Proceedings of the Cognitive Robotics Seminar
Dagstuhl, Germany
Feb. 22–26, 2010

Gerhard Lakemeyer, Hector Levesque, Fiora Pirri

Contents

Introduction, Lakemeyer, Levesque, Pirri 2
Cognitive robotics, Lakemeyer 3
Mapping and meaning, Little 22
Collaborative activity and human-robot interaction, Kruijff 23
Learning functional object categories and event classes, Cohn 24
Acquiring models through human-robot interaction, Wachsmuth 25
Attending to motion: an object-based approach, Belardinelli 27
Plan recognition in the situation calculus, Lespérance 38
Coming up with good excuses, Nebel 39
On first-order definability, Liu 47
Towards a non-Prolog implementation of Golog, Ferrein 54
Challenges for domestic service robots, Behnke 69
Robust and efficient visual SLAM, Calway 70
Constraint-based plan management, Pecora 71
Detecting human activities in video and still images, Hlavac 79
Context and place categorization for assistive robotics, Little 80
Modeling robot behavior through machine learning, Ghallab 81
Combining modelling and learning for skill acquisition, Sammut 83
Attentive monitoring and adaptive control, Finzi 84
Combining planning and motion planning, Amir 92
Fast replanning, Koening 100
Spatial computing, Freksa 101
Embodied cognition and human-robot interaction, Trafton 102
The GLAIR cognitive architecture, Shapiro 103
Collaborative unmanned aircraft systems, Doherty 115
Plan execution of hybrid under-actuated systems, Williams 129
Stream-based reasoning in DyKnow, Heintz 137
Self-maintenance for autonomous robots, Schiffer 153
Improving performance of plans though learning, Leonetti 161

Dagstuhl Seminar Proceedings 10081
Cognitive Robotics
http://drops.dagstuhl.de/portals/10081

The Dagstuhl 2010 Cognitive Robotics Workshop

Gerhard Lakemeyer	Hector J. Levesque	Fiora Pirri
RWTH Aachen	University of Toronto	Sapienza Universita di Roma

Research in robotics has traditionally emphasized low-level sensing and control tasks including sensory processing, path planning, and manipulator design and control. Research in Cognitive Robotics, on the other hand, emphasizes those cognitive functions that enable robots and software agents to reason, act and perceive in changing, incompletely known, and unpredictable environments. Such robots must, for example, be able to reason about goals, to choose actions and to focus on patterns, objects and events according to the task execution and the cognitive states of other agents, by taking into account time, resources, and the consequences of their decisions. In short, cognitive robotics is concerned with integrating reasoning, perception, and action within a uniform theoretical and implementation framework.

The term cognitive robotics and the vision that knowledge representation and reasoning plays a fundamental role in the design of cognitive robots was first laid out by the late Ray Reiter in his lecture on receiving the Research Excellence Award by the International Joint Conference on Artificial Intelligence (IJCAI) in 1993. Since 1998, biannual Cognitive Robotics workshop with Dagstuhl being the seventh in this series.

While the earlier workshops were largely a forum for presenting state-of-the-art research results, the purpose of the Dagstuhl event was to broaden the view and bring together people from various disciplines to shed new light on the issues in cognitive robotics. In this respect we were very fortunate to have participants from areas such as robotics, machine learning, cognitive vision, computational neuroscience, and knowledge representation and reasoning.

Given the diversity of the group, we spent the first day with tutorial-style presentation, starting out with an overview of Cognitive Robotics in the sense of Ray Reiter's vision by Gerhard Lakemeyer. This was followed by presentations on Computational Neuroscience by Laurent Itti, Planning and Execution Monitoring by Brian Williams, Probabilistic Reasoning by Eyal Amir, Cognitive Vision by Jim Little, and Human-Robot Interaction by Geert-Jan Kruijff. The rest of the Workshop consisted of research presentations, a panel, and three breakout discussion groups on the following topics: the nature of perception, symbolic and numerical uncertainty, and the role of automated reasoning.

The Proceedings collected here represent submissions from all but 2 of the participants at the Workshop in the order they were given. In some cases we have short abstracts of the talks, and in others, full research papers. We thank the participants and the copyright holders for permission to use these papers.

Cognitive Robotics*

Hector J. Levesque
Dept. of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 3A6
hector@cs.toronto.edu

Gerhard Lakemeyer
Dept. of Computer Science
RWTH Aachen
52056 Aachen
Germany
gerhard@cs.rwth-aachen.de

This chapter is dedicated to the memory of Ray Reiter. It is also an overview of cognitive robotics, as we understand it to have been envisaged by him.[1] Of course, nobody can control the use of a term or the direction of research. We apologize in advance to those who feel that other approaches to cognitive robotics and related problems are inadequately represented here.

1 Introduction

In its most general form, we take *cognitive robotics* to be the study of the knowledge representation and reasoning problems faced by an autonomous robot (or agent) in a dynamic and incompletely known world. To quote from a manifesto by Levesque and Reiter [42]:

> "Central to this effort is to develop an understanding of the relationship between the knowledge, the perception, and the action of such a robot. The sorts of questions we want to be able to answer are
>
> - to execute a program, what information does a robot need to have at the outset vs. the information that it can acquire *en route* by perceptual means?
> - what does the robot need to know about its environment vs. what need only be known by the designer?
> - when should a robot use perception to find out if something is true as opposed to reasoning about what it knows was true in the past?
> - when should the inner workings of an action be available to the robot for reasoning and when should the action be considered primitive or atomic?

[0]Reprinted from: Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation,* Chapter 23, pp. 869–886, Copyright (2007), with permission from Elsevier.

[1]To the best of our knowledge, the term was first used publicly by Reiter at his lecture on receiving the IJCAI Award for Research Excellence in 1993.

> and so on. With respect to robotics, our goal (like that of many in AI) is *high-level robotic control*: develop a system that is capable of generating actions in the world that are appropriate as a function of some current set of beliefs and desires. What we do *not* want to do is to simply engineer robot controllers that solve a class of problems or that work in a class of application domains. For example, if it turns out that online reasoning is unnecessary for some task, we would want to know what it is about the task that makes it so."

We take this idea of knowledge representation and reasoning *for the purpose of* high-level robotic control to be central to cognitive robotics [75]. This connects cognitive robotics not only to (traditional, less cognitive) robotics but also, as discussed later, to other areas of AI such as planning and agent-oriented programming.

To illustrate the knowledge representation and reasoning issues relevant to high-level robotic control, we will use Reiter's variant of the situation calculus. There are several reasons for this: we, the authors, have worked with the situation calculus and hence feel most comfortable with it; the situation calculus is a very expressive formalism which can be used to model many of the features relevant to cognitive robotics; it was already introduced at length in a chapter of this volume (which we assume as a prerequisite), so that we do not need to present it from scratch; and last but not least, it is a tribute to Ray Reiter. For a book length treatment of cognitive robotics *not* based on the situation calculus, see [85].

The structure of the this chapter is as follows. In Section 2, we discuss some of the knowledge representation issues that arise in the context of cognitive robotics. In Section 3, we turn to problems in automated reasoning in the same setting. In Section 4, we examine how knowledge representation and reasoning come to bear on the issue of high-level agent control. Finally, in Section 5, we briefly draw conclusions and suggest a direction for future research.

2 Knowledge representation for cognitive robots

As a special sort of knowledge-based system, cognitive robots need to represent knowledge about relevant parts of the world they inhabit. What makes them special is the emphasis on knowledge about the *dynamics* of the world, including, the robot's own actions. In currently implemented systems, knowledge about objects in the world can be very simple, as in robotic soccer [21], where little is known beyond their position on a soccer field, to the very complex, involving knowledge about the actual shape of the objects [60, 71]. Likewise, knowledge about actions can be as simple as taking an action to be a discrete change of position from A to B, or fairly involved with probabilistic models of success and failure [23, 22].

But whatever the application, the key feature of cognitive robotics is the focus on a changing world. A suitable knowledge representation language must at the very least provide *fluents*, that is, predicate or function symbols able to change their values as a result of changes in the world. For our purposes, we will use the situation calculus; but there are many other possible choices, modal vs. non-modal, state-based vs. history-based, time-based vs. action-based, and so on.[2] Each of these will need to address similar sorts of

[2]While planning languages like STRIPS [28] or PDDL [57] also qualify and have been used to control

issues such as the frame, qualification, and ramification problems, discussed in the Situation Calculus chapter, and in [70].

2.1 Varieties of actions

In its simplest setting, the situation calculus is used to model actions that change the world in a discrete fashion and instantaneously. For robotic applications, this is usually far too limited and we need much richer varieties. Let us begin with actions which are continuous and have a duration. A simple idea to accommodate both is due to Pinto [58], who proposed to split, say, a *pickup* action into two (instantaneous) *startPickup* and *endPickup* actions with an additional time argument and a new fluent *Pickingup* with the following successor state axiom:

$$\begin{aligned} \mathit{Pickingup}(x,t,do(a,s)) \;\equiv\; & \exists t'(a = \mathit{startPickup}(x,t') \wedge t' \leq t) \\ \vee \quad & \mathit{Pickingup}(x,t,s) \wedge \neg\exists t'(a = \mathit{endPickup}(x,t') \wedge t' \leq t). \end{aligned}$$

While this works fine for some applications,[3] having to explicitly specify time points when an action starts and ends is often cumbersome if not impossible. An alternative approach, first introduced by Pinto [58] and later adapted by Grosskreutz and Lakemeyer [30] is to define fluents as continuous functions of time. For example, a robot's location while moving may be approximated by a linear function taking as arguments the starting time of the moving action and the robot's velocity. Using the special action called *waitFor*(ϕ) time advances until the condition ϕ becomes true. The use of *waitFor* was actually inspired by robot programming languages like RPL [53]. For an approach to continuous change in the event calculus see [72].

The situation calculus also deals with actions whose effects are deterministic, that is, where there is no doubt as to which fluents change and which do not. In practice, however, the world is often not that clear cut. For example, the robot's gripper may be slippery and the *pickup* action may sometimes fail, that is, sometimes it holds the object in its gripper afterwards and sometimes it does not. There have been a number of proposals to model nondeterministic effects such as [82, 27, 4]. On a more fine-grained level, which is often more appropriate in robotics applications, one also attaches probabilities to the various outcomes. Reiter's stochastic situation calculus [66], for example, achieves this by appealing to nature choosing among various deterministic actions according to some probability distribution. For example, imagine that when the robot executes a *pickup* action, nature actually chooses one of two deterministic actions *pickupS* and *pickupF*,which stand for a successful and failed attempt and which occur, say, with probabilities .95 and .05, respectively. A nice feature of this approach is that successor state axioms can be defined as usual because they only appeal to nature's choices, which are then deterministic.

---

robots [55, 61, 20], they are more limited in that they only specify planning problems, but do not lend themselves to a general representation and reasoning framework for cognitive robots as advocated by Reiter.

[3] Thinking of *all* actions as instantaneous in this way also has the advantage of reducing the need for true action parallelism, allowing us to use the much simpler variant of interleaved concurrency [17].

2.2 Sensing

In the situation calculus, actions are typically thought of as changes to the world, in particular, those which are due to a robot's actuators. Sensing actions, which provide the robot with information about what the world is like but leave the world unchanged otherwise, are of equal importance from a robot's perspective. Various ways to model sensing in the situation calculus have been proposed. One is to introduce a special fluent $\mathit{SF}(a, s)$ (for *sensed fluent value*) and axioms describing how the truth value of *SF* becomes correlated with those aspects of a situation which are being sensed by action a [41]. For example, suppose we have a sensing action $\mathit{senseRed}(x)$, which registers whether the colour of object x is red. This can be captured by the following axiom:

$$\mathit{SF}(\mathit{senseRed}(x), s) \;\equiv\; \mathit{Colour}(x, \mathit{red}, s).$$

The idea is that, when the robot executes *senseRed*, its sensors or perhaps more concretely, its image processing system, returns a truth value, which then tells the robot whether the object in question is red. We can use this predicate to define what the robot learns by doing actions $a_1, a_2, \ldots, a_n$ in situation s and obtaining binary sensing results $r_1, r_2, \ldots, r_n$:

$$\begin{aligned}
\mathit{Sensed}(\langle\,\rangle, \langle\,\rangle, s) &\stackrel{\text{def}}{=} True; \\
\mathit{Sensed}(\vec{a}\cdot A, \vec{r}\cdot 1, s) &\stackrel{\text{def}}{=} \mathit{SF}(A, do(\vec{a}, s)) \land \mathit{Sensed}(\vec{a}, \vec{r}, s); \\
\mathit{Sensed}(\vec{a}\cdot A, \vec{r}\cdot 0, s) &\stackrel{\text{def}}{=} \neg \mathit{SF}(A, do(\vec{a}, s)) \land \mathit{Sensed}(\vec{a}, \vec{r}, s).
\end{aligned}$$

In general, of course, sensing results are not binary. For example, reading the temperature could mean returning an integer or real number. See [79] on how these can be represented. Noisy sensors can be dealt with as well, as shown in [3, 73]. For the distinction between sensing and perception, see [59].

Sensing the colour of an object is usually deliberate, that is, the robot chooses to actively execute an appropriate sensing action. There are, however, cases where sensing results are provided in a more passive fashion. Consider, for example, a robot's need to localize itself in its environment. In practice, this is often achieved using probabilistic techniques such as [86], which continuously output estimates of a robot's pose relative to a map of the environment. Grosskreutz and Lakemeyer [32] show how to deal with this issue using so-called *exogenous* actions. These behave like ordinary non-sensing actions, which change the value of fluents like the robot's location. The only difference is that they are not issued by the robot "at will," but are provided by some external means. See also [15, 68] for how passive sensors can be represented by other means. Exogenous actions are not limited to account for passive sensing. In general, they can be used to model actions which are not under the control of the robot, including those performed by other agents.

2.3 Knowledge

When a robot has a model of its environment in the form of, say, a basic action theory, this represents what the agent *knows* or *believes* about the world. Yet so far there is no explicit notion of knowledge as part of the theory, and this may not be necessary, if we are interested only in the logical consequences of that theory. However, this changes when we need to refer to what the robot does *not* know, which is useful, for example, when

deciding whether or not to sense. We need an explicit account of knowledge also when it comes to knowledge about the mental life (including knowledge) of other agents. In the situation calculus, knowledge is modeled possible-world style[4] by introducing a special fluent $K(s', s)$, which is read as "situation s' is (epistemically) accessible from s." Let $\phi[s]$ be a formula that is uniform in s. Then knowing ϕ at a situation s, written as $\mathit{Knows}(\phi, s)$, means that ϕ is true in all accessible situations:

$$\mathit{Knows}(\phi, s) \stackrel{\text{def}}{=} \forall s'.K(s', s) \supset \phi[s'].$$

This idea of reifying possible worlds was first introduced by Moore [54]. Later, Scherl and Levesque [79] showed that the way an agent's knowledge changes as a result of actions can be captured by a successor state axiom for the fluent K:

$$K(s'', \mathit{do}(a, s)) \equiv \exists s'.s'' = \mathit{do}(a, s') \land K(s', s) \land [\mathit{SF}(a, s') \equiv \mathit{SF}(a, s)].$$

In words: a situation s'' is accessible after action a is performed in s just in case it is the result of doing a in some other situation s' which is accessible from s and which agrees with s on the value of *SF*. The effect of this axiom is, roughly, that it eliminates from further consideration all those situations which disagree with the result of sensing. For example, if a $\mathit{senseRed}(A)$ action returns the value *true*, only those situations remain accessible after performing the action where A is red. Note that this notion of epistemic alternatives generalizes the situation calculus discussed in the chapter of this volume in that we now assume that there are initial situations other than S_0.[5] One nice feature of the successor state axiom for K is that general properties of the accessibility relationship like reflexivity or transitivity only need to be stipulated for initial situations, as they are guaranteed to hold ever after [79]. For a treatment of knowledge and sensing in the fluent calculus, see [83]. For approach to knowledge in the situation calculus that avoids using additional situations, see [19].

Besides knowledge, there are many other mental attitudes that a cognitive robot may find useful to model. Proposals exist, for example, to model *goal* or *ability*, also using a possible-world semantics [78, 39, 50, 36]. The issue of belief change after receiving information that conflicts with what is currently known about the world has also been addressed [76, 77]. Here a preference relation over situations plays an essential role.

3 Reasoning for cognitive robots

The research problems in cognitive robotics are not limited to problems in representation seen in the previous section. We are fundamentally concerned with how these representations are to be *reasoned* with, and furthermore, as we will see in the next section, how this reasoning can be used to control the behaviour of the robots.

3.1 Projection via progression and regression

There are two related reasoning tasks that play a special role in cognitive robotics. The main one is called the (temporal) *projection task*: determining whether or not some condition

[4]Modeling knowledge using possible worlds is due to Hintikka [35].

[5]Instead of a single tree rooted at S_0, we now have a forest of trees each with their own initial situation.

will hold after a sequence of actions has been performed starting in some initial state. The second one is called the *legality task*: determining whether a sequence of actions *can* be performed starting in some initial state. Assuming we have access to the preconditions of actions, legality reduces to projection, since we can determine legality by verifying that the preconditions of each action in the sequence are satisfied in the state just before the action is executed. Projection is a very basic task since it is necessary for a number of other larger tasks, including planning and high-level program execution, as we will see in the next section.

