is to execute σ_1 and σ_2 concurrently until σ_2 ends. The program σ_1 has a higher priority than σ_2, meaning that whenever both σ_1 and σ_2 are about to execute an action at the same time, σ_1 takes precedence. We remark that pGOLOG only provides deterministic instructions.

α	primitive action
$\phi?$	wait/test action[6]
$[\sigma_1, \sigma_2]$	sequence
$if(\phi, \sigma_1, \sigma_2)$	conditional
$while(\phi, \sigma)$	loop
$prob(p, \sigma_1, \sigma_2)$	probabilistic execution
withPol(σ_1, σ_2)	prioritized execution until σ_2 ends
proc$(\beta(\vec{x})\ \sigma)$	procedure definition

To illustrate the use of pGOLOG, we will now model the possible effects of *paint* by the pGOLOG program *paintProc*. Intuitively, if the widget is already processed, trying to *paint* it results in an error. Otherwise, 10 seconds after activation of *paint* the widget will become under-coated, and finally after 30 seconds *paint* will result in the widget being painted with probability 95% (there is also a 5% chance that *paint* will remain effectless). To model the effects of *paint*, we make use of the fluents *PA*, *FL*, *PR* and *ER* with the obvious meaning to represent the properties of our example domain, and assume successor state axioms that ensure that the truth value of *PA* is only affected by the primitive actions *setPA* and *clipPA*, whose effect is to make it TRUE resp. FALSE; similarly for the other fluents.

proc(*paintProc*,
  [*waitTime*(10), *if*(*PR*, *setER*, *setUC*),
   waitTime(20), *if*(*PR*, *setER*, *prob*(0.95, *setPA*))]).

n seconds. It essentially corresponds to a test *start* $\geq \tau + n?$, where τ refers to the time where *waitTime* was invoked. We will see in the Section 5 how this and similar pGOLOG models of the robot's low-level processes are used to update the robot's beliefs during on-line execution.

Formal Semantics The semantics of pGOLOG is defined using a so-called transition semantics similar to ConGolog [Giacomo *et al.*, 2000]. It is based on defining single steps of computation and, as we use a probabilistic framework, their relative probability. There is a function $transPr(\sigma, s, \delta, s')$ which, roughly, yields the transition probability associated with a given program σ and situation s as well as a new situation s' that results from executing σ's first primitive action in s, and a new program δ that represents what remains of σ after having performed that action.[7] Furthermore, there is another predicate *Final*(σ, s) which specifies which configurations (σ, s) are final, meaning that the computation can be considered completed. This is the case, roughly, when the remaining program is nil, but not if there is still a primitive action or test action to be executed.

For space reasons, we only list a few of the axioms for $transPr$ and *Final*. Let us first look at *withPol* and *prob* informally: the execution of σ_2 with policy σ_1 means that one action of one of the programs is performed, whereby actions which can be executed earlier are always preferred. If both σ_1 and σ_2 are about to execute an action at the same time, the policy σ_1 takes precedence. The whole *withPol* construct is completed as soon as σ_2 is completed. The execution of *prob*(p, σ_1, σ_2) results in the execution of a dummy, i.e. effectless action *tossHead* or *tossTail* with probability p resp. $1-p$ with remaining program σ_1, resp. σ_2. Let nil be the empty program and α a primitive action.

$transPr(nil, s, \delta, s') = 0$
$transPr(\alpha, s, \delta, s') =$
  if $Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s)$ **then** 1 **else** 0
$transPr([\sigma_1, \sigma_2], s, \delta, s') =$
  if $\delta = [\delta', \sigma_2]$ **then** $transPr(\sigma_1, s, \delta', s')$
  else if $Final(\sigma_1, s)$ **then** $transPr(\sigma_2, s, \delta, s')$
    else 0
$transPr(prob(p, \sigma_1, \sigma_2), s, \delta, s') =$
  if $\delta = \sigma_1 \wedge s' = do(tossHead, s)$ **then** p **else**
    if $\delta = \sigma_2 \wedge s' = do(tossTail(start, s)$ **then** $1-p$
    else 0
$transPr(withPol(\sigma_1, \sigma_2), s, \delta, s') =$
  if $\exists \delta_1 . \delta = withPol(\delta_1, \sigma_2) \wedge transPr(\sigma_1, s, \delta_1, s') > 0$
    $\wedge \neg Final(\sigma_2) \wedge \forall \delta_2, s_2 . transPr(\sigma_2, s, \delta_2, s_2) > 0 \supset$
      $start(s') \leq start(s_2)$ **then** $transPr(\sigma_1, s, \delta_1, s')$
  else if $\exists \delta_2 . \delta = withPol(\sigma_1, \delta_2) \wedge$
    $transPr(\sigma_2, s, \delta_2, s') > 0 \wedge \forall \delta_1, s_1 .$
      $transPr(\sigma_1, s, \delta_1, s_1) > 0 \supset start(s') < start(s_1)$
    then $transPr(\sigma_2, s, \delta_2, s')$ **else** 0