We can summarize the definition of projection from the Situation Calculus chapter as follows: given an action theory $\mathcal{D}$, a sequence of ground action terms, $a_1, \ldots, a_n$, and a formula $\phi[s]$ that is uniform in s, the task is to determine whether or not

$$\mathcal{D} \models \phi[do(\vec{a}, S_0)].$$

As explained in that chapter, one of the main results proved by Reiter in his initial paper on the frame problem [65] is that the projection problem can be solved by *regression*: when $\mathcal{D}$ is a basic action theory (as defined in the earlier chapter), there is a regression operator $\mathcal{R}$, such that for any ϕ uniform in s,

$$\mathcal{D} \models \phi[do(\vec{a}, S_0)] \quad \text{iff} \quad \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \models \phi'[S_0],$$

where $\mathcal{D}_{S_0}$ is the part of $\mathcal{D}$ that characterizes S_0, and $\phi' = \mathcal{R}(\phi, \vec{a})$. So to solve the projection problem, it is sufficient, to regress the formula using the given actions, and then to determine whether result holds in the initial situation, a much simpler entailment.

Regression has proven to be a powerful method for reasoning about a dynamic world, reducing it to reasoning about a static initial situation. However, it does have a serious drawback. Imagine a long-lived robot that has performed thousands or even millions of actions in its lifetime, and which at some point, needs to determine whether some condition currently holds. Regression involves transforming this condition back through the thousands or millions of actions, and then determining whether the transformed condition held initially. This is not an ideal way of staying up to date.

The alternative to regression is *progression*. In this case, we look for a progression operator $\mathcal{P}$ that can transform an initial database $\mathcal{D}_{S_0}$ into the database that results after performing an action. More precisely, we want to have that

$$\mathcal{D} \models \phi[do(\vec{a}, S_0)] \quad \text{iff} \quad \mathcal{D}_{una} \cup \mathcal{D}'_0 \models \phi[S_0],$$

where $\mathcal{D}_{S_0}$ is the part of $\mathcal{D}$ that characterizes S_0, and $\mathcal{D}'_0 = \mathcal{P}(\mathcal{D}_{S_0}, \vec{a})$. The idea is that as actions are performed, a robot would change its database about the initial situation, so that to determine if ϕ held after doing actions $\vec{a}$, it would be sufficient to determine if ϕ held in the progressed situation (with no further actions), again a much simpler entailment. Moreover, unlike the case with regression, a robot can use its *mental idle time* (for example, while it is performing physical actions) to keep its database up to date. If it is unable to keep up, it is easy to imagine using regression until the database is fully progressed.

There are, however, drawbacks with progression as well. For one thing, it is geared to answering questions about the *current* situation only. In progressing a database forward, we effectively lose the historical information about what held in the past. It is, in other words,

a form of *forgetting* [48, 38]. While questions about a current situation can reasonably expected to be the most common, they are not the only meaningful ones.

A more serious concern with progression is that it is not always possible. As Lin and Reiter show [49], there are simple cases of basic action theories where there is no operator $\mathcal{P}$ with the properties we want. (More precisely, the desired $\mathcal{D}'_0$ would not be first-order definable.) To have a well-defined projection operator, it is necessary to impose further restrictions on the sorts of action theories we will use.

3.2 Reasoning in closed and open worlds

So far, we have assumed like Reiter, that $\mathcal{D}_{S_0}$ is any collection of formulas uniform in S_0. Regression reduces the projection problem to that of calculating logical consequences of $\mathcal{D}_{S_0}$. In practice, however, we would like to reduce it to a much more tractable problem than ordinary first-order logical entailment. It it is quite common for applications to assume that $\mathcal{D}_{S_0}$ satisfies additional constraints: domain closure, unique names, and the closed-word assumption [64]. With these, for all practical purposes, $\mathcal{D}_{S_0}$ does behave like a database, and the entailment problem becomes one of database query evaluation. Furthermore, progression is well defined, and behaves like an ordinary database transaction.

Even without using (relational) database technology, the advantage of having a $\mathcal{D}_{S_0}$ constrained in this way is significant. For example, it allows us to use Prolog technology directly to perform projection. For example, to find out if $(\phi \vee \psi)$ holds, it is sufficient to determine if ϕ holds or if ψ holds; to find out if $\neg\phi$ holds, it is sufficient to determine if ϕ does not hold (using negation as failure), and so on. None of these are possible with an unconstrained $\mathcal{D}_{S_0}$.

This comes at a price, however. The unique name, domain closure and closed-world assumptions amount to assuming that we have *complete knowledge* about S_0: anytime we cannot infer that ϕ holds, it will be because we are inferring that $\neg\phi$ holds. We will never have the status of ϕ undecided.

This is obviously a very strong assumption in a cognitive robotic setting, where it is quite natural to assume that a robot will not know everything there is to know about its world. Indeed we would expect that a cognitive robot might start with incomplete knowledge, and only acquire the information it needs by actively *sensing* its environment as necessary.

A proposal for modifying Reiter's proposal for the projection problem along these lines was made by de Giacomo *et al* [15]. They show that a modified version of regression can be made to work with sensing information. They also consider how closed-world reasoning can be used in an open world using what they call *just-in-time queries*. In a nutshell, they require that queries be evaluated only in situations where enough sensing has taken place to give complete information about the query. Overall, the knowledge can be incomplete, but it will be locally complete, and allow us to use closed-world techniques.

Another independent proposal for dealing effectively with open-world reasoning is that of Liu and Levesque [51]. (A related proposal is made by Son and Baral [80] and by Amir and Russell [1].) They show that what they call *proper knowledge bases* represent open-world knowledge. They define a form of progression for these knowledge bases that provides an efficient solution to the projection problem that is always logically sound, and under certain circumstances, also logically complete. The restrictions involve the type of

successor-state axioms that appear in the action theory $\mathcal{D}$: they require action theories that are *local-effect* (actions only change the properties of the objects that are parameters of the action) and *context-complete* (either the actions are context-free or there is complete knowledge about the context of the context-dependent ones).

4 High-level control for cognitive robots

As noted earlier, one distinguishing characteristic of the area of cognitive robotics is that the knowledge representation and reasoning are for a particular purpose: the control of robots or agents. We reason about a world that is changing as the result of actions taken by agents *because* we are attempting to decide what to do, what actions to take towards some goal. This is in contrast, for example, to reasoning for the purposes of answering questions or generating explanations.

4.1 Classical planning

Perhaps the clearest case of this application of knowledge representation and reasoning is in *classical planning* [29]. As discussed in the Situation Calculus chapter, we are given an action theory $\mathcal{D}$ of the sort discussed above and a goal formula, $\phi[s]$ that is uniform in some situation variable s. The task is to find a sequence of ground actions terms $\vec{a}$ such that

$$\mathcal{D} \models \phi[do(\vec{a}, S_0)] \wedge Executable(do(\vec{a}, S_0)).$$

Thus, we are looking for a sequence of actions which, according to what we know in $\mathcal{D}$, can be legally executed starting in S_0 and result in a state where ϕ holds.

Think of having a robot, and wanting it to achieve some goal ϕ. Instead of simply programming it directly, we get the robot to use what is known about the initial state of the world and the actions available to figure out what to do to achieve the goal. This has the very desirable effect that if information about the world changes, that is, if we learn something new, or discover that something old was incorrect, it will not be necessary to reprogram the robot. All we need do is revise its beliefs. Using the terminology of Zenon Pylyshyn [62], we have an architecture that is *cognitively penetrable* in that the behaviour of the robot can be altered by simply changing its beliefs about the world.

In practice, very little of the actual research in classical planning is formulated using the situation calculus in this way. Rather, it is expressed in the more restrictive notation of STRIPS [28]. Instead of an action theory, we have an the initial database formulated as a set of atomic formulas (with an implicit closed-world assumption), and a collection of actions formulated as operators on databases, with preconditions and effects characterized by the additions and deletions they would make to a current database. Although STRIPS has a very operational flavour, it is possible to reconstruct its logical basis in the situation calculus [44, 49].

Despite the restrictions imposed by STRIPS, the classical planning task remains extremely difficult. Even in the propositional case (and with complete knowledge about the initial world state), the problem is NP-hard [10]. While many optimizations exist for many special cases, nobody would consider planning as a *practical* way of generating the millions

of action that might be required of a long-lived robot to achieve long-term goals starting from some initial state.

But this is an unreasonable picture anyway. Nobody would expect *people* to deal with their long-term goals by first closing their eyes and computing a sequence of millions of action, and then blindly carrying out the sequence to achieve the goal, even assuming such a sequence were to exist. This is an *offline* view of how to decide what to do. We need to consider a much more *online* view of high-level control, where as actions are taken, new information that is acquired gets to contribute to the decision-making. Instead of planning in advance for all possible long-term contingencies, we need to be able to get a robot to achieve some part of a goal, assess its current situation, and plan for the rest with the new information taken into account.

4.2 High-level offline robot programming

In an attempt to come up with a more flexible sort of control, one of the directions that has proven to be quite fruitful is the *high-level programming* [42] found in languages such as those in the Golog family [43, 17, 16, 66] and variants like FLUX [84]. Virtually all of the high-level control currently considered in cognitive robotics is of this sort. This brings cognitive robotics closer to the area of *agent-oriented programming* or AOP (see [33, 63], for example).[6]

By a high-level program, we mean a program that contains the usual programming features (like sequence, conditional, iteration, recursive procedures, concurrency) and some novel ones:

- the *primitive statements* of the program are the actions that are characterized by an action theory;
- the *tests* in the program are conditions about the world formulated in the underlying knowledge representation language;
- programs may contain *nondeterministic* operations, where a reasoned choice must be made among alternatives.[7]

Instead of planning given a goal, we now consider program execution given a high-level program. In the situation calculus, Levesque *et al* [43] make this precise as follows: they define an operator $Do(\delta, s, s')$ that maps any high-level program δ into a formula of the situation calculus with two free variables s and s'. Intuitively $Do(\delta, s, s')$ is intended to say that if program δ starts in situation s, one of the situations it may legally terminate in (since the program need not be deterministic) is s'. This is defined inductively on the structure of the program:

Primitive action: $Do(A, s, s') \stackrel{\text{def}}{=} Poss(A, s) \wedge s' = do(A, s);$

Test: $Do(\phi?,\, s,\, s') \stackrel{\text{def}}{=} \phi[s] \wedge s' = s;$

[6] This is perhaps a difference of emphasis: cognitive robotics tends to emphasize the robotic interaction with the world, whereas AOP tends to emphasize the mental state of the agent executing the program.

[7] In many applications, we can preserve the effectiveness of an essentially deterministic situation calculus by pushing the nondeterminism into the programming.

Sequence: $Do(\delta_1;\delta_2,\, s,\, s') \;\stackrel{\text{def}}{=}\; \exists s''.\, Do(\delta_1,\, s,\, s'') \wedge Do(\delta_2,\, s'',\, s);$

Nondeterministic branch: $Do(\delta_1|\delta_2,\, s,\, s') \;\stackrel{\text{def}}{=}\; Do(\delta_1,\, s,\, s') \vee Do(\delta_2,\, s,\, s');$

Nondeterministic value: $Do(\pi x.\, \delta,\, s,\, s') \;\stackrel{\text{def}}{=}\; \exists x.\, Do(\delta, s, s');$

Nondeterministic iteration: $Do(\delta^*,\, s,\, s') \;\stackrel{\text{def}}{=}$

$$\forall P[\forall s_1 P(s_1, s_1) \;\wedge\; \forall s_1 s_2 s_3 (P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s')].$$

Other programming common constructs can be defined in terms of these:

if ϕ **then** δ_1 **else** $\delta_2 \;\stackrel{\text{def}}{=}\; (\phi?;\delta_1)\,|\,(\neg\phi?;\delta_2);$

while ϕ **do** $\delta \;\stackrel{\text{def}}{=}\; (\phi?;\delta)^*;\neg\phi?.$

The *offline high-level program execution task* then is the following: given a high-level program δ find a sequence of actions $\vec{a}$ such that

$$\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0)).$$

As with planning, we solve this task and then give the resulting action sequence to the robot for execution.

While this is still completely offline like planning, it does allow for far more flexibility in the specification of behaviour. Consider, for example, a high-level program like the following

$$A_1\,;\, A_2\,;\, A_3\,;\, \ldots\, A_n\,;\, \phi?$$

where each A_i is a primitive action and ϕ is some condition. This program can only be executed in one way, that is, by performing the A_i in sequence and then confirming that ϕ holds in the final state (or fail otherwise). We would naturally expect that solving the execution task for this program would be trivial, even if n were large, since the program already contains the answer. At the other extreme, consider a program like the following:

while $\neg\phi$ **do** $\pi a.\, a$

This is a very nondeterministic program. It says: while ϕ is false, pick an action a and do it. A correct execution of this program is a sequence of actions that can be legally executed and such that ϕ holds in the final state. But finding such a sequence is precisely the planning task for ϕ. So the execution task for this program is no different than the general planning task. However, it is between these two extremes that we can see advantages over planning. Consider this variant:

while $\neg\phi$ **do** $\pi a.\, Acceptable(a)?\,;\, a$

In this case, we have modified the previous program to include a test that the nondeterministically selected action a must satisfy. Assuming we have appropriate domain-dependent knowledge (represented in $\mathcal{D}$) about this *Acceptable* predicate, we can constrain the planning choices at each stage anyway we like, such as in the forward filtering of [2]. Similarly, we can generalize the first example as in the following:

$A_1\,;\, A_2\,;\, A_3\,;\, [$**while** $\neg\psi$ **do** $\pi a.a]\,;\, A_4\,;\, (A_5\,|\,B_5)\,;\, \phi?.$

In this case, we begin the same way, but then we must solve a (presumably easier) subplanning problem to achieve ψ, then perform A_4, followed by either A_5 or B_5 as appropriate. In nutshell, what we see here is that the high-level program can provide as much or as little procedural guidance as deemed necessary for high-level robot control.

This strategy has proven to be very effective. Among some of the applications built in this way, we mention an automated banking agent that involved a 40-page Golog program [67]. This is an example of high-level specification that would have been completely infeasible formulated as a planning problem.

When a program contains nondeterministic actions, all that matters about the actual choices is that they lead to a successful execution of the entire program. There is no reason to prefer one execution over another. However, real decision making often involves determining which choices are *better* than others. One way to address this issue is to attach numerical *rewards* to situations. Consider, for example, a robot whose only job is to collect objects, but with a preference for red ones. We might use the following successor state axiom for *reward*:

$$\begin{aligned}
&\mathit{reward}(do(a,s)) = r \;\equiv \\
&\qquad \exists x(a = \mathit{pickup}(x) \wedge \mathit{Colour}(x,\mathit{red},s) \wedge r = \mathit{reward}(s) + 10) \quad \vee \\
&\qquad \exists x(a = \mathit{pickup}(x) \wedge \neg\mathit{Colour}(x,\mathit{red},s) \wedge r = \mathit{reward}(s) + 5) \quad \vee \\
&\qquad \neg\exists x(a = \mathit{pickup}(x) \wedge r = \mathit{reward}(s)).
\end{aligned}$$

The operator $\mathit{Do}(\delta, s, s')$ introduced above is then replaced by $\mathit{BestDo}(\delta, s, s')$ which selects sequences of actions that maximize accumulated reward. Note that, in the above example, this does not necessarily mean that the robot will always pick up a red object if one is available, as even higher rewards may be unattainable if a red object is picked up now. When combining the idea of maximizing rewards with probabilistic actions, we obtain a decision-theoretic version of Golog, which was first proposed in [8].

4.3 High-level online robot programming

The version of high-level programming we have considered so far has been *offline*. A more online version is considered by de Giacomo *et al* [16, 69]. Instead of using *Do* to define the complete execution of a program, they consider the single-step method first-used to define the offline execution of ConGolog [17]. This is done in terms of two predicates, $\mathit{Final}(\delta, s)$, and $\mathit{Trans}(\delta, s, \delta', s')$. Intuitively, $\mathit{Final}(\delta, s)$ holds when program δ can legally terminate in situation s, and $\mathit{Trans}(\delta, s, \delta', s')$ holds when program δ can legally take one step resulting in situation s', with δ' remaining to be executed. It is then possible to redefine the *Do* in terms of these two predicates:

$$\mathit{Do}(\delta, s, s') \stackrel{\text{def}}{=} \exists\delta'(\mathit{Trans}^*(\delta, S_0, \delta', s') \wedge \mathit{Final}(\delta', s')),$$

where Trans^* is defined as the reflexive transitive closure of *Trans*.[8]

Now imagine that we started with some program δ_0 in S_0, and that at some later point we have executed certain actions $a_1, \ldots a_k$, and that we have obtained sensing results $r_1, \ldots r_k$ from them, with program δ remaining to be executed. The *online high-level program execution task* then is to find out what to do next, defined by:

[8]Much of the work with *Trans* and *Final* requires quantifying over and therefore reifying programs. Some care is required here to ensure consistency since programs may contain formulas in them. See [17] for details.

- stop, if $\mathcal{D} \cup \textit{Sensed}(\vec{a}, \vec{r}, S_0) \models \textit{Final}(\delta, do(\vec{a}, S_0))$;
- return the remaining program δ', if

  $$\mathcal{D} \cup \textit{Sensed}(\vec{a}, \vec{r}, S_0) \models \textit{Trans}(\delta, do(\vec{a}, S_0), \delta', do(\vec{a}, S_0)),$$

  and no action is required in this step;
- return action b and δ', if

  $$\mathcal{D} \cup \textit{Sensed}(\vec{a}, \vec{r}, S_0) \models \textit{Trans}(\delta, do(\vec{a}, S_0), \delta', do(b, do(\vec{a}, S_0))).$$

So the online version of program execution uses the sensing information that has been accumulated so far to decide if it should terminate, take a step of the program with no action required, or take a step with a single action required. In the case that an action is required, the robot can be instructed to perform the action, gather any sensing information this provides, and the online execution process iterates.

The online execution of a high-level program has the advantage of not requiring a reasoner to determine a lengthy course of action, requiring perhaps millions of actions, before executing the first step in the world. It also gets to use the sensing information provided by the first n actions performed so far in deciding what the $(n + 1)$ action should be. On the other hand, once an action has been executed in the world, there may be no way of backtracking if it is later found out that a nondeterministic choice was resolved incorrectly. In other words, an online execution of a program may fail where an offline execution would succeed.

To deal with this issue, de Giacomo *et al* propose a new programming construct, a *search operator*. The idea is that given any program δ the program $\Sigma(\delta)$ executes online just like δ does offline. In other words, before taking any action, it first ensures using offline reasoning that this step can be followed successfully by the rest of δ. More precisely, we have that

$$\textit{Trans}(\Sigma(\delta), s, \Sigma(\delta'), s') \equiv \textit{Trans}(\delta, s, \delta', s') \wedge \exists s^*.\textit{Do}(\delta', s', s^*).$$

If δ is the entire program under consideration, $\Sigma(\delta)$ emulates complete offline execution. But consider $[\delta_1 \,;\, \delta_2]$. The execution of $\Sigma([\delta_1 \,;\, \delta_2])$ would make any choice in δ_1 depend on the ability to successfully complete δ_2. But $[\Sigma(\delta_1) \,;\, \delta_2]$ would allow the execution of the two pieces to be done separately: it would be necessary to ensure the successful completion of δ_1 before taking any steps, but consideration of δ_2 is deferred. If we imagine, for example, that δ_2 is a large high-level program, with hundreds of pages of code, perhaps containing Σ operators of its own, this can make the difference between a scheme that is practical and one that is only of theoretical interest.

The idea of interleaving execution and search has also been applied to decision-theoretic Golog [81, 21]. Here, instead of just searching for a successful execution of a sub-program, an optimal sub-plan is generated which maximizes the expected accumulated reward.

Being able to search still raises the question of how much offline reasoning should be performed in an online system. The more offline reasoning we do, the safer the execution will be, as we get to look further into the future in deciding what choices to make now. On the other hand, in spending time doing this reasoning, we are detached from the world and will not be as responsive. This issue is very clearly evident in time-critical applications such

as robot soccer [21] where there is very little time between action choices to contemplate the future. Sardina has cast this problem as the choice between deliberation and reactivity [68], and see also [6].

Another issue arises in this setting is the form of the offline reasoning. Since an online system allows for a robot to acquire information during execution (via sensing actions, or passive sensors, or exogenous events), how should the robot deal with this during offline deliberation [12]. The simplest possibility is to say that it ignores any such information in the plan for the future that it is constructing. A more sophisticated approach would have it construct a plan that would prescribe different behaviour depending on the information acquired during executing. This is *conditional planning* (see, for example, [7, 56]) and one form of this has been incorporated in high-level execution by Lakemeyer [37]. Another possibility is to attempt to *simulate* what will happen external to the robot, and use this information during the deliberation [40]. In [31], this idea is taken even further: at deliberation time a robot uses, for example, a model of its navigation system by computing, say, piece-wise linear approximations of its trajectory; at execution time, this model is then replaced by the real navigation system, which provides position updates as exogenous actions.

Another issues arises whenever a robot performs at least some amount of lookahead in deciding what to do. What should the robot do when the world (as determined by its sensors) does not conform to its predictions (as determined by its action theory)? First steps in logically formalizing this possibility were taken by de Giacomo *et al* [18] in what they call *execution monitoring*. In [21], a simple form of execution monitoring is implemented for soccer-playing robots. Here, the assumptions made by the decision-theoretic planner are explicitly encoded in the generated plan. During execution, these assumptions are re-evaluated against the current world model and, in case of a disagreement, the plan is discarded and a new one generated. See also [26, 34, 22, 23, 24] for related approaches.

5 Conclusion

Cognitive robotics is a reply to the criticism that knowledge representation and reasoning has been overly concerned with reasoning in the abstract and not concerned enough with the dynamic world of an embodied agent. It attempts to address the sort of representation and reasoning problems an autonomous robot would face in trying to decide what to do. In many ways, it has only scratched the surface of the issues that need to be dealt with.

A number of cognitive robotic systems have been implemented on a variety of robotic platforms, using the sort of ideas discussed in this chapter, based either on the situation calculus or on one of the other related knowledge representation formalisms. For a sampling of these systems, see [14, 13, 5, 74, 21, 21, 11, 25]. Perhaps the most impressive demonstration to date was that of the museum tour-guide robot reported in [9].

A fundamental question in the area of cognitive robotics (that Reiter had begun to examine) is the relationship between pure logical representations of incomplete knowledge and the more numerical measures of uncertainty. A start in this direction is the work on the stochastic situation calculus [66] as well as that on noisy sensors and effectors and decision-theoretic Golog, noted above.

On an even broader scale, a much tighter coupling of the high-level control program and

other parts of a robot's software, like mapping and localization, or even vision, is called for. For example, when localization fails and a robot gets lost, it should be possible to use high-level control to do a reasoned failure recovery. Making progress along these lines requires a deep understanding of both cognitive and more traditional robotics, and should help to reduce the gap that currently exists between the two research communities.

References

[1] E. Amir and S. Russell, Logical filtering. *Proc. of the IJCAI-03 Conference*, pages 75–82, Acapulco, 2003.

[2] F. Bacchus and F. Kabanza, Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[3] F., Bacchus, J. Halpern, and H. Levesque, Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, 111, 1999, 171–208.

[4] C. Baral, Reasoning about actions: non-deterministic effects, constraints, and qualification. *Proc. of the IJCAI-95 Conference*, 2017–2026, Montreal, 1995.

[5] C. Baral, L. Floriano, A. Hardesty, D. Morales, M. Nogueira, T. C. Son, From theory to practice: the UTEP robot in the AAAI 96 and AAAI 97 robot contests. *Proc. of the Agents-98 Conference,* 32–38, 1998.

[6] C. Baral and T. Son, Relating theories of actions and reactive control. *Electronic Transactions of Artificial Intelligence*, 2(3-4):211-271, 1998.

[7] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking. *Proc. of the IJCAI-01 Conference*, 473–478, Seattle, 2001.

[8] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, Decision-theoretic, high-level agent programming in the situation calculus. *Proc. of the AAAI-00 Conference*, pages 355–362, 2000.

[9] W. Burgard, A. B. Cremers, D. Fox, D Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, S. Thrun, Experiences with an interactive museum tour-guide robot. *Artificial Intellelligence* 114(1-2):3-55, 1999.

[10] T. Bylander, The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[11] A. Carbone, A. Finzi, A. Orlandini, F. Pirri, and G. Ugazio, Augmenting situation awareness via model-based control in rescue robots. *Proc. of IROS-2005 Conference* Edmonton, Canada, 2005.

[12] M. Dastani, F. de Boer, F. Dignum, W. van der Hoek, M. Kroese, J.-J. Meyer, Programming the deliberation cycle of cognitive robots. *Proc. of the 3rd International Cognitive Robotics Workshop*, Edmonton, 2002.

[13] G. de Giacomo, L. Iocchi, D. Nardi, R. Rosati, Moving a robot: the KR & R approach at work. *Proc. of the KR-96 Conference*, 198–209, 1996.

[14] G. de Giacomo, L. Iocchi, D. Nardi, R. Rosati, Planning with sensing for a mobile robot. *Proc. of the ECP-97 Conference*,Toulouse, France. 1997.

[15] G. de Giacomo, and H. Levesque, Projection using regression and sensors. *Proc. of the IJCAI-99 Conference*, Stockholm, Sweden, August 1999, 160–165.

[16] G. de Giacomo, Y. Lespérance, H. Levesque, and S. Sardiña, On the semantics of deliberation in Indigolog. *Annals of Mathematics and Artificial Intelligence*, 41, 2–4, 2004, 259–299.

[17] G. de Giacomo, Y. Lespérance, and H. Levesque, ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121, 2000, 109–169.

[18] G. de Giacomo, R. Reiter, M. Soutchanski, Execution Monitoring of High-Level Robot Programs. *Proc. of the KR-98 Conference*, Trento Italy, 1998.

[19] R. Demolombe, R. and M Pozos Parra, A simple and tractable extension of situation calculus to epistemic logic. *Proc. of the ISMIS-2000 Conference*, 515–524, 2000.

[20] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and K. Wiklund, The WITAS Unmanned Aerial Vehicle Project. *Proc. ECAI-00*, Berlin, 747–755, 2000.

[21] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line decision-theoretic Golog for unpredictable domains. *Proc. of 27th German Conference on AI*, 322–336, 2004.

[22] A. Finzi and F. Pirri, Diagnosing failures and predicting safe runs in robot control. *Proc. of the Commonsense 2001 Conference,* 105–113. New York, 2001.

[23] A. Finzi and F. Pirri, Combining probabilities, failures and safety in robot control. *Proc. of the IJCAI-01 Conference*, Seattle, August 2001.

[24] A. Finzi and F. Pirri, Representing flexible temporal behaviors in the situation calculus. *Proc. of the IJCAI-05 Conference*, 436-441, 2005.

[25] A. Finzi, F. Pirri, M. Pirrone, and M. Romano, Autonomous mobile manipulators managing perception and failures. *Proc. of the Agents-01 Conference,* 196–201, Montreal 2001.

[26] M. Fichtner, A. Großmann, M. Thielscher, Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 57, 371–392, 2003.

[27] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain and H. Turner, Nonmonotonic causal theories, *Artificial Intelligence,* 153:49–104, 2004.

[28] R. Fikes and N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[29] M. Ghallab, D, Nau, and P. Traverso, *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.

[30] H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In Werner Horn, editor, *Proc. of the ECAI-2000 Conference*, pages 548–552, 2000.

[31] H. Grosskreutz and G. Lakemeyer, ccGolog: An action language with continuous change. *Logic Journal of the IGPL,* Oxford University Press, 2003.

[32] H. Grosskreutz and G. Lakemeyer, On-line execution of cc-Golog plans. *Proc. of the IJCAI-01 Conference*, 12–18, 2001.

[33] K. Hindriks, F. de Boer, W. van der Hoek, J.-J. Ch. Meyer, A formal semantics for an abstract agent programming language. *Proc. of the ATAL-97 Conference*, June 1998.

[34] K. Hindriks, F. de Boer, W. van der Hoek, J.-J. Ch. Meyer, Failure, monitoring and recovery in the agent language 3APL. *Proc. of the AAAI-98 Fall Symp. on Cognitive Robotics*, 68–75, 1998.

[35] J. Hintikka,, *Knowledge and Belief.* Cornell University Press, Ithaca, 1962.

[36] W. van der Hoek, J.J. Meyer, B. Linder, On agents that have the ability to choose. *Studia logica*, 66(1), 79-119, 2000.

[37] G. Lakemeyer, On sensing and off-line interpreting in GOLOG. In *Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter,* Springer, Berlin, 173-187, 1999.

[38] G. Lakemeyer, Relevance from an epistemic perspective, *Artificial Intelligence,* 97(1-2):137-167, 1997.

[39] Y. Lespérance, H. Levesque, F. Lin, R. Scherl, Ability and knowing how in the situation calculus. *Studia Logica*, 66, 165–186, October 2000.

[40] Y. Lespérance and H.-K. Ng, Integrating planning into reactive high-level robot programs. *Proc. of the Second International Cognitive Robotics Workshop,* Berlin, Germany, 49–54, 2000.

[41] H. Levesque, What is planning in the presence of sensing? *Proc. of AAAI-96 Conference*, Portland, OR, Aug. 1996, 1139–1146.

[42] H. Levesque and R. Reiter, Beyond planning. *AAAI Spring Symposium on Integrating Robotics Research*, Working notes, Palo Alto, CA, March 1998.

[43] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl, GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[44] V. Lifschitz, On the semantics of STRIPS. *Proc. of the 1986 Workshop Reasoning about Actions and Plans*, pages 1–9. Morgan Kaufmann, 1987.

[45] F. Lin, Embracing causality in specifying the indirect effects of actions. *Proc. of the IJCAI-95 Conference*, pages 1985–1991. Montreal, 1995.

[46] F. Lin and R. Reiter. How to progress a database II: The STRIPS connection. *Proc. the IJCAI-95 Conference*, 2001–2007, 1995.

[47] F. Lin and R. Reiter, State constraints revisited. *Journal of Logic and Computation,* 4(5):655-678, 1994.

[48] F. Lin and R. Reiter. Forget it! *Proc. of the AAAI Fall Symposium on Relevance,* New Orleans, USA, November 1994.

[49] F. Lin, R. Reiter, How to progress a database. *Artificial Intelligence*, 92(1–2):131–167, 1997.

[50] B. Linder, W. van der Hoek, J.J. Meyer, Formalizing motivational attitudes of agents: On preferences, goals and commitments. *Proc. of the ATAL-96 Conference*, 17–32, Berlin, 1996.

[51] Y. Liu, H. Levesque, Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions. *Proc. of the IJCAI-05 Conference*, Edinburgh, August 2005.

[52] J. McCarthy and P. Hayes, Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, pages 463–502. University of Edinburgh Press, 1969.

[53] D. McDermott, Robot planning. *AI Magazine* 13(2):55–79, 1992.

[54] R. Moore, A formal theory of knowledge and action. *Formal Theories of the Commonsense World*, Ablex, Norwood, NJ, 319–358, 1985.

[55] N. Nilsson, Shakey the robot. SRI Technical report, 1984.

[56] F. Bacchus and R. Petrick, Modeling an agent's incomplete knowledge during planning and execution. *Proc. of the KR-98 Conference*, Trento, Italy, 1998.

[57] M. Fox and D. Long, PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of AI Research*, 20:61–124, 2003.

[58] J. Pinto, Integrating discrete and continuous change in a logical framework. *Computational Intelligence, 14(1)*, 1997.

[59] F. Pirri and A. Finzi, An approach to perception in theory of actions: Part I. *Electronic Transaction on Artificial Intelligence,* 3(41):19–61, 1999.

[60] F. Pirri and M. Romano, A situation-Bayes view of object recognition based on symgeons. *Proc. of the Third International Cognitive Robotics Workshop*, Edmonton, 2002.

[61] F. F. Ingrand, R. Chatila, R. Alami and F. Robert, PRS: A high level supervision and control language for autonomous mobile robots. *Proc. Int. Conf. on Robotics and Automation*, 1996.

[62] Z. Pylyshyn, *Computation and Cognition: Toward a Foundation for Cognitive Science*. MIT Press, Cambridge, Massachusetts, 1984.

[63] A. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language. *Agents Breaking Away*, Springer-Verlag, 1996.

[64] R. Reiter, On closed world data bases. *Logic and Databases*, pages 55–76. Plenum Press, New York, 1987.

[65] R. Reiter, The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, New York, 1991.

[66] R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, Massachusetts, 2001.

[67] S. Ruman, *GOLOG as an Agent-Programming Language: Experiments in Developing Banking Applications.* M. Sc., Dept. of Computer Science, University of Toronto, January 1996.

[68] S. Sardina, *Deliberation in agent programming languages.* Ph. D. thesis, Dept. of Computer Science, University of Toronto, June 2005.

[69] S. Sardiña, *Indigolog: Execution of guarded action theories.* M. Sc. Thesis, Dept. of Computer Science, University of Toronto, April 2000.

[70] M. P. Shanahan, *Solving the Frame Problem,* MIT Press, 1997.

[71] M. P. Shanahan, A logical account of perception incorporating feedback and expectation. *Proc. of the KR-02 Conference*, 3–13, 2002

[72] M. P. Shanahan, Representing continuous change in the event calculus. *Proc. of the ECAI-90 Conference*, 1990.

[73] M. P. Shanahan, Noise and the Common Sense Informatic Situation for a Mobile Robot, *Proc. of the AAAI-96 Conference*, 1098–1103, 1996.

[74] M.P.Shanahan, Reinventing Shakey. In *Logic-Based Artificial Intelligence,* Jack Minker (Ed.), Kluwer Academic, 233–253, 2000.

[75] M. P. Shanahan, M. Witkowski, High-Level Robot Control Through Logic. *Proc. of the ATAL-2000 Conference*, 104–121, 2001.

[76] S. Shapiro, M. Pagnucco, Y. Lespérance, H. Levesque, Iterated belief change in the situation calculus. *Proc. of the KR-2000 Conference*, Breckenridge CO, April 2000, 527–538.

[77] S. Shapiro and M. Pagnucco, Iterated belief change and exogenous actions in the situation calculus. *Proc. of the ECAI-04 Conference*, 878–882, 2004.

[78] S. Shapiro, Y. Lesperance, H. Levesque, Goal change. *Proc. of the IJCAI-05 Conference*, Edinburgh, August 2005.

[79] R. Scherl, H. Levesque, Knowledge, action, and the frame problem. *Artificial Intelligence*, 144, 2003, 1–39.