Final$(\alpha, s) \equiv$ FALSE
Final$(nil, s) \equiv$ TRUE
Final$(withPol(\sigma_1, \sigma_2), s) \equiv$ *Final*(σ_2, s)

So far, we have only defined which successor configurations can be reached through a single transition. The predicate $doPr(\sigma, s, s')$ defines the probability of an execution trace s' of program σ starting in s, that is the probability to end up in a final configuration with situation component s' after a *sequence* of transitions. In the following axiom, $transPr^*(\delta, s, \delta', s')$ refers to the transitive closure of $transPr$. Intuitively, if $\langle \delta', s' \rangle$ can be reached from $\langle \delta, s \rangle$, then $transPr^*(\delta, s, \delta', s')$ is the product of the probabilities of each transition along the path from $\langle \delta, s \rangle$ to $\langle \delta', s' \rangle$.[8]

$doPr(\delta, s, s') =$
  if $\exists \delta', p . p > 0 \wedge transPr^*(\delta, s, \delta', s') = p \wedge Final(\delta', s')$
  then p **else** 0

4 A Control Architecture for Acting under Uncertainty

In modern robot control architectures like RHINO [Burgard *et al.*, 2000], the robot's high-level controller does not directly affect the world by operating the robot's physical sensors and effectors, but instead is connected to a basic task-execution

[6] Here, a condition ϕ stands for a situation calculus formula where *now* may be used to refer to the current situation; when no confusion arises, we will simply leave out the *now* argument from the fluents altogether. Similarly, the term $\phi[s]$ denotes the formula obtained by substituting the situation variable s for all occurrences of *now* in fluents appearing in ϕ.

[7] Note that the use of a transition semantics necessitates the reification of programs as first order terms in the logical language, an issue we gloss over completely here (see [Giacomo *et al.*, 2000] for details). For space reasons, we also completely gloss over the definition of *proc*, which requires a second order definition of $transPr$.

[8] Defining the transitive closure of $transPr$ requires second order logic; for space reasons, we omit the definition, and refer the interested reader to [Grosskreutz and Lakemeyer, 2000b].

level which provides specialized low-level processes like navigation, object recognition or grasping objects. We will now describe how this type of architecture can be reconstructed in a logic-based framework; the architecture presented here is essentially an extention of [Grosskreutz and Lakemeyer, 2001], adapted to stochastic scenarios. In particular, we allow for the robot's uncertainty about the state of the world, account for the fact that low-level processes have uncertain outcomes, and show how to deal with processes like *inspect* which provide information about the state of the world, i.e. how to integrate *sensing* into our architecture. The resulting overall architecture is illustrated in Figure 1.

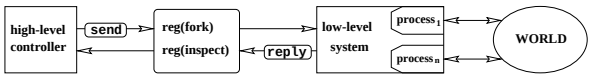

Figure 1: Robot Control Architecture for Acting under Uncertainty

In order to reason about the effects of a high-level plan, we need a model of every part of the robot control architecture illustrated in Figure 1.[9] Let us start with a representation of the state of the world.

4.1 The State of the World

While the original situation calculus allows us to talk only about the actual state of the world, in scenarios like the ship/reject example we have to represent uncertain beliefs about the state of the world. To do so, we follow [Bacchus *et al.*, 1999] and characterize the probabilistic epistemic state of a robot by a *set of situations considered possible*, and the *likelihood* assigned to the different possibilities. More specifically, there is a binary functional fluent $p(s', s)$ which can be read as "in situation s, the agent thinks that s' is possible with weight $p(s', s)$."[10] All weights must be non-negative and situations considered impossible will be given weight 0 (we do not require that the weights sum to 1). Furthermore, all situations considered possible in S_0 must be initial.

$$\forall s'.p(s', S_0) > 0 \supset \forall s'', a''.s' \neq do(a'', s'')$$

For example, in the introductory ship/reject domain the world is in one of two states, $s1$ and $s2$, which occur with probability 0.3 and 0.7, respectively. All other situations have likelihood 0. The following axiom makes this precise together with what holds and does not hold in each of the two states.

$$\begin{array}{l}\forall s.p(s, S_0) > 0 \supset \neg PA(s) \wedge \neg PR(s) \wedge \neg ER(s) \wedge \\ \exists s_1, s_2.p(s_1, S_0) = 0.3 \wedge p(s_2, S_0) = 0.7 \wedge \\ \quad FL(s_1) \wedge \neg FL(s_2) \wedge \\ \quad \forall s.s \neq s_1 \wedge s \neq s_2 \supset p(s, S_0) = 0\end{array}$$

Belief Based on p, [Bacchus *et al.*, 1999] define $Bel(\phi, s)$, the agent's degree of belief that ϕ holds in situation s, to be an abbreviation for the following term expressible in second-order logic (as before ϕ is a situation calculus formula where *now* is be used to refer to the current situation).

$$\Sigma_{\{s':\phi[s']\}} p(s', s) / \Sigma_{s'} p(s', s)$$

Intuitively, $Bel(\phi, s)$ is the normalized sum of the weights of all situations s' considered possible in s that satisfy ϕ. In our example, $Bel(FL, S_0)$ is 0.3.