[80] T. Son and C. Baral, Formalizing sensing actions – A transition function based approach. *Artificial Intelligence*, 125(1–2):19–91, 2001.

[81] M. Soutchanski, An on-line decision-theoretic golog interpreter. *Proc. of the IJCAI-01 Conference,* Seattle, Washington, 2001.

[82] M. Thielscher, Modeling actions with ramifications in nondeterministic, concurrent, and continuous domains - and a case study. *Proc. of the AAAI-00 Conference* 497-502, 2000.

[83] M. Thielscher, Representing the knowledge of a robot. *Proc. of the KR-2000 Conference*, Breckenridge, 109-120, 2000.

[84] M. Thielscher, FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4–5):533–565, 2005.

[85] M. Thielscher, *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Springer, 2005.

[86] S. Thrun, Robotic mapping: A survey. In G. Lakemeyer and B. Nebel (Eds.) *Exploring Artificial Intelligence in the New Millennium.* Morgan Kaufmann, 2002.

Mapping and meaning

Jim Little
Computer Science
University of British Columbia

I propose to discuss the state of the art in mapping and robot localization, both of which are sufficient to enable robots to understand where they are. Many tasks such as manipulation and search simply require relative localization.

The next challenge is to situate the activities and tasks of mobile robots in maps labeled with semantic information. The labels can be learned from annotated image databases. Spatial relations among objects provide clues towards identifying the kinds of activities that associate with sites in living and working spaces. Visual recognition is becoming more capable of recognizing generic objects so a robot can determine that it is in, for example, a kitchen and therefore will expect food preparation and cooking activities in that locale.

Places supply prior information about the events and actions that occur there. I will discuss how place categorization proceeds from object recognition and how it enables robots to act in cooperation with people at home and at work.

Collaborative activity and human-robot interaction

Kruijff, Geert-Jan M.
DFKI Saarbrucken

The tutorial will highlight issues in processing situated dialogue in human-robot interaction. We see such interaction as part of a larger collaborative activity. The human and the robot are engaged in a form of joint activity, and dialogue as (inter)action is part of that. This makes utterance meaning more than "just content." A robot needs to figure out what the utterance refers to, what new information it provides – but also, why somebody said it, and how the robot is supposed to act upon it. Processing and managing dialogue therefore needs to be integrated with modeling multi-agent beliefs and -intentions. We discuss the implications of environment uncertainty and -dynamics on how such multi-agent models need to be modeled and maintained. This leads to more general implications for cognitive architecture design. Particularly, we focus on the grounding of multi-agent models in various types of short- and long-term memories, and how that grounding can make it possible to use these models as interface between different types of deliberative cognitive processes like planning, dialogue, and reasoning. The tutorial illustrates the discussed approaches on several robot systems.

Learning functional object categories and event classes from video

Cohn, Anthony G.
University of Leeds

In this talk I present ongoing work at Leeds, in collaboration with Krishna Sridhar and David Hogg on inducing functional object categories from (qualitative) spatio-temporal descriptions of the participating objects. First we mine event classes from activity descriptions expressed using qualitative spatio-temporal graphs. We presume that events may overlap temporally, and have shared participants. I present two techniques for doing this: a bottup up level-wise technique, and a top down technique which presupposes an event generation model, and where the learning task consist of finding the best explanation of the observed activity under various assumptions. Good explanations are characterised by having few event classes, where each event class is compact (in terms of how similar its instance are) and by a notion we call "interactivity", which measures the extent to which an event consists of highly interacting objects, without "bystanders". Having formed these event classes, object categories can be formed by clustering those objects which take the same roles in a particular event. We have experimented with these techniques in two domains: a kitchen scenario, and aircraft turnovers. I also briefly mention a techniuque for robustly obtaining qualitative spatial relations from noisy video data using a trained HMM.

Acquiring scene models through human-robot interaction

Sven Wachsmuth
with Agnes Swadzba, Ingo Ltkebohle, Julia Peltason, Robert Haschke
University of Bielefeld, Germany

The success of human-robot interaction critically depends on aligned representations of the mutual intentions as well as the surrounding scene. This cannot be computed from purely bottom-up processing. Even if the general goal is clear for a human interaction partner, the variation of utterances how they would instruct a robot is extremely large. In terms of the visual perception of the environment we have to deal with previously unknown objects, places and complex scene structures including tables, chairs, shelfs, etc.

Thus, the goal of a cognitive robot is to learn as much as possible from the direct interaction with humans in a structured way. Most work in this field concentrates on the speech interpretation and gesture recognition side assuming that a propositional scene representation is available. Less work has been dedicated to the extraction of relevant scene structures that underlies these propositions. Psycholinguistic studies have shown that people mostly use clear hierarchical structures in the verbalization of spatial scene description. Objects are typically put in spatial relation if they have a common supporting structure or are in a direct supporting relation. This information can be exploited to hypothesize the underlying scene structure from verbal descriptions. We generate 3D plane hypotheses from scene objects referenced by the speaker. These define grouping relations between 3D planar patches that are extracted from depth images recorded by a Time-of-Flight camera (Swadzba et al. 2009). Using this scheme scene structures can be acquired through human-robot interaction on a semantic and functionally defined level. Without assuming any pre-known model of the specific room, we show that the system aligns its sensor-based room representation to a semantically meaningful representation typically used by the human descriptor.

Another aspect of such teaching scenarios is that untrained users have an insufficient expectation about successful human-robot interactions. Having the task to teach the robot, there is a large variability in verbalization behavior of laypersons. Thus, human-robot interaction is significantly improved if the system provides dialog structure and engages the human in an explanatory teaching scenario. We specifically target untrained users, who are supported by mixed-initiative interaction using verbal and non-verbal modalities. The principles of dialog structuring are based on a novelty detection and curiousity behavior of the robot on the one side and user clarification dialogs on the other side. In the scenario a person needs to teach object names and manipulation grips to the robot. System development is followed and interleaved with an interactive evaluation approach. The system is based on an extensible, event-based interaction architecture. It was shown in a video study that users benefit from the provided dialog structure resulting in a predictable and successful human-robot

interaction (Ltkebohle et al. 2009).

References

Ltkebohle, I., Peltason, J., Schillingmann, L., Elbrechter, C., Wrede, B., Wachsmuth, S., Haschke, R. , “The Curious Robot - Structuring Interactive Robot Learning”, International Conference on Robotics and Automation, Kobe, Japan, IEEE, 14/05/2009.

Swadzba, A., Vorwerg, C., Wachsmuth, S., Rickheit, G. , “A Computational Model for the Alignment of Hierarchical Scene Representations in Human-Robot Interaction”, International Joint Conference on Artificial Intelligence, Pasadena, CA, USA, AAAI Press, pp. 1857-1863, 14/07/2009.

Attending to Motion: an object-based approach

Anna Belardinelli

Abstract Visual attention is the biological mechanism allowing to turn mere sensing into conscious perception. In this process, object-based modulation of attention provides a further layer between low-level space/feature-based region selection and full object recognition. In this context, motion is a very powerful feature, naturally attracting our gaze and yielding rapid and effective shape distinction.
Moving from a pixel-based account of attention to the definition of proto-objects as perceptual units labelled with a single saliency value, we present a framework for the selection of moving objects within cluttered scenes. Through segmentation of motion energy features, the system extracts coherently moving proto-objects defining them as consistently moving blobs and produces an object saliency map, by evaluating bottom-up distinctiveness of each object candidate with respect to its surroundings, in a center-surround fashion.

1 Introduction

Cognitive architecture for autonomous robotics often rely on the capability to deal with objects, act upon them, recognize scenes and partners and hence behave properly in a given situation. These higher level cognitive functions can take place only if the flow of the huge and multimodal information stream coming from the sensory system is firstly processed and filtered by an attention mechanism, which extracts relevant patterns and conveniently codes and prioritizes what has to be further processed.

Although most computational models of attention are location-based, there is growing evidence for an object-based account of attention [21]. In his Theory of Visual Attention [3], Bundesen defines matematically how our visual system could

Anna Belardinelli
Cognitive Interaction Technology Center of Excellence (CITEC), Bielefeld University, Universitätsstrasse 25, Bielefeld, Germany e-mail: anna.belardinelli@cit-ec.uni-bielefeld.de

assess top-down relevance of each object in the stimulus. That is, proto-objects (or perceptual files, which consist of selected regions) are the basic units of attention, upon which a priority value is computed. Objects are then fed into a WTA network, providing access to working memory for those winning the race. Such proto-objects can be defined in a flexible way and upon different features.

Research on visual attention, and modelling thereof, has concentrated in the past decades mostly on static stimuli, characterized through a wealth of features, accounting for bottom-up attentional capture and accordance with task-related requirements. Yet, we live in a highly dynamic world, populated with moving things, which call for a selective mechanism much more compellingly than static objects do. Early detection and selection of the most salient kind of motion can sometimes make the difference in the struggle for survival. Even simple insects do have some form of motion perception [8], but usually quite limited color vision. In a very cluttered scene, moving objects are supposed to attract our gaze very effectively, as shown by [4], where motion contrast accounts for most of the fixations. On a neurophysiological level, motion information is indeed processed even along a different, more direct pathway, the dorsal pathway, as opposed to other features needed for object recognition [12]. If attending to static objects is the prerequisite of perception for action (like searching for a cup and grasp for it), attending to motion fosters perception for reaction and interaction, being tied to events evolving in time and triggering our response (such as an approaching danger or person). Embedding motion in a visual attention model would then move into the direction suggested by [23] of considering gaze orienting in real-world environments instead of end up with a model of picture viewing.

Without disregarding the importance of the deployment of attention to static features, our model builds upon a novel approach for extracting and prioritizing moving objects in a scene. In a previous work [2], we introduced a basic framework for producing motion saliency maps from spatiotemporal filtering. That model was not broadly tuned in the frequency domain and produced a pixel-based saliency map. Motion is a quite distinctive property which naturally induces segmentation of the scene within foreground and background (see [16] for an application to background subtraction), hence provides a more straightforward extraction of object units than color [22] or edge features [17], or spreading of activation around a salient location [24].

As usual when designing an attention architecture, in the case of attending to motion the problem is to identify and prioritize salient regions, namely, not just detecting moving objects but defining which one requires to be first attended. Saliency is not intrinsic in the location nor in the object but it is defined relatively to its surround, in a contrast based way, and according to relevance to the task. In this paper, we try to bring all these ideas together and extend our model to account for multiscale motion, proto-object extraction and object saliency evaluation. Saliency is given by means of center-surround computations both on a location-based and an object-based level. Relevance is given by tuning the model according to the given task. Proto-objects (in the following termed objects) are defined as blobs of consistent motion energy and coherent direction. Objects standing out from the surround-

ing with respect to amount of energy or direction are hence given larger saliency. In the next sections, we describe the components of our system and present some results. Section 2 explains our implementation of the energy model [1] for motion perception, Section 3 proposes the definition and characterization of moving proto-objects and how to compute their saliency. Finally, Section 4 shows some experimental simulations and results.

2 Motion feature extraction and prioritization

We extend the implementation of the energy model for coherent motion sensing by [1] introduced in [2]. The basic idea is that coherent motion can be selected inside an intensity frame buffer by filtering in the oriented edges and bars, left by objects moving in the spatiotemporal volume. Instead of just one couple of Gabor filters in quadrature for extracting right/left-ward and up/down-ward motion from $x-t$ and $y-t$ planes respectively, we designed a Gabor filter bank to extract motion information at different spatio-temporal scales (frequency bands) and velocities (filter orientations), trying to sample most of the spatiotemporal frequency domain included in the window $u,v \in [0,0.5]$, to comply with the sampling theorem. That is, we code each voxel in a Gabor space, according to its oriented energy response, analogously to the coding suggested for modeling our visual system [10]. Gabor filters have been long known to resemble orientation sensitive receptive fields present in our visual cortex and to represent band-pass functions conveniently localized both in the space and in the frequency domain [6]. This is still valid in the spatiotemporal domain, as measured by [7] in the receptive fields of simple cells in V1 and as obtained via ICA (Independent Component Analysis) computation on video sequences by [13]. In both studies, resulting receptive fields resemble 3D Gabor filters (whose central slices are 2D Gabor filters as well) at different orientations and frequencies.

Basically, given a frame buffer $\mathscr{B}$, we filter any vertical (column-temporal dimensions) or horizontal (row-temporal dimensions) plane $I(s,t)$ in $\mathscr{B}$ with every filter $G_{\theta,f}$ in the bank, in its odd (superscript o) and even (superscript e) component:

$$E_{\theta,f}(s,t) = (G^{o}_{\theta,f}(s,t) \star I(s,t))^2 + (G^{e}_{\theta,f} \star I(s,t))^2 \quad (1)$$

where $s = \{x,y\}$, $f = \{0.0938, 0.1875, 0.3750\}$ (the max spanned frequency is 0.5 cyc/pixel, the frequency bandwidth is 1 octave), $\theta = \{\pi/6, \pi/3, 2/3\pi, 5/6\pi\}$. That is, we designed a filter bank with 4 orientations ($\theta = 0, \pi/2$ were left out, as corresponding to static or flickering edges) and 3 frequency bands.

From combination of opponent filter pairs (i.e filters with same slope but opposite orientation, θ and $(\pi-\theta)$) we can extract a measure of direction-selective energy at a specific velocity. For instance, in our case right-sensitive filters have $\theta_r = \{\pi/6, \pi/3\}$, while left-sensitive filters have $\theta_l = \{(\pi-\pi/6), (\pi-\pi/3)\}$. A

measure of the total rightward (leftward) energy at a specific frequency can hence be obtained by summing rightward (leftward) energy accross velocities:

$$R_f = \sum_i \Big| \frac{E_{\theta_{r_i},f} - E_{\theta_{l_i},f}}{E_{\theta_{r_i},f} + E_{\theta_{l_i},f}} \Big|_{\geq 0} \quad L_f = \sum_i \Big| \frac{E_{\theta_{r_i},f} - E_{\theta_{l_i},f}}{E_{\theta_{r_i},f} + E_{\theta_{l_i},f}} \Big|_{\leq 0} \tag{2}$$

where the $|\cdot|$ operator selects points greater/less than zero, corresponding to rightward/leftward motion. The same can be done for upwards (downwards) energy computation, by taking $s = y, \theta_u = \theta_r$ and $\theta_d = \theta_r$.
In this way we obtain 4 feature volumes R, L, U, D at different frequencies.

Subsequently, we operate a first attentional processing by applying normalization and center-surround operators to the frames of each feature buffer. Due to receptive field center-surround interactions, indeed, ganglion cells are usually described as firing more strongly whenever a central location is more contrasted with respect to its surroundings. Again, this holds in the motion domain as well, as shown by [19]: locations displaying different motion in terms of energy module or direction pop out from the surroundings and are enhanced in the saliency map. Center-surround inhibition is usually obtained via accross-scale differences [15] or center-neighborhood differences at the same scale [11]. We chose the second way, as faster due to the use of integral images. At the same time, feature maps need to be normalized to the same range and weighted according to the number of occurring local maxima, so that a feature map with many activation peaks is given less weight than one with few peaks. This can be realized in a biological plausible manner by iteratively filtering the feature frames with a DoG (Difference of Gaussians) filter and taking each time just the non negative values [14]. We then compose horizontal and vertical features to obtain a measure of horizontal and vertical energy and sum accross frequencies:

$$E_h = \sum_f (\mathcal{N}(CS(R_f)) + \mathcal{N}(CS(L_f))) \quad E_v = \sum_f (\mathcal{N}(CS(U_f)) + \mathcal{N}(CS(D_f))) \tag{3}$$

Here $\mathcal{N}(\cdot)$ and $CS(\cdot)$ denote the normalization and center-surround operator, respectively, which are applied to each $x - y$ frame of the feature buffers.

To illustrate the effectiveness of our procedure we use a purely bottom up synthetic stimulus, depicted in Fig.1a. The sequence (256 x 256 x 5) displays nine squares at random positions moving downwards at 1 $pixel/frame$ velocity and just one square moving rightwards at the same velocity, representing the oddball (marked by a red circle). In the horizontal feature map (Fig. 1b) correctly just the oddball is shown, while in the vertical feature map (Fig. 1c) just the vertical moving dots are shown. Due to normalization these latter have less energy (see colorbar), albeit moving at the same velocity as the horizontally moving one.