4.2 Communication between low-level Processes and high-level Controller

We assume that the entire communication between the high-level controller and the low-level processes is achieved through a set of registers, and model them by the special functional fluent $reg(id, s)$. The high-level interpreter can affect the value of *reg* by means of the special action $send(id, val)$ which assigns $reg(id, s)$ the value *val*. The intuition is that in order to activate a low-level process, the high-level controller executes a *send* action. For example, the execution of $send(fork, paint)$ would tell the execution system to start the paint process.[11]

On the other hand, the low-level processes can provide the high-level controller with sensor information by means of the exogenous[12] action $reply(id, val)$. The following successor state axiom specifies how *reg* changes its value.

$$\begin{array}{l}Poss(a, s) \supset [reg(id, do(a, s)) = val \equiv \\ \quad a = send(id, val) \vee a = reply(id, val) \vee \\ \quad reg(id, s) = val \wedge \neg(\exists r, v.a = send(r, v) \vee a = reply(r, v))]\end{array}$$

We assume that initially, the value of the fluent *reg* is *nil* for all *id*, and that the robot know about this.

$$\forall id.reg(id, S_0) = nil$$

$$\forall s, id.p(s, S_0) > 0 \supset reg(id, s) = reg(id, S_0)$$

4.3 The Low-Level Execution System

Next, let us model the low-level execution system, starting with the individual low-level processes. As mentioned in Section 3, we model all low-level processes by probabilistic pGOLOG programs. While we have already modeled *paint* by the procedure *paintProc*, we model *ship* and *reject* by the following two pGOLOG programs. We assume that both processes take 10 seconds to complete execution, whereupon they confirm completion by means of a $reply(processed, t)$ action.

$$\begin{array}{l}proc(shipProc, \\ \quad [waitTime(10), \\ \quad\quad if(PR \vee FL, setER), setPR, reply(processed, t)])\end{array}$$

$$\begin{array}{l}proc(rejectProc, \\ \quad [waitTime(10), \\ \quad\quad if(PR \vee \neg FL, setER), setPR, reply(processed, t)])\end{array}$$

Sensor Processes Next, we turn to the process *inspect*. At this point, we have to explain what we mean by sensing. To us, sensing means: activate a sensor. This "activation" has as an effect a sensor reading. In the example, sensing happens

[9] Although the appropriate level of detail at which the low-level processes should be modeled is application dependent, we remark that a robot controller that lacks a model of the effects of its actions is intrinsically incapable to reason about the effects of its actions.

[10] Having more than one initial situation means that Reiter's induction axiom for situations [Levesque *et al.*, 1998] no longer holds, just as in [Bacchus *et al.*, 1999].

[11] The term *fork* was chosen in analogy to the procedure `fork` used in UNIX- like operating systems to create new concurrent processes.

[12] Here, an exogenous action is an action not under the control of the high-level controller.

through the activation of the *inspect* process, whose effect is to provide a $reply(inspect, OK)$ or $reply(inspect, \overline{OK})$ answer. We assume that the high-level controller is aware of all exogenous *reply* actions, as opposed to "actions" like *setPA* which are solely used to model the effects of the low-level processes. Sensing is thus realized by means of special low-level processes, which we call *sensor processes* and which communicate (pre-processed) sensor readings by means of exogenous *reply* actions.[13]

Although during real execution the actual low-level process *inspect* provides the answer, we need a model of the behavior of the sensor to update the robot's beliefs after *OK* or $\overline{OK}$ answers. The following pGOLOG program describes the possible effects of *inspect*:[14]

$proc(inspectProc,$
$\quad if(FL,[waitTime(10), prob(0.9, replay\overline{OK}, replyOK)],$
$\quad\quad [waitTime(10), replyOK]))$

Directly Observables *reply* actions like the above provide the high-level controller with information because they assign $reg(inspect, s)$ a value which is correlated with the value of *FL* (i.e. *OK* or $\overline{OK}$) and because unlike in the case of *FL* there is no uncertainty about the value of *reg*. Therefore, we distinguish *reg* from other fluents and call it *directly observable*, following [Grosskreutz and Lakemeyer, 2000b]. Directly observable fluents are such that the agent always has perfect information about them - like the display of one's watch or a fuel gauge in the car. Formally, we call a relational fluent P directly observable wrt a pGOLOG theory iff the following formula holds:

$\forall s, s', \vec{x}.\ [S_0 \sqsubseteq s \wedge p(s', s) > 0] \supset P(\vec{x}, s') \equiv P(\vec{x}, s)$

Directly observable functional fluents are defined similarly. We remark that the initial and successor state axioms for *reg* presented in this section together with the successor state axiom for p in the following section guarantee that in our example *reg* is in fact directly observable.

The Overall Low-Level Execution System Finally, we need a formal model of the execution system as a whole, i.e. of the robots operating system, which ensures that *send* actions result in the activation of the corresponding low-level process. The following program *kernelProc* describes the "kernel process" of the robot's operating system.

$proc(kernelProc, [reg(fork) \neq nil?,$
$\quad if(reg(fork) = inspect,$
$\quad\quad [reply(fork, nil), withPol(inspectProc, kernelProc)],$
$\quad\quad if(reg(fork) = paint,$
$\quad\quad\quad [reply(fork, nil), withPol(paintProc, kernelProc)],$
$\quad\quad\quad ...,\ \mathbf{else}\ [reply(fork, nil), kernelProc])...)])$

As long as $reg(fork)$ is nil, nothing happens. If $reg(fork)$ is assigned the name of a low-level process, then $reg(fork)$ is reset to nil, and the low-level process is run concurrently to the operating system's kernel process. We stress that pGOLOG programs such as the above are not intended for actual execution. Their purpose is solely to provide a *model* of the behavior of the low-level process.[15]

4.4 The High-level Controller

In order to ensure that the high-level controller will always have the necessary knowledge to evaluate tests within high-level robot plans, we consider only a subset of the pGOLOG programs as legal high-level plans. This subset of pGOLOG, to which we refer to as GOLOG_{rp}, consists of all programs whose tests are restricted to *directly observable fluents*, and which only execute actions that *only affect directly observables*. We gloss over the technical details. As an example, the following GOLOG_{rp} plan activates both *inspect* and *paint*, waits for their completetion and finally processes the widget according to the result of *inspect*.[16]

$Prg_{ex} \doteq$
$\quad [forkPaint, start \geq 30, forkInspect, reg(inspect) \neq nil?,$
$\quad\ if(OK, forkShip, forkReject), reg(processed) \neq nil?,]$

We remark that during on-line execution of a GOLOG_{rp} plan, whenever the high-level plan executes a *send* action, the interpreter checks whether this signals an activation of a low-level process and, as a side-effect, activates the actual low-level process if necessary.

The Passage of Time during On-line Execution Finally, a word on the passage of time during on-line execution of a high-level plan. In order to synchronise the internal clock, i.e. the value of the fluent *start* with the actual time during on-line execution, the high-level controller periodically generated exogenous $tUpdate(t)$ events, where t refers to the actual time. As described in Section 2, the effect of a *tUpdate* is then to assign *start* the actual time.[17]

5 Belief Update

Right now, we have a model of the robot's control architecture, of its beliefs about the state of the world, and of the execution system of the robot including models of the low-level processes. Based on this model, we will now specify how to update the robot's belief state as a result of the activation of noisy low-level processes and of the receipt of *reply* messages. We refer to this task as *(probabilistic) belief update*, following [Bacchus *et al.*, 1999].[18]

Although not quite obvious, the specification of a successor state axiom for the fluent p is not sufficent to represent the updated belief state. To see why, let us consider the situation S_{uc} where the robot has activated the *paint* process in

[13] Note that this view of sensing significantly differs from the well-known sensing actions of [Levesque, 1996].

[14] We use *replyOK* as an abbreviation for $reply(inspect, OK)$, similarly for $replay\overline{OK}$.

[15] In fact, pGOLOG programs like *inspectProc* or *paintProc* cannot be executed by the high-level controller because it has only uncertain information about the value of non-observable fluents like *FL*, resp. because it cannot directly execute actions like *setPA*.

[16] We use *OK* as an abbreviation for $reg(inspect, OK)$, *forkInspect* as an abbreviation for $send(fork, inspect)$, and similarly for *forkPaint*, *forkShip* and *forkReject*.

[17] We assume that the difference $\Delta = t_{i+1} - t_i$ between two subsequent updates $tUpdate(t_i)$ and $tUpdate(t_{i+1})$ is smaller than the minimal delay between the execution time of any two actions of the pGOLOG models which have different execution time. Furthermore, we assume that if a *reply* is modeled to happen at time t, then during on-line execution the high-level controller will generate a *tUpdate* action causing *start* to advance to t before the actual *reply* action happens.

[18] See [Shapiro *et al.*, 2000] for the connection to the general area of belief update and belief revision.

the initial situation through *send*(*fork*, *paint*) at time 0, after which it has waited for 15 seconds. Intuitively, the epistemic state should reflect the fact that the activation of the low-level process *paint* has affected the truth value of *UC*. But this is not sufficient. Additionally, the robot should be aware of the fact that unlike in S_0, in S_{uc} the low-level process *is active*, has already executed *setUC*, and is about to probably execute *setPA*. Thus, the *paint* process is no longer correctly characterized by *paintProc*, but instead by the remaining fragment of *paintProc* after 15 seconds have passed.