E_h and E_v can be regarded as the projection on the x and y axes of the salient motion energy present in the frame buffer. Hence from these components we can achieve, for every voxel, magnitude and phase of the salient energy:

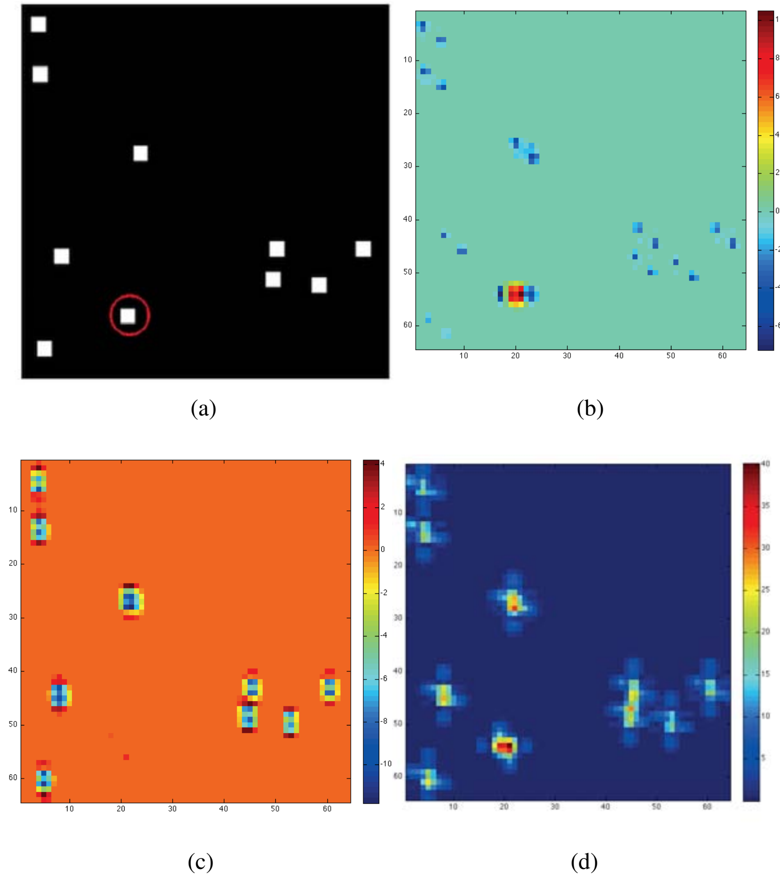


Fig. 1: Application of the salient energy extraction framework to a synthetic display (a) containing a pop-out object, represented by the red-circled square, moving horizontally, while the other squares move vertically. (b), the horizontal motion feature map and (c) the vertical motion feature map are shown, both at f=0.3750. (d), the temporal average of the module of salient energy, achieved by merging horizontal and vertical energy at different frequencies.

$$|E(x,y,t)| = \sqrt{E_h(x,y,t)^2 + E_v(x,y,t)^2} \tag{4}$$

$$\angle E(x,y,t) = \arctan(E_v(x,y,t)/E_h(x,y,t)) \tag{5}$$

A saliency map derived from magnitude map be seen in Fig.1d, obtained by averaging the $|E|$ frame buffer along time. Top-down modulation at this level can be implemented by selecting the filter parameters (number of orientations, number of frequency bands, orientation and frequency bandwidths) according to the current task. In this way, one can decide to attend just to a particular direction of movement, to a particular scale of objects or to a particular velocity range.

3 Proto-object formation and saliency evaluation

In the previous section, we have shown how to obtain a saliency map enhancing relevant motion zones. Such a map is yet pixel-based and, as said in the introduction, an object-based map would best help subsequent processing for object recognition and action selection. We need to evaluate the saliency of an object with respect to its entirety and with respect to the surrounding background, not just by considering each single pixel it is composed of. Indeed, even if motion processing attains to the dorsal, or ”where”-, pathway, nevertheless attentional processes can modulate segregation and grouping of the visual input into ”object tokens” across both pathways [20].

To this end, we extracted proto-object patches defined as blobs of consistent motion in terms of module and direction. As the Gestalt law of common fate states, points moving with similar velocity and direction are perceptually grouped together in a single object. A simple segmentation on the module map would not be sufficient, since adjacent objects moving in different directions would be merged. Hence, we threshold the temporally averaged magnitude map $|E(x,y)|$ to discard points with too low energy and apply the mean shift algorithm to the phase of the remaining points in the average phase map $\angle E(x,y)$. The mean shift algorithm is a kernel-based mode-seeking technique, broadly used for data clustering and segmentation [5]. Being non-parametric, it has the advantage that it does not need the number of clusters to be specified previously. We cluster in this way objects with a certain amount of energy according to their direction. Application of this procedure to the synthetic stimuli presented above gave the results presented in Fig.2a. The vertically moving squares are assigned to a class while the horizontally moving square belongs to a different class.

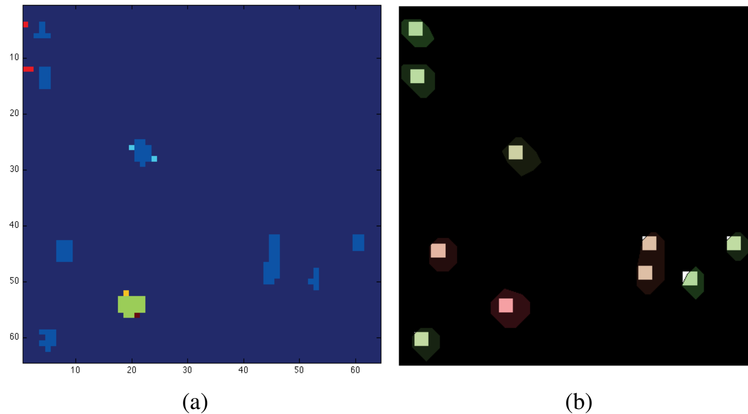
Once we have labelled regions we can extract the object convex hulls by means of morphological operations and can compute their saliency. Again, we define object saliency as proportional to motion contrast in terms of module and orientation, in a center-surround fashion. Given an object **o**, defined by its bounding box, and given its surround $N(\mathbf{o})$ of size proportional to the area of **o**, similarly to [17], we have:

$$S_{mag}(\mathbf{o}) = \langle |E(x,y)| \rangle_{(x,y)\in(o)} - \langle |E(x,y)| \rangle_{(x,y)\in N((o))} \tag{6}$$

where the $\langle\cdot\rangle$ operator computes the mean of the points in the subscript set.

For orientation saliency, since some non rigid objects can display more than one direction but still a dominating general direction, we compute the histograms of the orientations of the object **o**, weighted according to the energy module, as:

$$h(i) = \sum_{\angle E(x,y)\in i} |E(x,y)| \tag{7}$$



(a) (b)

Fig. 2: Object segmentation and prioritization. (a), the result of the mean shift segmentation on directions relative to salient energy is displayed. Each cluster is denoted by a different color. (b), convex hull patches corresponding to segmented objects are superimposed to the original frame: color is determined by saliency, with the least salient object having RGB=(0, 1, 0) and the most salient being displayed in pure red with RGB=(1, 0, 0).

where i represents the i-th bin. In so doing, the more likely orientations are the ones relative to high energy points. Orientation saliency is hence given by the similarity between the orientation distributions of the object and of its surround. Similarity is evaluated through the Bhattacharyya distance:

$$S_{or}(\mathbf{o}) = 1 - \sum_i \left(\sqrt{h_o(i) * h_{N(o)}(i)}\right) \tag{8}$$

Hence, the more the orientation distribution of the object differs from that of the surrounding, the greater the orientation saliency.

Finally, the overall saliency of the object is calculated as linear combination of the two components:

$$S(\mathbf{o}) = \alpha S_{mag}(\mathbf{o}) + \beta S_{or}(\mathbf{o}) \tag{9}$$

Both S_{mag} and S_{or} are normalized to the interval $[0, 1]$. α and β are taken equal to 0.5 in the case of pure bottum-up selection, but can be top-down biased for task-driven selection.

In Fig.2b, the segmented patches with color intensity proportional to the overall saliency are superimposed on the original frame. The oddball is correctly shown as the object with the highest saliency, the most reddish.

4 Experiments and discussion

Having tested the effectiveness of our framework on synthetic stimuli, where the pop-out target can be easily and univocally identified by every subject, we made some experiments with real world sequences. In particular, we took some sequences from the Getty Image footage [1], those taken with fixed camera and displaying multiple moving objects. In a crowded scene, indeed, such as a station or a crossroad (see Fig. 3 and 4), there is a wealth of moving objects competing for attention capture and therefore a prioritization and selection mechanism is extremely useful. In the experiments depicted in Fig. 3 and 4, it can be noticed how differently moving objects even very close to each other can be discriminated according to their distinctiveness from other motion patterns in the surroundings. Since the final saliency is evaluated on the whole object region, it is not said that the object containing the most salient pixel is the most salient object too.

The presented framework can be tuned and refined in a number of ways to make it more or less selective and task-oriented. A major limitation, at the moment, is the constraint of stationary camera. This limits its current biological plausibility, since humans are able to discriminate scene motion from ego-motion when moving the head or the body, due to the Vestibular-Ocular Reflex (VOR) present in our visual system. Similarly, this limit can be overcome by applying stabilization techniques to the buffer frame, or modelling the motion distribution of the background and applying background subtraction as in [16].

The main novelty of our system is the definition of moving proto-objects which is related to the their amount of motion and direction distinctiveness. We have shown how this approach can successfully select and prioritize relevant motion within a crowded scene. This is based on low-level processing and relies on the extraction of coherent motion in different directions. Further higher-level processing will have to be combined with specific task descriptions and a more elaborated description of motion patterns in terms of frequency and spatiotemporal signatures. Interesting issues still to be investigated are the temporal scale and resolution that are needed to recognize these patterns (we arbitrarily took a 5 frames temporal span for computational needs) and how far such a system can get without object continuity and indexing [18].

References

1. Adelson, E.H., Bergen, J.R.: Spatiotemporal energy models for the perception of motion. J. Opt. Soc. Am. A **2**(2), 284–299 (1985)
2. Belardinelli, A., Pirri, F., Carbone, A.: Motion saliency maps from spatiotemporal filtering. Attention in Cognitive Systems pp. 112–123 (2009)
3. Bundesen, C.: A theory of visual attention. Psychological review **97**(4), 523–547 (1990)

[1] http://www.gettyimages.com/

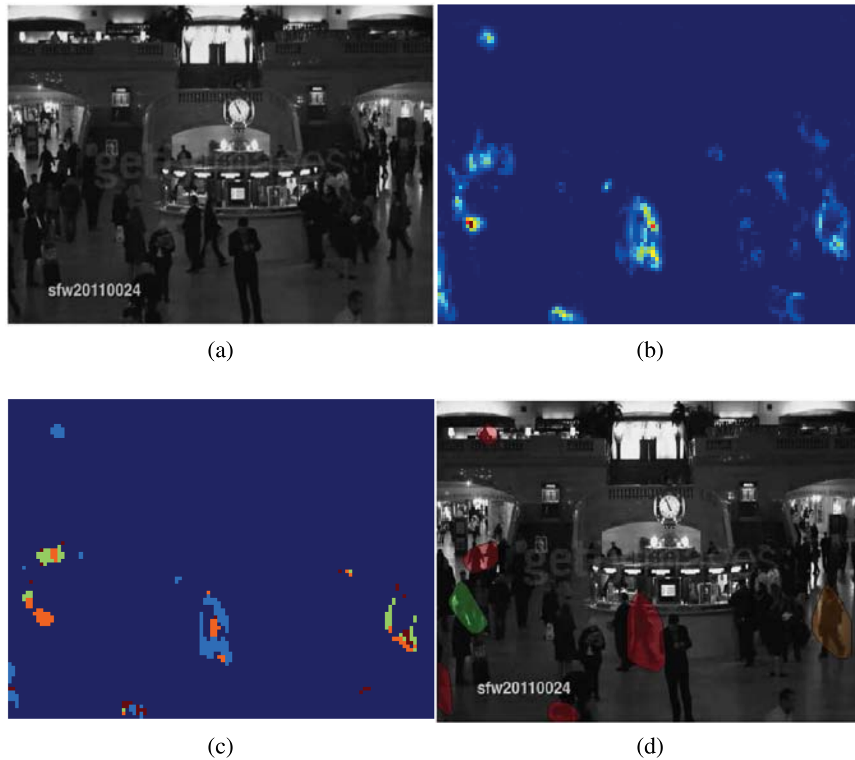


Fig. 3: Object-based saliency selection applied to a real world sequence. (a), an original frame. (b) the temporal average of the energy module. (c), the segmented phase map and (d) objects with their saliency are shown.

4. Carmi, R., Itti, L.: Visual causes versus correlates of attentional selection in dynamic scenes. Vision Research **46**(26), 4333–4345 (2006)
5. Comaniciu, D., Meer, P.: Mean shift: a robust approach toward feature space analysis. Pattern Analysis and Machine Intelligence, IEEE Transactions on **24**(5), 603–619 (2002)
6. Daugman, J.G.: Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. J. Opt. Soc. Am. A **2**(7), 1160–1169 (1985)
7. DeAngelis, G.C., Ohzawa, I., Freeman, R.D.: Spatiotemporal organization of simple-cell receptive fields in the cat's striate cortex. i. general characteristics and postnatal development. Journal of neurophysiology **69**(4), 1091–1117 (1993)
8. Egelhaaf, M., Borst, A., Reichardt, W.: Computational structure of a biological motion-detection system as revealed by local detector analysis in the fly's nervous system. J. Opt. Soc. Am. A **6**(7), 1070–1087 (1989)
9. Field, D.J.: Relations between the statistics of natural images and the response properties of cortical cells. J Opt Soc Am A **4**(12), 2379–2394 (1987)
10. Frintrop, S., Klodt, M., Rome, E.: A real-time visual attention system using integral images. In: Proceedings of the 5th International Conference on Computer Vision Systems (2007)
11. Goodale, M.A., Milner, A.D.: Separate visual pathways for perception and action. Trends in neurosciences **15**(1), 20–25 (1992). URL http://view.ncbi.nlm.nih.gov/pubmed/1374953

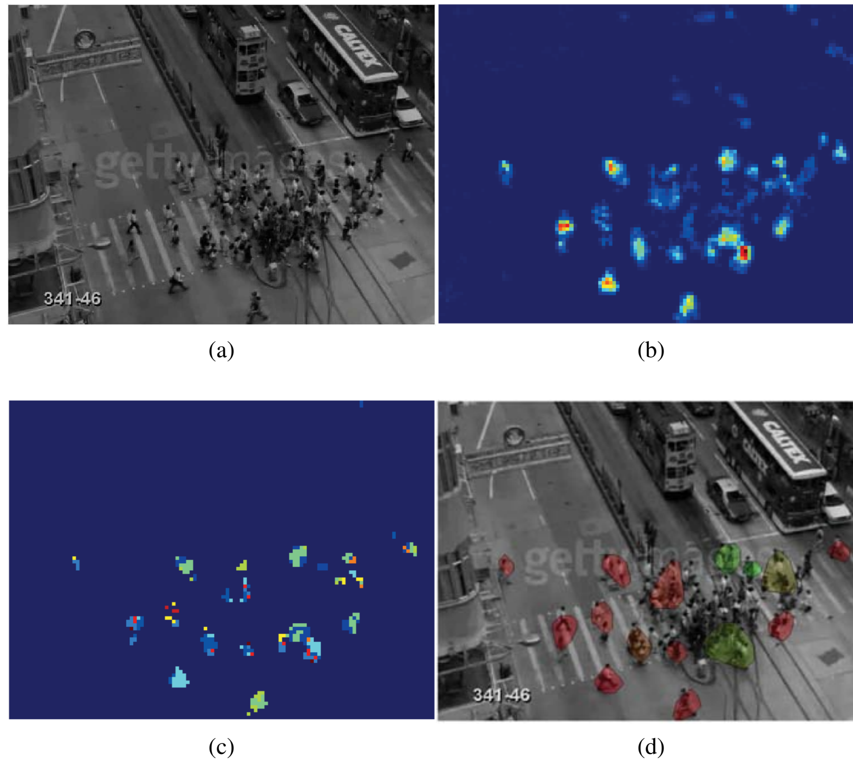


Fig. 4: Object-based saliency selection applied to a second real world sequence. (a), an original frame. (b) the temporal average of the energy module. (c), the segmented phase map and (d) objects with their saliency are shown.

12. van Hateren, J.H., Ruderman, D.L.: Independent component analysis of natural image sequences yields spatio-temporal filters similar to simple cells in primary visual cortex. Proceedings: Biological Sciences **265**(1412), 2315–2320 (1998)
13. Itti, L., Koch, C.: Feature combination strategies for saliency-based visual attention systems. Journal of Electronic Imaging **10**(1), 161–169 (2001)
14. Itti, L., Koch, C., Niebur, E.: A model of saliency-based visual attention for rapid scene analysis. IEEE Transactions on Pattern Analysis and Machine Intelligence **20**(11), 1254–1259 (1998)
15. Mahadevan, V., Vasconcelos, N.: Spatiotemporal saliency in dynamic scenes. IEEE Transactions on Pattern Analysis and Machine Intelligence **32**, 171–177 (2009)
16. Orabona, F., Metta, G., Sandini, G.: A proto-object based visual attention model. In: Attention in Cognitive Systems. Theories and Systems from an Interdisciplinary Viewpoint, pp. 198–215 (2008)
17. Pylyshyn, Z.W.: Visual indexes, preconceptual objects, and situated vision. Cognition **80**(1-2), 127–158 (2001)
18. Rosenholtz, R.: A simple saliency model predicts a number of motion popout phenomena. Vision Research **39**(19), 3157 – 3163 (1999)
19. Schneider, W.X.: VAM: A neuro-cognitive model for visual attention control of segmentation, object recognition, and space-based motor action. Visual Cognition **2**(2-3), 331–376 (1995)

20. Scholl, B.J.: Objects and attention: the state of the art. Cognition **80**(1-2), 1–46 (2001)
21. Sun, Y., Fisher, R., Wang, F., Gomes, H.M.: A computer vision model for visual-object-based attention and eye movements. Computer Vision and Image Understanding **112**(2), 126–142 (2008)
22. Tatler, B.: Current understanding of eye guidance. Visual Cognition pp. 777–789 (2009)
23. Walther, D., Koch, C.: Modeling attention to salient proto-objects. Neural Networks **19**(9), 1395 – 1407 (2006)

Plan recognition in a situation calculus-based agent programming framework

Yves Lesperance, Alexandra Goultiaeva, and Joshua McClymont
Dept. of Computer Science and Engineering, York University
and Dept. of Computer Science, University of Toronto

Plan recognition is capability that can be useful in many contexts, such as producing helpful computer interfaces, monitoring and assisting cognitively impaired people, and detecting suspicious behavior. In this talk, I will present a formal model of plan recognition for inclusion in a cognitive agent programming framework. The model is based on the Situation Calculus and the ConGolog agent programming language. This provides a very rich plan specification language. Our account supports incremental recognition, where the set of hypotheses about matching plans is progressively filtered as more actions are observed. This is specified using a transition system account. The model also supports hierarchically structured plans and recognizes subplan relationships. We will also discuss work in progress on developing a version of the framework that uses a Bayesian approach to compute the probability of matching plan hypotheses. We will present an application of this to building ”smarter” non-player characters in video games.