The example suggests that the appropriate pGOLOG model of the low-level processes is not the same for all situations, but depends on the history of actions. Thus, we associate with every possible situation a specific pGOLOG model. Formally, we introduce a special functional fluent $ll(s', s)$ that can be read as "in situation s, the robot thinks that if the world is in state s' then the low-level processes can be characterized by the pGOLOG program $ll(s', s)$." The following axiom states that in the initial situation the low-level processes are as described by *kernelProc* (defined above).

$$\forall s.p(s, S_0) \supset ll(s, S_0) = \textit{kernelProc}.$$

In order to specify successor state axioms for $p(s^*, do(a, s))$ and $ll(s^*, do(a, s))$, stating how the world and the low-level processes evolve from a situation s to its successor situation $do(a, s)$, we have to distinguish two cases: (i) a is a *reply* action performed by a sensor process; and (ii) a is an action executed by the high-level controller or a *tUpdate* action. The reason that we have to distinguish *reply* actions from other "ordinary" actions is that *reply* actions provide sensing information, as captured by the pGOLOG model of the sensing processes (like, for example, *inspectProc*).[19]

5.1 Ordinary Actions

Let us first consider the second case. Our solution is that the low-level processes execute up to the point where one of the following conditions occur:

1. they are blocked, i.e. waiting for a $\phi?$ condition to become true;
2. or they are about to execute an *reply* action.

While the first condition is fairly obvious, the reason that we mind *reply* actions is that the high-level controller is aware of all *reply* actions, and a is no *reply* action. We will now formalize the idea to execute a program σ in s until a configuration $\langle \delta, s' \rangle$ is reached where one of the above conditions is true. For this, we use the special function $transPr^{\triangleleft}(\sigma, s, \delta, s')$ which specifies the probability to end up in $\langle \delta, s' \rangle$ starting in $\langle \sigma, s \rangle$. In the following formulas, $\Re(a)$ is a shorthand for $\exists r, v.a = reply(r, v)$.

$$\begin{array}{l} transPr^{\triangleleft}(\sigma, s, \delta, s') = \\ \quad \textbf{if } transPr^*(\sigma, s, \delta, s') > 0 \wedge \\ \quad\quad \forall a^*, s^*.s \sqsubset do(a^*, s^*) \sqsubseteq s' \supset \neg\Re(a^*) \wedge \\ \quad\quad \forall \delta^*, s^*.transPr(\delta, s', \delta^*, s^*) > 0 \supset \\ \quad\quad\quad \exists a^*.s^* = do(a^*, s') \wedge \Re(a^*) \\ \quad \textbf{then } transPr^*(\sigma, s, \delta, s') \textbf{ else } 0 \end{array}$$

While the second line of the **if** condition verifies that $\langle \delta, s' \rangle$ can be reached from $\langle \sigma, s \rangle$ without executing any *reply* action, the last two lines verify that all successor configurations of $\langle \delta, s' \rangle$ can only be reached by a violation of one of the above conditions, meaning that the simulation has been pursued as far as possible.

Using $transPr^{\triangleleft}$, we can define which configurations $\langle s^*, ll^* \rangle$ have been reached by the low-level processes in $do(a, s)$ together with their weight (assuming that a is no *reply* action). Intuitively, these are all configuration that result from the execution via $transPr^{\triangleleft}$ of a configuration $\langle ll(s', s), s' \rangle$ considered possible in s. Their weight is the product of the weight of s' in s and the transition probability as specified by $transPr^{\triangleleft}$. The predicate *advConfig*$(s^*, ll^*, do(a, s))$ makes this precise.

$$\begin{array}{l} \textit{advConfig}(s^*, ll^*, do(a, s)) = p \equiv \\ \quad \exists s', p', p^*.p(s', s) = p' \wedge \\ \quad transPr^{\triangleleft}(ll(s', s), s', ll^*, s^*) = p^* \wedge \\ \quad p' > 0 \wedge p^* > 0 \wedge p = p' \cdot p^* \vee \\ \quad \neg\exists s'.[p(s', s) > 0 \wedge transPr^{\triangleleft}(ll(s', s), s', ll^*, s^*) > 0] \wedge \\ \quad p = 0 \end{array}$$

5.2 *reply* Actions

Now that we have formalized how the low-level processes evolve if a is an ordinary action, let us turn to the other case where a is a *reply* action. Intuitively, the observation of a *reply* should sharpen the belief state of the robot. For example, if the robot observes a *reply*$\overline{OK}$ action after activation of *inspect*, it can rule out those situations from its belief state where $\neg FL$ holds. In general, the observation of a *reply* action can be used to *rule out* those situations whose associated pGOLOG model of the low-level processes ll is not about to execute this very *reply* action. To make this precise, we define the predicate *adv&filter*$(s^*, ll^*, do(a, s))$ which - if a is an *reply* action - preserves only those configurations of *advConfig* whose pGOLOG-component is about to execute a.