Coming up With Good Excuses: What to do When no Plan Can be Found

Moritz Göbelbecker and **Thomas Keller**
and **Patrick Eyerich** and **Michael Brenner** and **Bernhard Nebel**
University of Freiburg, Germany
{goebelbe, tkeller, eyerich, brenner, nebel}@informatik.uni-freiburg.de

Abstract

When using a planner-based agent architecture, many things can go wrong. First and foremost, an agent might fail to execute one of the planned actions for some reasons. Even more annoying, however, is a situation where the agent is *incompetent*, i.e., unable to come up with a plan. This might be due to the fact that there are principal reasons that prohibit a successful plan or simply because the task's description is incomplete or incorrect. In either case, an explanation for such a failure would be very helpful. We will address this problem and provide a formalization of *coming up with excuses* for not being able to find a plan. Based on that, we will present an algorithm that is able to find excuses and demonstrate that such excuses can be found in practical settings in reasonable time.

Introduction

Using a planner-based agent architecture has the advantage that the agent can cope with many different situations and goals in flexible ways. However, there is always the possibility that something goes wrong. For instance, the agent might fail to execute a planned action. This may happen because the environment has changed or because the agent is not perfect. In any case, recovering from such a situation by recognizing the failure followed by replanning is usually possible (Brenner and Nebel 2009).

Much more annoying than an execution failure is a failure to find a plan. Imagine a household robot located in the living room with a locked door to the kitchen that receives the order to tidy up the kitchen table but is unable to come up with a plan. Better than merely admitting it is *incompetent* would be if the robot could provide a good *excuse* – an explanation of *why* it was not able to find a plan. For example, the robot might recognize that if the kitchen door were unlocked it could achieve its goals.

In general, we will adopt the view that an excuse is a counterfactual statement (Lewis 1973) of the form that a small change of the planning task would permit the agent to find a plan. Such a statement is useful when debugging a domain description because it points to possible culprits that prevent finding a plan. Also in a regular setting a counterfactual explanation is useful because it provides a hint for where to start when trying to resolve the problem, e.g., by asking for help from a human or exploring the space of possible repair actions.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

There are many ways to change a planning task so that it becomes possible to generate a plan. One may change

- the goal description,
- the initial state, or
- the set of planning operators.

Obviously, some changes are more reasonable than others. For example, weakening the goal formula is, of course, a possible way to go. We would then reduce the search for excuses to over-subscription planning (Smith 2004). However, simply ignoring goals would usually not be considered as an excuse or explanation.

On the other hand, changing the initial state appears to be reasonable, provided we do not make the trivial change of making goal atoms true. In the household robot example, changing the state of the door would lead to a solvable task and thus give the robot the possibility to actually realize the reasons of its inability to find a plan.

In some cases, it also makes sense to add new objects to the planning task, e.g., while the robot is still missing sensory information about part of its environment. Thus, we will consider changes to the object domain as potential changes of the task, too. Note that there are also situations in which removing objects is the only change to the initial state that may make the problem solvable. However, since these situations can almost always be captured by changing those objects' properties, we ignore this special case in the following.

Finally, changing the set of planning operators may indeed be a "better" way, e.g., if an operator to unlock the door is missing. However, because the number of potential changes to the set of operators exceeds the number of changes to the initial state by far, we will concentrate on the latter in the remainder of the paper, which also seems like the most intuitive type of explanation.

The rest of the paper is structured as follows. In the next section, we introduce the formalization of the planning framework we employ. After that we sketch a small motivating example. Based on that, we will formalize the notion of excuses and determine the computational complexity of finding excuses. On the practical side, we present a method

that is able to find good excuses, followed by a section that shows our method's feasibility by presenting empirical data on some IPC planning domains and on domains we have used in a robotic context. Finally, we discuss related work and conclude.

The Planning Framework

The planning framework we use in this paper is the ADL fragment of PDDL2.2 (Edelkamp and Hoffmann 2004) extended by multi-valued fluents as in the SAS^+ formalism (Bäckström and Nebel 1995) or functional STRIPS (Geffner 2000).[1] One reason for this extension is that modeling using multi-valued fluents is more intuitive. More importantly, changing fluent values when looking for excuses leads to more intuitive results than changing truth values of Boolean state variables, since we avoid states that violate implicit domain constraints. For example, if we represent the location of an object using a binary predicate $at(\cdot,\cdot)$, changing the truth value of a ground atom would often lead to having an object at two locations simultaneously or nowhere at all. The domain description does not tell us that, if we make a ground atom with the *at* predicate true, we have to make another ground atom with the identical first parameter false. By using multi-valued fluents instead, such implicit constraints are satisfied automatically.

Of course, our framework can be applied to any reasonable planning formalism, since it is simply a matter of convenience to have multi-valued fluents in the language. This being said, a **planning domain** is a tuple $\Delta = \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}, \mathcal{O} \rangle$, where

- $\mathcal{T}$ are the **types**,
- $\mathcal{C}_\Delta$ is the set of **domain constant symbols**,
- $\mathcal{S}$ is the set of **fluent** and **predicate symbols** with associated arities and typing schemata, and
- $\mathcal{O}$ is the set of **planning operators** consisting of preconditions and effects.

A **planning task** is then a tuple $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, where

- Δ is a planning domain as defined above,
- $\mathcal{C}_\Pi$ is a set of **task-dependent constant symbols** disjoint from $\mathcal{C}_\Delta$,
- s_0 is the description of the **initial state**, and
- s^* is the **goal** specification.

The initial state is specified by providing a set s_0 of ground atoms, e.g., `(holding block1)` and ground fluent assignments, e.g., `(= (loc obj1) loc2)`. As usual, the description of the initial state is interpreted under the *closed world assumption*, i.e., any logical ground atom not mentioned in s_0 is assumed to be false and any fluent not mentioned is assumed to have an undefined value. In the following sections we assume that $\mathcal{S}$ contains only fluents and no predicates at all. More precisely, we will treat predicates as fluents with a domain of $\{\bot, \top\}$ and a default value of $\bot$ instead of unknown.

The goal specification is a closed first-order formula over logical atoms and fluent equalities. We say that a planning task is **solvable** iff there is a plan Ψ that transforms the state described by s_0 into a state that satisfies the goal specification s^*.

Sometimes we want to turn an initial state description into a (sub-)goal specification. Assuming that plan Ψ solves the task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, by $nec(s_0, \Psi, s^*)$ we mean the formula that describes the setting of the fluents necessary for the correct execution of Ψ started in initial state s_0 leading to a state satisfying s^*. Note that $s_0 \models nec(s_0, \Psi, s^*)$ always holds.

Motivating Examples

The motivation for the work described in this paper mostly originated from the DESIRE project (Plöger et al. 2008), in which a household robot was developed that uses a domain-independent planning system. More often than not a sensor did not work the way it was supposed to, e.g., the vision component failed to detect an object on a table. If the user-given goal is only reachable by utilizing the missing object, the planning system naturally cannot find a plan. Obviously, thinking ahead of everything that might go wrong in a real-life environment is almost impossible, and if a domain-independent planning system is used, it is desirable to also realize flaws in a planning task with domain-independent methods. Furthermore, we wanted the robot to not only realize that something went wrong (which is not very hard to do after all), but it should also be able to tell the user what went wrong and why it couldn't execute a given command.

However, not only missing objects may cause problems for a planning system. Consider a simple planning task on the KEYS-domain, where a robot navigates between rooms through doors that can be unlocked by the robot if it has the respective key (see Fig. 1). The goal of such a task is to have the robot reach a certain room, which in this example is $room_1$.

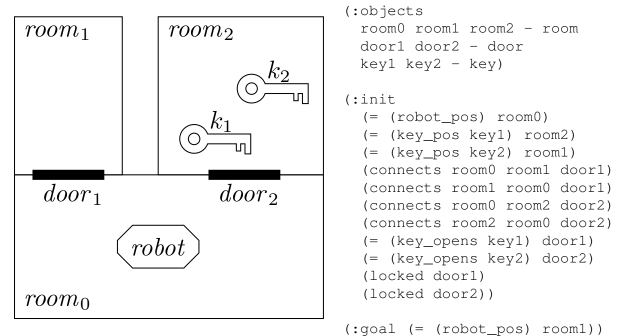


Figure 1: Unsolvable task in the *Keys* domain with corresponding PDDL code.

Obviously there exists no plan for making the robot reach its goal. What, however, are the *reasons* for this task being

[1]Multi-valued fluents have been introduced to PDDL in version 3.1 under the name of "object fluents".

unsolvable? As we argued in the introduction, the answer to this question can be expressed as a counterfactual statement concerning the planning task. Of course, there are numerous ways to change the given problem such that a solution exists, the easiest one certainly being to already have the goal fulfilled in the initial state. An only slightly more complicated change would be to have the door to $room_1$ state) in which case the robot could directly move to its destination, or have the robot already carry the key to that door (changing the value of `(key_pos key1)` from `room2` to `robot`) or even a new one (adding an object of type key with the required properties), or simply have one of the keys in $room_0$ (e.g. `(= (key_pos key1) room0)`).

Having multiple possible excuses is, as in this case, rather the rule than the exception, and some of them are more reasonable than others. So, the following sections will answer two important questions. Given an unsolvable planning task, *What is a good excuse?* and *How to find a good excuse?*

Excuses

As spelled out above, for us an excuse is a change in the initial state (including the set of objects) with some important restrictions: We disallow the deletion of objects and changes to any fluents that could contribute to the goal. For example, we may not change the location of the robot if having the robot at a certain place is part of the goal.

A ground fluent f **contributes** to the goal if adding or deleting an assignment $f = x$ from a planning state can make the goal true. Formally, f contributes to s^* iff there exists a state s with $s \not\models s^*$ such that $s \cup \{f = x\} \models s^*$ for some value x.

Given an unsolvable planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, an **excuse** is a tuple $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ that implies the solvable **excuse task** $\Pi^\chi = \langle \Delta, \mathcal{C}_\chi, s_\chi, s^* \rangle$ such that $\mathcal{C}_\Pi \subseteq \mathcal{C}_\chi$ and if $(f = x) \in s_0 \triangle s_\chi$ (where $\triangle$ denotes the symmetric set difference) then f must not contribute to s^*.

The changed initial state s_χ is also called **excuse state**.

It should be noted that it is possible that no excuse exists, e.g., if there is no initial state such that a goal state is reachable. More precisely, there is no excuse iff the task is solvable or all changes to the initial state that respect the above mentioned restrictions do not lead to a solvable task.

Acceptable Excuses

If we have two excuses and one changes more initial facts than the other, it would not be an acceptable excuse, e.g., in our example above moving both keys to the room where the robot is would be an excuse. Relocating any one of them to that room would already suffice, though.

So, given two excuses $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ and $\chi' = \langle \mathcal{C}_{\chi'}, s_{\chi'} \rangle$, we say that χ is at least as **acceptable** as χ', written $\chi \preceq \chi'$, iff $\mathcal{C}_\chi \subseteq \mathcal{C}_{\chi'}$ and $s_0 \triangle s_\chi \subseteq s_0 \triangle s_{\chi'}$. A minimal element under the ordering $\preceq$ is called an **acceptable excuse**.

Good Excuses

Given two acceptable excuses, it might nevertheless be the case that one of them subsumes the other if the changes in one excuse can be explained by the other one.

In the example from Fig. 1, one obvious excuse χ would lead to a task in which $door_1$ was unlocked so that the robot could enter $room_1$. This excuse, however, is unsatisfactory since the robot itself could unlock $door_1$ if its key was located in $room_0$ or if $door_2$ was unlocked. So any excuse χ' that contains one of these changes should subsume χ.

We can formalize this subsumption as follows: Let $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ be an acceptable excuse to a planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$ with the plan Ψ solving Π^χ. Another acceptable excuse $\chi' = \langle \mathcal{C}_{\chi'}, s_{\chi'} \rangle$ to Π is called **at least as good as** χ, in symbols $\chi' \sqsubseteq \chi$, if χ' is an acceptable excuse also to $\langle \Delta, \mathcal{C}_\Pi, s_0, nec(s_{\chi'}, \Psi, s^*) \rangle$. We call χ' better than χ, in symbols $\chi' \sqsubset \chi$, iff $\chi' \sqsubseteq \chi$ and $\chi \not\sqsubseteq \chi'$.

In general, good excuses would be expected to consist of changes to so-called *static facts*, facts that cannot be changed by the planner and thus cannot be further regressed from, as captured by the above definition. In our example this could be a new key – with certain properties – and perhaps some additional unlocked doors between rooms.

However, there is also the possibility that there are cyclic dependencies as in the children's song *There's a Hole in the Bucket*. In our example, one excuse would be χ, where the door to $room_2$ is unlocked. In a second one, χ', the robot carries key k_2. Obviously, $\chi \sqsubseteq \chi'$ and $\chi' \sqsubseteq \chi$ hold and thus χ and χ' form a cycle in which all excuses are equally good.

In cases with cyclic dependencies, it is still possible to find even "better" excuses by introducing additional objects, e.g., a new door or a new key in our example. However, cyclic excuses as above, consisting of χ and χ', appear to be at least as intuitive as excuses with additional objects. For these reasons, we define a **good** excuse χ as one such that there either is no better excuse or there exists a different excuse χ' such that $\chi \sqsubseteq \chi'$ and $\chi' \sqsubseteq \chi$.

Perfect Excuses

Of course, there can be many good excuses for a task, and one may want to distinguish between them. A natural way to do so is to introduce a cost function that describes the cost to transform one state into another. Of course, such a cost function is just an estimate because the planner has no way to transform the initial state into the excuse state.

Here, we will use a cost function $c(\cdot)$, which should respect the above mentioned acceptability ordering $\preceq$ as a minimal requirement. So, if $\chi' \preceq \chi$, we require that $c(\chi') \leq c(\chi)$. As a simplifying assumption, we will only consider additive cost functions. So, all ground fluents have predefined costs, and the cost of an excuse is simply the sum over the costs of all facts in the symmetric difference between initial and excuse state. Good excuses with minimal costs are called **perfect excuses**.

Computational Complexity

In the following, we consider ordinary propositional and DATALOG planning, for which the problem of deciding plan existence – the PLANEX problem – is PSPACE- and EXPSPACE-complete, respectively (Erol, Nau, and Subrahmanian 1995). In the context of finding excuses, we will mainly consider the following problem for acceptable, good, and perfect excuses:

- EXCUSE-EXIST: Does there exist any excuse at all?

In its unrestricted form, this problems is undecidable for DATALOG planning.

Theorem 1 *EXCUSE-EXIST is undecidable for* DATALOG *planning.*

Proof Sketch. The main idea is to allow an excuse to introduce an unlimited number of new objects, which are arranged as tape cells of a Turing machine. That these tape cells are empty and have the right structure could be verified by an operator that must be executed in the beginning. After that a Turing machine could be simulated using ideas as in Bylander's proof (Bylander 1994). This implies that the Halting problem on the empty tape can be reduced to EXCUSE-EXIST, which means that the latter is undecidable. ■

However, an excuse with an unlimited number of new objects is, of course, also not very intuitive. For these reasons, we will only consider BOUNDED-EXCUSE-EXIST, where only a polynomial number of new objects is permitted. As it turns out, this version of the problem is not more difficult than planning.

Lemma 2 *There is a polynomial Turing reduction from PLANEX to BOUNDED-EXCUSE-EXIST for acceptable, good, or perfect excuses.*

Proof Sketch. Given a planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$ with planning domain $\Delta = \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}, \mathcal{O} \rangle$, construct two new tasks by extending the set of predicates in $\mathcal{S}$ by a fresh ground atom a leading to $\mathcal{S}'$. In addition, this atom is added to all preconditions in the set of operators resulting in $\mathcal{O}'$. Now we generate:

$$\begin{aligned} \Pi' &= \langle \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}', \mathcal{O}' \rangle, \mathcal{C}_\Pi, s_0, s^* \rangle \\ \Pi'' &= \langle \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}', \mathcal{O}' \rangle, \mathcal{C}_\Pi, s_0 \cup \{a\}, s^* \rangle \end{aligned}$$

Obviously, Π is solvable iff there exists an excuse for Π' and there is no excuse for Π''. ■

It is also possible to reduce the problems the other way around, provided the planning problems are in a deterministic space class.