$$\begin{array}{l} \textit{adv\&filter}(s^*, ll^*, do(a, s)) = p \equiv \exists s'.s^* = do(a, s') \wedge \\ \quad [\neg\Re(a) \wedge \textit{advConfig}(s', ll^*, do(a, s)) = p \vee \\ \quad \Re(a) \wedge [\exists s'', ll'', p'', p^*.\textit{advConfig}(s'', ll'', do(a, s)) = p'' \\ \qquad\qquad \wedge transPr^*(ll'', s'', ll^*, s^*) = p^* \wedge \\ \qquad\qquad p'' > 0 \wedge p^* > 0 \wedge p = p'' \cdot p^* \vee \\ \qquad\qquad \neg\exists s'', ll''.[\textit{advConfig}(s'', ll'', do(a, s)) > 0 \\ \qquad\qquad\quad \wedge transPr^*(ll'', s'', ll^*, s^*) > 0] \wedge p = 0]] \end{array}$$

If a is an ordinary action, *adv&filter* is almost like *advConfig*; the only difference is that all situations s^* considered in $do(a, s)$ now "end" with action a, i.e. $\exists s'.s^* = do(a, s')$. However, if a is a *reply* action, then we keep only those situations s^* in the belief state whose associated pGOLOG model correctly predicted that the *reply* action a would be executed next.

Successor State Axioms for p and ll It can be shown that the function *adv&filter* is well-defined, meaning that any configuration $\langle s^*, ll^* \rangle$ with positive weight is assigned exactly one weight. Furthermore, it can be shown that for each situation s^* there is at most one ll^* such that *adv&filter*$(s^*, ll^*, do(a, s)) > 0$. Therefore, p and ll can simply be defined as the situation resp. pGOLOG component of *adv&filter*.

$$\begin{array}{l} p(s^*, do(a, s)) = p \equiv \exists ll^*.\textit{adv\&filter}(s^*, ll^*, do(a, s)) = p \\ \quad \wedge p > 0 \vee \forall ll^*.\textit{adv\&filter}(s^*, ll^*, do(a, s)) = 0 \wedge p = 0 \end{array}$$

$$\begin{array}{l} ll(s^*, do(a, s)) = ll^* \equiv \textit{adv\&filter}(s^*, ll^*, do(a, s)) > 0 \vee \\ \quad \forall ll'.\textit{adv\&filter}(s^*, ll', do(a, s)) = 0 \wedge ll^* = nil \end{array}$$

[19] As stated above, we assume that the high-level controller is not aware of any other low-level "actions" than the *reply* actions.

5.3 Examples

To illustrate how p and ll evolve, and in particular how the perception of an exogeneous *reply* action is used to sharpen the robots beliefs, we will now consider the value of p and ll in different situations. We begin with the situation $S_{\text{inspect}} \doteq do([fork, inspect), reply(fork, nil), tUpdate(1), ..., tUpdate(10)], S_0)$, already mentioned above (we assume that a *tUpdate* action is generated every second). Let Γ be the foundational axioms of Section 2 (except for the induction axiom) together with the successor state axioms for p and ll, action precondition axioms stating that all *set* and *clip* actions are always possible, successor state axioms for the fluents *PA*, *FL*, *PR* and *ER*, and the probabilistic characterization of the initial state of Section 4. Then, from Γ we can deduce that in S_{inspect} two situations are considered possible. Intuitively, the first one corresponds to the case where the widget is flawed and the second one to the case where it is not flawed. Furthermore, we can deduce that these situations have an associated pGOLOG model of the low-level processes that accounts for the fact that the *paint* process is active and about to provide a *reply*.

$$\begin{aligned}\Gamma \models\ & \forall s', ll'.p(s', S_{\text{inspect}}) > 0 \wedge ll(s', S_{\text{inspect}}) = ll' \equiv \\ & \exists s^*.s' = do([send(fork, inspect), reply(fork, nil), ..., \\ & \qquad tUpdate(10)], s'_0) \\ & \wedge [ll' = conc(kernelProc, \\ & \qquad [start \geq 10?, prob(0.9, replyOK, replyOK)]) \\ & \quad \vee ll' = conc(kernelProc, [start \geq 10?, replyOK])]\end{aligned}$$

We remark that so far the belief concerning the value of *FL* remains unchanged ($\Gamma \models Bel(FL, S_{\text{inspect}}) = Bel(FL, S_0)$). Now assume that the inspect process provides a *reply*$\overline{OK}$ answer, leading to Situation $S_{\neg ok} \doteq do(replyOK, S_{\text{inspect}})$. Intuitively, we would expect that after this observation the robot no longer considers a situation possible where the widget is not flawed. Indeed, we can deduce that in $S_{\neg ok}$ the robot only considers one situation possible, and that *FL* holds in this situation.

$$\begin{aligned}\Gamma \models\ & p(s', S_{\neg ok}) = p \wedge p > 0 \equiv \exists s'_0.\ p(s'_0, S_0) > 0 \wedge \\ & s' = do([send(fork, inspect), reply(fork, nil), ..., \\ & \qquad tUpdate(10), tossHead, replyOK], s'_0) \wedge \\ & FL(s') \wedge FL(s'_0)\end{aligned}$$

Intuitively, the only situation that remains in the belief state corresponds to the simulation trace where *FL* holds, and *inspect* correctly reports *reply*$\overline{OK}$. This corresponds to the execution of the first branch of the *prob* instruction in the pGOLOG model *inspectProc*, leading to a *tossHead* action in the resulting execution trace. All other simulation traces would end up in a *replyOK* answer, and are thus ruled out from the belief state (see *adv&filter*). We remark that the resulting belief state implies $Bel(FL, S_{\neg ok}) = 1$.

Similarly, if the robot would observes *replyOK*, we could deduce that only two situations are considered possible in the resulting situation $do(replyOK, S_{\text{inspect}})$: one corresponding to the widget beeing flawed (prob. 30%) *and* inspect erroneously reporting *OK* (prob. 10%), and another one where the widget is not flawed (prob. 70%). The resulting belief in $\neg FL$ would then correspond to the normalized probability of the second case, which is $0.7/(0.7 + 0.3 * 0.1) = \frac{70}{73}$.

As another example, let us consider the situation $S_{uc} \doteq do([send(fork, paint), ..., tUpdate(15)], S_0)$, where the *paint* process is active for 15 seconds. In this situation, the low-level process paint has already caused *UC* to become true, and is waiting until time 30 whereat it may cause *PA* to become true.

$$\begin{aligned}\Gamma \models\ & p(s', S_{uc}) > 0 \equiv \\ & \exists s^*.s' = do([send(fork, paint), ..., tUpdate(10), setUC, \\ & \qquad tUpdate(11), ..., tUpdate(15)], s^*) \wedge \\ & ll(s', S_{uc}) = conc(kernelProc, \\ & \qquad [start \geq 30?, if(PR, setER, \\ & \qquad\qquad prob(0.95, setPA))])\end{aligned}$$

Some seconds later, the robots belief in *PA* will rise to 95% due to the fact that it will assume that paint has finished execution. However, the robot's belief in the widget beeing flawed will remain unchanged.

6 Belief-Based Programming

As an application of belief update, we will now introduce the concept of *belief-based programs*, GOLOG$_{rp}$ programs that appeal to the robot's beliefs at execution time.[20] In particular, we introduce a special epistemic test $BTest(\phi, p, s)$, which is true if in situation s the robot's belief in ϕ is p. Formally, $BTest(\phi, p, s)$ is a defined relational fluent which is true iff $Bel(\phi, s) = p$. Using *BTest* within test conditions, a GOLOG$_{rp}$ plan can appeal to the robot's beliefs *at execution time*. As an example, the following plan specifies that the robot is to activate the *inspect* process until it is sufficiently confident about whether the widget is flawed or not. Thereafter, the widget is painted and processed.[21]

$$\begin{aligned}& proc(savePaint, \\ & \quad [\ while(\exists p.BTest(FL, p) \wedge \neg[p = 1 \vee p \leq 0.001], \\ & \qquad send(inspect, nil), forkInspect, reg(inspect) \neq nil?), \\ & \quad forkPaint, waitTime(30), \\ & \quad if(BTest(FL, 1), forkReject, forkShip), \\ & \quad reg(processed) \neq nil?]\end{aligned}$$

The while loop is executed until the robot is sure that the widget is flawed, i.e. $Bel(FL) = 1$, or the probability that the widget is flawed drops below 0.001, meaning that the robot is sufficiently confident that the widget is ok. We remark that due to the fact that after the observation of $\overline{OK}$ the robot is sure that the widget is flawed, the above program causes at most three activations of *inspect*.[22]

Unlike ordinary GOLOG programs which are conditioned on facts about the world, in belief-based programs like the above actions are conditioned on the robot's belief state at execution time. As the example illustrates, belief-based programs allow the programmer to provide domain dependent procedural knowledge in a natural way. From a pragmatic point of view, belief-based programming can be an attractive alternative to probabilistic planning because it represents a much simpler computational problem. While probabilistic planners are searching from scrach for an (optimal) plan, which in the worst case means that an exponential number of candidate plans has to be projected, the execution of a belief-based program only requires the computation of the belief state of the robot along the execution of a given plan.

[20] This is similar to Reiter's notion of knowledge-based programming [Reiter, 2000]. However, we remark that here we are dealing with *degrees of belief*.

[21] As usual, we leave out the *now* argument in the tests, in particular in the epistemic fluent *BTest*.

[22] As we have seen in the previous section, the observation of *one OK* answer causes the robot's belief in $\neg FL$ to rise to $70/73 = 0.7/(0.7 + 0.3 * 0.1)$. Similarly, the observation of two resp. three *OK*s cause the robot's belief to rise to $0.7/(0.7 + 0.3 * 0.1 * 0.1)$ resp. $0.7/(0.7 + 0.3 * 0.1 * 0.1 * 0.1)$.

Implementation Just as in the case of ConGolog, it is straightforward to implement a pGOLOG interpreter in PROLOG. We remark that our implementation was able to execute the above belief-based plan in a fraction of a second.