Lemma 3 *The BOUNDED-EXCUSE-EXIST problem can be Turing reduced to the PLANEX problem – provided PLANEX is complete for a space class that includes PSPACE.*

Proof Sketch. By Savitch's theorem (1980), we know that NSPACE($f(n)$) $\subseteq$ DSPACE($(f(n))^2$), i.e., that all deterministic space classes including PSPACE are equivalent to their non-deterministic counterparts. This is the main reason why finding excuses is not harder than planning.

Let us assume that the plan existence problem for our formalism is XSPACE-complete. Given a planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, the following algorithm will determine whether there is an excuse:

1. If Π is solvable, return "no".
2. Guess a $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ and verify the following:
   a) $\mathcal{C}_\Pi \subseteq \mathcal{C}_\chi$;
   b) $\Pi^\chi = \langle \Delta, \mathcal{C}_\chi, s_\chi, s^* \rangle$ is solvable;

This non-deterministic algorithm obviously needs only XSPACE using an XSPACE-oracle for the PLANEX problem. Since the existence of an excuse implies that there is a perfect excuse (there are only finitely many different possible initial states), the algorithm works for all types of excuses. ■

From the two lemmas, it follows immediately that the EXCUSE-EXIST problem and the PLANEX problem have the same computational complexity.

Theorem 4 *The BOUNDED-EXCUSE-EXIST problem is complete for the same complexity class as the PLANEX problem for all planning formalisms having a PLANEX problem that is complete for a space class containing PSPACE.*

Using similar arguments, it can be shown that we can compute which literals in the initial state can be relevant or are necessary for an excuse. By guessing and verifying using PLANEX-oracles, these problems can be solved and are therefore in the same space class as the PLANEX problems, provided they are complete for a space class including PSPACE.

Candidates for Good Excuses

The range of changes that may occur in acceptable excuses is quite broad: The only excuses forbidden are those that immediately contribute to the goal. We could try to find acceptable excuses and apply goal regression until we find a good excuse, but this would be highly suboptimal, because it might require a lot of goal regression steps. Therefore, we first want to explore if there are any constraints (on fluent or predicate symbols, source or target values) that must be satisfied in any *good* excuse.

In order to analyze the relations between fluent symbols, we apply the notion of *causal graphs* and *domain transition graphs* (Helmert 2006) to the abstract domain description.

The **causal graph** CG_Δ of a planning domain $\Delta = \langle \mathcal{T}, \mathcal{C}_\Delta, \mathcal{S}, \mathcal{O} \rangle$ is a directed graph $(\mathcal{S}, A)$ with an arc $(u, v) \in A$ if there exists an operator $o \in \mathcal{O}$ so that $u \in \mathbf{pre}(o)$ and $v \in \mathbf{eff}(o)$ or both u and v occur in $\mathbf{eff}(o)$. If $u = v$ then (u, v) is in A iff the fluents in the precondition and effect can refer to distinct instances of the fluent.

The causal graph captures the dependencies of fluents on each other; to analyze the ways the values of one fluent can change, we build its domain transition graph. In contrast to the usual definition of domain transition graphs (which is based on grounded planning tasks), the domain of a fluent f can consist of constants as well as free variables. This fact needs to be taken into account when making statements about the domain transition graph (e.g., the reachability of a variable of type t implies the reachability of all variables of subtypes of t). For the sake of clarity, we will largely gloss over this distinction here and treat the graph like its grounded counterpart.

If $dom(f)$ is the domain of a fluent symbol $f \in \mathcal{S}$, its **domain transition graph** $\mathcal{G}_f$ is a labeled directed graph

$(dom(f), E)$ with an arc $(u, v) \in E$ iff there is an operator $o \in \mathcal{O}$ so that $f = u$ is contained in $\mathbf{pre}(o)$ and $f = v \in \mathbf{eff}(o)$. The label consists of the preconditions of o minus the precondition $f = u$. An arc from the unknown symbol, $(\perp, v)$, exists if f does not occur in the precondition. We also call such an arc $(u, v) \in \mathcal{G}_f$ a **transition** of f from u to v and the label of (u, v) its precondition.

For example, the domain transition graph of the robot_pos fluent has one vertex consisting of a variable of type room and one edge $(\texttt{room}, \texttt{room})$ with the label of $\texttt{connected}(room_1, room_2, door) \wedge \texttt{open}(door)$.

In order to constrain the set of possible excuses, we restrict candidates to those fluents and values that are relevant for achieving the goal. The **relevant domain**, $dom_{rel}(f)$, of a fluent f is defined by the following two conditions and can be calculated using a fixpoint iteration: If $f = v$ contributes to the goal, then $v \in dom_{rel}(f)$. Furthermore, for each fluent f' on which f depends, $dom_{rel}(f')$ contains the subset of $dom(f')$ which is (potentially) required to reach any element of $dom_{rel}(f)$.

A **static change** is a change for which there is no path in the domain-transition graph even if all labels are ignored. Obviously all changes to *static variables* are static, but the converse is not always true. For example, in most planning domains, if in a planning task a non-static fluent f is undefined, setting f to some value x would be a static change.

In the following, we show that in some cases it is sufficient to consider static changes as candidates for excuses in order to find all good excuses. To describe these cases, we define two criteria, mutex-freeness and strong connectedness, that must hold for static and non-static fluents, respectively.

We call a fluent f **mutex-free** iff changing the value of an instance of f in order to enable a particular transition of a fluent f' that depends on f does not prevent any other transition. Roughly speaking, excuses involving f' are not good, because they can always be regressed to the dependencies f without breaking anything else. Two special cases of mutex-free fluents are noteworthy, as they occur frequently and can easily be found by analyzing the domain description: If a fluent f is *single-valued*, i.e. there are no two operators which depend on different values for f, it is obviously mutex-free. A less obvious case is free variables. Let o be an operator that changes the fluent $f(p_1, \ldots p_n)$ from p_v to p'_v. A precondition $f'(q_1, \ldots q_n) = q_v$ of o has free variables iff there is at least one variable in $q_1, \ldots, q_n$ that doesn't occur in $\{p_1, \ldots p_n, p_v, p'_v\}$. Here the mutex-freeness is provided because we can freely add new objects to the planning task and thus get new grounded fluents that cannot interfere with any existing fluents.

For example, consider the unlock operator in the KEYS-domain. Its precondition includes $\texttt{key_pos}(key) = \texttt{robot} \wedge \texttt{key_opens}(key) = door$. Here *key* is a free variable, so it is always possible to satisfy this part of the precondition by introducing a new key object and setting its position to the robot and its key_opens property to the door we want to open. As we do not have to modify an existing key, all actions that were previously applicable remain so.

The second criterion is the connectedness of the domain transition graph. We call a fluent f **strongly connected** iff the subgraph of $\mathcal{G}_f$ induced by the relevant domain of f is strongly connected. This means that once f has a value in $dom_{rel}(f)$ any value that may be relevant for achieving the goal can be reached in principle. In practice, most fluents have this property because any operator that changes a fluent from one free variable to another connects all elements of that variable's type.

Theorem 5 *Let Δ be a domain with an acyclic causal graph where all non-static fluents are strongly connected and all static fluents are mutex-free.*

Then any good excuse will only contain static changes.

Proof. First note that a cycle free causal graph implies that there are no co-occurring effects, as those would cause a cycle between their fluents.

If an excuse $\chi = \langle \mathcal{C}_\chi, s_\chi \rangle$ for $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$ is an excuse that contains non-static changes, then we will construct an excuse $\chi' = \langle \mathcal{C}_{\chi'}, s_{\chi'} \rangle \sqsubset \chi$ containing only static facts which can explain χ. As all static fluents are mutex-free, no changes made to the initial state $s_{\chi'}$ to fulfill static preconditions can conflict with changes already made to s_χ, so we can choose the static changes in χ' to be those that make all (relevant) static preconditions true.

Let f, v, v' be a non-static change, i.e., $f = v \in s_0$ and $f = v' \in s_\chi$. This means that there exists a path from v to v' in $\mathcal{G}_f$. If all preconditions along this path are static, we are done as all static preconditions are satisfied in $s_{\chi'}$. If there are non-static preconditions along the path from v to v', we can apply this concept recursively to the fluents of those preconditions. As there are no co-occurring effects and the relevant part of each non-static fluent's domain transition graph is strongly connected, we can achieve all preconditions for each action and restore the original state later. ■

We can easily extend this result to domains with a cyclic causal graph:

Theorem 6 *Let Δ be a domain where all non-static fluents are strongly connected, all static fluents are mutex-free and each cycle in the domain's causal graph contains at least one mutex-free fluent.*

Then any good excuse will only contain static changes or changes that involve a fluent on a cycle.

Proof. We can reduce this case to the non-cyclic case, by removing all effects that change the fluents f fulfilling the mutex-free condition, thus making them static. Let us call this modified domain Δ'.

Let χ be an excuse with non-static changes. Because Δ' contains only a subset of operators of Δ, any non-static change that can further be explained in Δ' can also be explained in Δ. So there exists an $\chi' \sqsubset \chi$, which means that χ cannot be a good excuse unless the changed fluent lies on a cycle so that $\chi \sqsubset \chi'$ may hold, too. ■

While these conditions may not apply to all common planning domains as a whole, they usually apply to a large enough set of the fluents so that limiting the search to static and cyclic excuses speeds up the search for excuses significantly without a big trade-off in optimality.

Finding Excuses Using a Cost-Optimal Planner

We use the results from the previous section to transform the problem of finding excuses into a planning problem by adding operators that change those fluents that are candidates for good excuses. If we make sure that those **change operators** can only occur at the start of a plan, we get an excuse state s_χ by applying them to s_0.

Given an (unsolvable) planning task $\Pi = \langle \Delta, \mathcal{C}_\Pi, s_0, s^* \rangle$, we create a transformed task with action costs $\Pi' = \langle \Delta', \mathcal{C}_{\Pi'}, s_0', s^* \rangle$ as follows.

We recursively generate the relevant domain for each fluent symbol $f \in \mathcal{S}$ by traversing the causal graph, starting with the goal symbols. During this process, we also identify cyclic dependencies. Then we check $\mathcal{G}_f$ for reachability, adding all elements of $dom(f)$ from which $dom_{rel}(f)$ is **not** reachable to $changes(f)$. If f is involved in a cyclic dependency we also add $dom_{rel}(f)$ to $changes(f)$.

To prevent further changes to the planning state after the first execution of a regular action, we add the predicate $started$ to $\mathcal{S}'$ and as a positive literal to the effects of all operators $o \in \mathcal{O}$.

For every fluent f (with arity n) and $v \in changes(f)$ we introduce a new operator set_v^f as follows:

$$\mathbf{pre}(set_v^f) = \neg started \ \wedge f(p_1 \dots p_n) = v \bigwedge_{i=1}^{n} \neg unused(p_i)$$

$$\mathbf{eff}(set_v^f) = \{ f(p_1 \dots p_n) = p_{n+1} \}$$

To add a new object of type t to the initial state, we add a number[2] of **spare objects** $sp_1^t \dots sp_n^t$ to $\mathcal{C}_{\Pi'}$. For each of these objects sp_i^t, the initial state s_0' contains the facts $unused(sp_i^t)$. We then add the operator $add^t(p)$ with $\mathbf{pre}(add^t) = unused(p) \wedge \neg started$ and $\mathbf{eff}(add^t) = \{\neg unused(p)\}$.

To prevent the use of objects that have not been activated yet we add $\neg unused(p_i)$ to each operator $o \in \mathcal{O}$ for each parameter p_i to $pre(o)$ if $pre(o)$ does not contain a fluent or positive literal with p_i as parameter.

Due to the use of the $started$ predicate, any plan Ψ can be partitioned into the actions before $started$ was set (those that change the initial state) and those after. We call the subplans Ψ_{s_0} and Ψ_Π, respectively.

As a final step we need to set the costs of the change actions. In this implementation we assume an additive cost function that assigns non-zero costs to each change and does not distinguish between different instances of a fluent, so $c(f)$ are the costs associated with changing the fluent f and $c(t)$ the costs of adding an object of type t. We set $c(set_v^f) = \alpha c(f)$ and $c(add^t) = \alpha c(t)$ with α being a scaling constant. We need to make sure that the costs of the change actions always dominate the total plan's costs as otherwise worse excuses might be found if they cause Ψ_Π to be shorter. We can achieve this by setting α to an appropriate upper bound of the plan length in the original problem Π.

From the resulting plan Ψ we can easily construct an excuse $\chi = \langle \mathcal{C}_\Psi, s_\Psi \rangle$ with $\mathcal{C}_\Psi = \mathcal{C}_\Pi \cup \{c : add^t(c) \in \Psi\}$ and s_Ψ being the state resulting from the execution of Ψ_{s_0} restricted to the fluents defined in the original Problem Π.

Theorem 7 *Let Π be a planning task, Ψ an optimal solution to the transformed task Π', and $\chi = \langle \mathcal{C}_\Psi, s_\Psi \rangle$ the excuse constructed from Ψ. Then χ is an acceptable excuse to Π.*

Proof. Ψ_Π only contains operators in Δ and constants from $\mathcal{C}_\chi$. Obviously Ψ_Π also reaches the goal from s_Ψ. So Π^χ is solvable and χ thus an excuse. To show that χ is acceptable, we need to show that no excuse with a subset of changes exists. If such an excuse χ' existed it could be reached by applying change operators (as the changes in χ' are a subset of those in χ). Then a plan Ψ' would exist with $c(\Psi'_{s_0}) < c(\Psi_{s_0})$ and, as the cost of Ψ_{s_0} always dominates the cost of Ψ_Π, $c(\Psi') < c(\Psi)$. This contradicts that Ψ is optimal, so χ must be acceptable. ■

Theorem 8 *Let Π be a planning task, Ψ an optimal solution to the transformed task Π', and $\chi = \langle \mathcal{C}_\Psi, s_\Psi \rangle$ the excuse constructed from Ψ. If χ changes only static facts, it is a perfect excuse.*

Proof. As χ contains only static facts, it must be a good excuse. From the definition of the cost function it follows that $c(\chi) = \alpha c(\Psi_{s_0})$, so existence of an excuse χ' with $c(\chi') < c(\chi)$ would imply, as in the previous proof, the existence of a plan Ψ' with $c(\Psi') < c(\Psi)$, contradicting the assumption that Ψ is optimal. ■

Cyclic Excuses

Solving the optimal planning problem will not necessarily give us a good excuse (unless the problem's causal graph is non-cyclic, of course). So if we get an excuse that changes a non-static fact, we perform a *goal regression* as described earlier. We terminate this regression when all new excuses have already been encountered in previous iterations or no excuse can be found anymore. In the former case we select the excuse with the lowest cost from the cycle, in the latter case we need to choose the last found excuse.

Note though, that this procedure will not necessarily find excuses with globally optimal costs: As there is no guarantee that $\chi' \sqsubset \chi$ also implies $c(\chi') \leq c(\chi)$ the goal regression might find excuses that have higher costs than a good excuse that might be found from the initial task Π.

Experiments

To test our implementation's quality we converted selected planning tasks of the IPC domains LOGISTICS (IPC'00), ROVERS and STORAGE (both IPC'06) to use object fluents, so that our algorithm could work directly on each problem's SAS^+ representation. In order to give our program a reason to actually search for excuses, it was necessary to create flaws in each problem's description that made it unsolvable. For each problem file, we modified the initial state by randomly deleting any number of valid fluents and predicates,

[2] As shown in the complexity discussion, the number of new objects might be unreasonably high. In some cases this number can be restricted further but this has been left out for space reasons. In practice we cap the number of spares per type with a small constant.

	sat 0	**opt** 0	**sat** 1	**opt** 1	**sat** 2	**opt** 2	**sat** 3	**opt** 3	**sat** 4	**opt** 4
logistics-04	0.78s	1.43s	0.69s (0.5)	0.94s (0.5)	0.71s (1.5)	1.02s (1.5)	0.53s (1.0)	0.57s (1.0)	0.52s (2.5)	1.29s (2.5)
logistics-06	0.75s	9.81s	0.74s (1.5)	28.12s (1.5)	0.65s (2.5)	101.47s (2.5)	0.65s (3.0)	55.05s (2.5)	0.62s (3.5)	43.57s (3.5)
logistics-08	1.27s	76.80s	1.27s (1.0)	276.99s (1.0)	1.17s (1.0)	46.47s (1.0)	1.08s (5.5)	1176.49s (3.5)	0.96s (5.5)	1759.87s (4.5)
logistics-10	2.62s	—	2.24s (2.0)	—	2.36s (5.5)	—	2.25s (4.0)	—	1.29s (5.5)	—
logistics-12	2.58s	—	2.66s (2.0)	—	2.66s (4.5)	—	2.28s (5.0)	—	1.89s (6.5)	—
logistics-14	4.73s	—	4.78s (2.5)	—	4.24s (6.0)	—	3.70s (7.5)	—	2.71s (6.0)	—
rovers-01	3.04s	3.61s	3.09s (0.5)	5.72s (0.5)	3.17s (1.5)	8.17s (1.5)	2.79s (5.5)	—	2.90s (7.5)	—
rovers-02	3.25s	3.79s	3.24s (0.5)	4.45s (0.5)	3.31s (2.5)	21.48s (2.5)	3.23s (3.0)	62.36s (3.0)	2.87s (6.5)	—
rovers-03	4.15s	5.53s	4.11s (0.5)	7.90s (0.5)	3.55s (2.5)	112.43s (2.5)	4.04s (5.5)	—	3.67s (6.5)	—
rovers-04	5.01s	6.53s	4.94s (1.0)	8.97s (0.5)	68.60s (5.0)	22.01s (2.0)	3.21s (6.0)	—	9.45s (12.0)	—
rovers-05	5.29s	—	6.23s (2.0)	925.61s (2.0)	7.25s (4.0)	—	5.82s (5.0)	790.57s (5.0)	6.32s (8.0)	—
storage-01	1.77s	1.83s	2.01s (0.5)	2.31s (0.5)	1.71s (3.0)	2.11s (2.0)	1.84s (5.0)	24.81s (4.0)	1.82s (4.5)	11.12s (3.5)
storage-05	11.14s	15.66s	10.85s (0.5)	37.09s (0.5)	8.25s (4.0)	53.38s (4.0)	10.25s (6.0)	—	31.70s (6.0)	—
storage-08	30.46s	101.32s	35.59s (1.5)	—	774.17s (5.5)	—	765.32s (7.5)	—	110.31s (8.5)	—
storage-10	88.07s	214.10s	62.93s (1.0)	—	64.56s (2.0)	—	423.71s (3.0)	—	257.10s (4.0)	—
storage-12	131.36s	—	—	—	—	—	—	—	—	—
storage-15	1383.65s	—	—	—	—	—	—	—	—	—

Table 1: Results for finding excuses on some IPC domains. All experiments were conducted on a 2.66 GHz Intel Xeon processor with a 30 minutes timeout and a 2 GB memory limit. We used two setting for the underlying Fast Downward Planner: **sat** is Weighted A* with the enhanced-additive heuristic and a weight of 5, **opt** is A* with the admissible LM Cut Heuristic. For each problem instance there are five versions: the original (solvable) version is referred to as 0 while versions 1 to 4 are generated according to the deletions described in the Experiments section. We used an uniform cost measure with the exception that assigning a value to a previously undefined fluent costs 0.5. Runtime results are in seconds; the excuses costs are shown in parentheses.

or by completely deleting one or more objects necessary to reach the goal in every possible plan (the latter includes the deletion of all fluents and predicates containing the deleted object as a parameter). For instance, in the LOGISTICS domain, we either deleted one or more *city-of* fluents, or all trucks located in the same city, or all airplanes present in the problem.