7 Discussion

Summarizing, we have shown how to update the probabilistic belief state of a robot during on-line execution of high-level GOLOG$_{rp}$ plans. To do so, we have modeled a layered robot control architecture within the pGOLOG framework, making use of probabilistic pGOLOG programs to model noisy low-level processes. In order to deal with sensing, we have introduced the concept of sensor processes, low-level processes whose activation results in exogenous *reply* actions. Finally, we have introduced belief-based programs, GOLOG$_{rp}$ programs whose tests appeal to the agent's beliefs at execution time. We remark that unlike approaches like [Levesque, 1996; Bacchus *et al.*, 1999], we represent the belief state of the agent by a set of possible situations *and an associated model of the state of execution of the low-level processes*, which allows us to account for noisy processes with temporal extent.

The whole framework, in particular the definition of p and ll, relies on the fact that pGOLOG programs are deterministic. As a result, it is not possible to specify unprioritized concurrency as done in ConGolog where the resulting course of actions is not uniquely determined. However, when we consider processes with temporal extent, this does not seem to be a severe restriction, because the priority of a process manifests only when two processes wish to execute an action at exactly the same time; actions with different execution times are not affected.

Probably the closest work to that reported in this paper is that of Bacchus, Halpern and Levesque [Bacchus *et al.*, 1999], to which we owe the characterization of the robot's epistemic state. However, while we manage solely with the *prob* instruction to represent noise, they make use of the concepts of *nondeterministic instructions*, *action-likelihood axioms* $OI(a, a', s)$ and *observation-indistinguishability* axioms $l(a, s)$, and represent the execution of noisy actions as atomic. This results in a simpler SSA for p, but at the cost of a more complex specification of the effects of the noisy sensors and effectors. Furthermore, it is not clear how to project a plan within their framework. On the other hand, probabilistic projection in the pGOLOG framework was already considered in [Grosskreutz and Lakemeyer, 2000b], and it would be relatively straightforward to consider both projection and belief update within pGOLOG.[23]

As for probabilistic planners like C-Buridan [Draper *et al.*, 1994], they usually completely ignore belief update. Besides, they represent processes as atomic actions. The latter also holds for the theory of POMDPs (which is concerned with both reasoning tasks), but whose computational cost is prohibitive already in relatively small domains. We believe that in many domains, the use of belief-based programs providing procedural knowledge is more promising than uninformed search for an optimal plan. In [Poole, 1998], Poole proposes an integration of decision theory and the situation calculus, which however is primarily concerned with the expected utility of a candidate plan. Finally, the recently proposed *DT-Golog* [Boutilier *et al.*, 2000] assumes full observability of the domain. All of these approaches do not account for the temporal extent of the low-level processes.

[23] One possibility would be to explicitly distinguish between on-line and off-line execution of GOLOG$_{rp}$ plans, and to make use of the special action *waitFor* to cause time to advance during projection, along the lines of [Grosskreutz and Lakemeyer, 2001].

References

[Bacchus *et al.*, 1999] F. Bacchus, J.Y. Halpern, and H. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence 111(1-2)*, 1999.

[Boutilier *et al.*, 2000] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI'2000*, 2000.

[Burgard *et al.*, 2000] W. Burgard, A.B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. Experiences with an interactive museum tour-guide robot. *Artificial Intelligence*, 114(1-2), 2000.

[de Giacomo and Levesque, 1999] G. de Giacomo and H.J. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundations for Cognitive Agents*, pages 86–102. Springer, 1999.

[Draper *et al.*, 1994] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. of AIPS'94*, 1994.

[Giacomo *et al.*, 2000] Guiseppe De Giacomo, Yves Lesperance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

[Grosskreutz and Lakemeyer, 2000a] H. Grosskreutz and G. Lakemeyer. cc-golog: Towards more realistic logic-based robot controllers. In *AAAI'2000*, 2000.

[Grosskreutz and Lakemeyer, 2000b] H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *ECAI'2000*, 2000.

[Grosskreutz and Lakemeyer, 2001] H. Grosskreutz and G. Lakemeyer. Online-execution of ccgolog plans. In *IJCAI'2001*, 2001.

[Kaelbling *et al.*, 1998] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence 101(1-2*, 1998.

[Kushmerick *et al.*, 1995] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[Levesque *et al.*, 1998] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. URL: http://www.ep.liu.se/ea/cis/1998/018/.

[Levesque, 1996] H. J. Levesque. What is planning in the presence of sensing. In *AAAI'96*, 1996.

[McCarthy, 1963] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University. Reprinted 1968 in Semantic Information Processing (M.Minske ed.), MIT Press, 1963.

[Poole, 1998] David Poole. Decision theory, the situation calculus and conditional plans. *Linköping Electronic Articles in Computer and Information Science*, 3(8), 1998. URL: http://www.ep.liu.se/ea/cis/1998/008/.

[Reiter, 2000] R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Second International Cognitive Robotics Workshop*, 2000.

[Shapiro *et al.*, 2000] S. Shapiro, M. Pagnucco, Y. Lesprance, and H. J. Levesque. Iterated belief change in the situation calculus. In *KR'2000*, 2000.