In order to not only test on problems that vary in the difficulty to find a plan, but also the difficulty to find excuses, we repeated this process four times, each repetition taking the problem gained in the iteration before as the starting point. This lead to four versions of each planning task, each one missing more initial facts compared to the original task than the one before.

Our implementation is based on the Fast Downward planning system (Helmert 2006), using a Weighted A^* search with an extended context-enhanced additive heuristic that takes action costs into account. Depicted are runtimes and the cost of the excuse found. Because this heuristic is not admissible, the results are not guaranteed to be optimal, so we additionally ran tests using A^* and the (admissible) landmark cut heuristics (Helmert and Domshlak 2009).

To judge the quality of the excuses produced we used a uniform cost measure, with one exception: The cost of the assignment of a concrete value to a previously undefined fluent is set to be 0.5. This kind of definition captures our definition of acceptable excuses via the symmetric set difference and also appears to be natural: Assigning a value to an undefined fluent should be of lower cost than changing a value that was given in the original task. Note that switching a fluent's value actually has a cost of 1.0.

As the results in Table 1 show, the time for finding excuses increases significantly in the larger problems. The principal reason for that is that previously static predicates like `connected` in the STORAGE domain have become non-static due to the introduction of change operators. This leads both to a much larger planning state (as they cannot be compiled away anymore), as well as a much larger amount of applicable ground actions. This effect can be seen in the first two columns which show the planning times on the unmodified problems (but with the added change operators).

As expected, optimal search was able to find excuses for fewer problems than satisficing search. Satisficing search came up with excuses for most problems in a few seconds with the exception of the storage domain, due to the many static predicates. The costs of the excuses found were sometimes worse than those found by optimal search, but usually not by a huge amount. If better excuses are desired, additional tests showed that using smaller weights for the Weighted A^* are a reasonable compromise.

It is interesting to note that for the satisficing planner the number of changes needed to get a solvable task has little impact on the planning time. The optimal search, on the other hand, usually takes much longer for the more flawed problems. A possible explanation for this behavior is that the number of possible acceptable excuses grows drastically the more facts we remove from the problem. This makes finding *some* excuse little harder, but greatly increases the difficulty of finding an *optimal* excuse.

While most of the excuses described in this paper can be found in the problems we created this way, it is very unlikely that a problem is contained that is unsolvable due to a cyclic excuse[3]. The aforementioned KEYS-domain on the other hand is predestined to easily create problems that are unsolvable because of some cyclic excuse. So for our second experiment, we designed problems on that domain with an increasing number of rooms n connected so that they form a cycle: for each room k, $k \neq n$, there is a locked door k leading to room $k+1$, and an additional, unlocked one between rooms n and 0 (each connection being valid only in the described direction). For each door k there is a key k which is placed in room k, with the exception of key 0 which is placed in room n and the key to the already unlocked door n which doesn't exist. Obviously a good excuse for every n remains the same: If the robot held the key to door 0 in the

[3]This is only possible if that cyclic excuse was already part of the task, but didn't cause a problem because there existed another, after the deletion nonexistent way to the goal.

rooms	sat	opt	rooms	sat	opt
3	0.91s (1)	0.97s (1)	10	19.20s (2)	368.09s (1)
4	1.2s (1)	1.72s (1)	11	57.39s (2)	849.69s (1)
5	1.75s (1)	4.23s (1)	12	72.65s (2)	1175.23s (1)
6	2.19s (2)	10.69s (1)	13	84.45s (2)	—
7	4.24s (2)	27.01s (1)	14	215.05s (2)	—
8	6.03s (2)	65.15s (1)	15	260.39s (2)	—
9	14.22s (2)	158.28s (1)	16	821.82s (2)	—

Table 2: Results for finding excuses on the KEYS domain. We used the same settings as in the experiments for Table 1, except that the weight in the satisficing run was 1. The rows labeled **rooms** give the number of rooms or the size of the cycle minus 1.

initial state, or if that door was unlocked, the task would easily be solvable. The number of necessary regression steps to find that excuse grows with n, though, which is why KEYS is very well suited to test the performance of the finding cyclic excuses part of our implementation.

As can be seen in the results in Table 2, the planning time both scales well with the size of the cycle and is reasonable for practical purposes.

Related Work

We are not aware of any work in the area of AI planning that addresses the problem of explaining why a goal cannot be reached. However, as mentioned already, there is some overlap with abduction (a term introduced by the philosopher Peirce), counterfactual reasoning (Lewis 1973), belief revision (Gärdenfors 1986), and consistency-based diagnosis (Reiter 1987). All these frameworks deal with identifying a set of propositions or beliefs that either lead to inconsistencies or permit to deduce an observation. There are parallels to our notions of acceptable, good and perfect approaches in these fields (Eiter and Gottlob 1995) – for non-cyclic excuses. The main difference to the logic-based frameworks is that in our case there is no propositional or first-order background theory. Instead, we have a set of operators that allows us to transform states. This difference might be an explanation why cyclic excuses are something that appear to be relevant in our context, but have not been considered as interesting in a purely logic-based context.

Conclusion

In this paper we have investigated situations in which a planner-based agent is incompetent to find a solution for a given planning task. We have defined what an *excuse* in such a situation might look like, and what characteristics such an excuse must fulfill to be accounted as *acceptable*, *good* or even *perfect*. Our main theoretical contribution is a thorough formal analysis of the resulting problem along with the description of a concrete method for finding excuses utilizing existing classical planning systems. On the practical side, we have implemented this method resulting in a system that is capable of finding even complicated excuses in reasonable time which is very helpful both for debugging purposes and in a regular setting like planner-based robot control.

As future work, we intend to extend our implementation to more expressive planning formalisms dealing with time and resources.

Acknowledgements

This research was partially supported by DFG as part of the collaborative research center SFB/TR-8 Spatial Cognition Project R7, the German Federal Ministry of Education and Research (BMBF) under grant no. 01IME01-ALU (DESIRE) and by the EU as part of the Integrated Project CogX (FP7-ICT-2xo15181-CogX).

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Comp. Intell.* 11(4):625–655.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *JAAMAS* 19(3):297–331.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *AIJ* 69(1–2):165–204.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Univ. Freiburg, Institut für Informatik, Freiburg, Germany.

Eiter, T., and Gottlob, G. 1995. The complexity of logic-based abduction. *Jour. ACM* 42(1):3–42.

Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *AIJ* 76(1–2):75–88.

Gärdenfors, P. 1986. Belief revision and the Ramsey test for conditionals. *The Philosophical Review* XCV(1):81–93.

Geffner, H. 2000. Functional STRIPS: a more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Dordrecht, Holland: Kluwer.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS 2009*, 162–169.

Helmert, M. 2006. The fast downward planning system. *JAIR* 26:191–246.

Lewis, D. K. 1973. *Counterfactuals*. Cambridge, MA: Harvard Univ. Press.

Plöger, P.-G.; Pervölz, K.; Mies, C.; Eyerich, P.; Brenner, M.; and Nebel, B. 2008. The DESIRE service robotics initiative. *KI* 4:29–32.

Reiter, R. 1987. A theory of diagnosis from first principles. *AIJ* 32(1):57–95.

Savitch, W. J. 1980. Relations between nondeterministic and deterministic tape complexity. *Journal of Computer and System Sciences* 4:177–192.

Smith, D. E. 2004. Choosing objectives in oversubscription planning. In *ICAPS 2004*, 393–401.

On First-Order Definability and Computability of Progression for Local-Effect Actions and Beyond

Yongmei Liu
Dept. of Computer Science
Sun Yat-sen University
Guangzhou 510275, China
ymliu@mail.sysu.edu.cn

Gerhard Lakemeyer
Dept. of Computer Science
RWTH Aachen
52056 Aachen, Germany
gerhard@cs.rwth-aachen.de

Abstract

In a seminal paper, Lin and Reiter introduced the notion of progression for basic action theories in the situation calculus. Unfortunately, progression is not first-order definable in general. Recently, Vassos, Lakemeyer, and Levesque showed that in case actions have only local effects, progression is first-order representable. However, they could show computability of the first-order representation only for a restricted class. Also, their proofs were quite involved. In this paper, we present a result stronger than theirs that for local-effect actions, progression is always first-order definable and computable. We give a very simple proof for this via the concept of forgetting. We also show first-order definability and computability results for a class of knowledge bases and actions with non-local effects. Moreover, for a certain class of local-effect actions and knowledge bases for representing disjunctive information, we show that progression is not only first-order definable but also efficiently computable.

1 Introduction

A fundamental problem in reasoning about action is *projection*, which is concerned with determining whether or not a formula holds after a number of actions have occurred, given a description of the preconditions and effects of the actions and what the world is like initially. Projection plays an important role in planning and in action languages such as Golog [Levesque *et al.*, 1997] or $\mathcal{A}$ [Gelfond and Lifschitz, 1993].

Two powerful methods to solve the projection problem are *regression* and *progression*. Roughly, regression reduces a query about the future to a query about the initial knowledge base (KB). Progression, on the other hand, changes the initial KB according to the effects of each action and then checks whether the formula holds in the resulting KB. One advantage of progression compared to regression is that after a KB has been progressed, many queries about the resulting state can be processed without any extra overhead. Moreover, when the action sequence becomes very long, as in the case of a robot operating for an extended period of time, regression simply becomes unmanageable. However, projection via progression has three main computational requirements which are not easy to satisfy: the new KB must be efficiently computed, its size should be at most linear in the size of the initial KB (to allow for iterated progression), and it must be possible to answer the query efficiently from the new KB.

As Lin and Reiter [1997] showed in the framework of Reiter's version of the situation calculus [Reiter, 2001], progression is second order in general. And even if it is first-order (FO) definable, the size of the progressed KB may be unmanageable and even infinite. Recently, Vassos and Levesque [2008] also showed that the second-order nature of progression is in general inescapable, as a restriction to FO theories (even infinite ones) is strictly weaker in the sense that inferences about the future may be lost compared to the second-order version.

Nevertheless, for restricted action theories, progression can be FO definable and very effective. The classical example is STRIPS, where the initial KB is a set of literals and progression can be described via the usual *add* and *delete* lists. Since STRIPS is quite limited in expressiveness, it seems worthwhile to investigate more powerful action descriptions which still lend themselves to FO definable progression. Lin and Reiter [1997] already identified two such cases, and recently Vassos *et al.* [2008] were able to show that for so-called local-effect actions, which only change the truth values of fluent atoms with arguments mentioned by the actions, progression is always FO definable. However, they showed the computability of the FO representation only for a special case.

In this paper, we substantially improve and extend the results of Vassos *et al.*. By appealing to the notion of forgetting [Lin and Reiter, 1994], we show that progression for arbitrary local-effect actions is always FO definable and computable. We extend this result to certain actions with non-local effects like the briefcase domain, where moving a briefcase implicitly moves all the objects contained in it. For the special case of so-called proper$^+$ KBs [Lakemeyer and Levesque, 2002] and a restricted class of local-effect actions we show that progression is not only first-order definable but also efficiently computable.

The rest of the paper is organized as follows. In the next section we introduce background material, including the notion of forgetting, Reiter's basic action theories, and progression. In Section 3 we present our result concerning local-effect actions. Section 4 deals with non-local effects and Section 5 with proper$^+$ KBs. Then we conclude.

2 Preliminaries

We start with a first-order language $\mathcal{L}$ with equality. The set of formulas of $\mathcal{L}$ is the least set which contains the atomic formulas, and if ϕ and ψ are in the set and x is a variable, then $\neg\phi$, $(\phi \wedge \psi)$ and $\forall x\phi$ are in the set. The connectives $\vee$, $\supset$, $\equiv$, and $\exists$ are understood as the usual abbreviations. To improve readability, sometimes we put parentheses around quantifiers. We use the "dot" notation to indicate that the quantifier preceding the dot has maximum scope, *e.g.*, $\forall x.P(x) \supset Q(x)$ stands for $\forall x[P(x) \supset Q(x)]$. We often omit leading universal quantifiers in writing sentences. We use $\phi \Leftrightarrow \psi$ to mean that ϕ and ψ are logically equivalent. Let ϕ be a formula, and let μ and μ' be two expressions. We denote by $\phi(\mu/\mu')$ the result of replacing every occurrence of μ in ϕ with μ'.

2.1 Forgetting

Lin and Reiter [1994] defined the concept of forgetting a ground atom or predicate in a theory. Intuitively, the resulting theory should be weaker than the original one, but entail the same set of sentences that are "irrelevant" to the ground atom or predicate.

Definition 2.1 Let μ be either a ground atom $P(\vec{t})$ or a predicate symbol P. Let M_1 and M_2 be two structures. We write $M_1 \sim_\mu M_2$ if M_1 and M_2 agree on everything except possibly on the interpretation of μ.

Definition 2.2 Let T be a theory, and μ a ground atom or predicate. A theory T' is a result of forgetting μ in T, denoted by forget$(T,\mu) \Leftrightarrow T'$, if for any structure M, $M \models T'$ iff there is a model M' of T such that $M \sim_\mu M'$.

Clearly, if both T' and T'' are results of forgetting μ in T, then they are logically equivalent. Similarly, we can define the concept of forgetting a set of atoms or predicates. In this paper, we are only concerned with finite theories. So henceforth we only deal with forgetting for sentences.

Lin and Reiter [1994] showed that for any sentence ϕ and atom p, forgetting p in ϕ is FO definable and can be obtained from ϕ and p by simple syntactic manipulations. Here we reformulate their result in the context of forgetting a finite number of atoms. We first introduce some notation.

Let Γ be a finite set of ground atoms. We call a truth assignment θ of atoms from Γ a Γ-model. Clearly, a Γ-model θ can be represented by a conjunction of literals. We use $\mathcal{M}(\Gamma)$ to denote the set of all Γ-models. Let ϕ be a formula, and θ a Γ-model. We use $\phi[\theta]$ to denote the result of replacing every occurrence of $P(\vec{t})$ in ϕ by the following formula:

$$\bigvee_{j=1}^{m}(\vec{t}=\vec{t}_j \wedge v_j) \vee (\bigwedge_{j=1}^{m}\vec{t} \neq \vec{t}_j) \wedge P(\vec{t}),$$

where for $j = 1, \ldots, m$, the truth value of $P(\vec{t}_j)$ is specified by θ, and v_j is the truth value.

Proposition 2.3 *Let ϕ be a formula, $M \sim_\Gamma M'$, $\theta \in \mathcal{M}(\Gamma)$, and $M \models \theta$. Then for any variable assignment σ, $M, \sigma \models \phi$ iff $M', \sigma \models \phi[\theta]$.*

Theorem 2.4 *forget*$(\phi,\Gamma) \Leftrightarrow \bigvee_{\theta\in\mathcal{M}(\Gamma)} \phi[\theta]$.

Proof: Let M be a structure. We show that $M \models \bigvee_{\theta\in\mathcal{M}(\Gamma)} \phi[\theta]$ iff there is a model M' of ϕ s.t. $M \sim_\Gamma M'$. Suppose the latter. Let $\theta \in \mathcal{M}(\Gamma)$ s.t. $M' \models \theta$. By Proposition 2.3, $M \models \phi[\theta]$. Now suppose $M \models \phi[\theta]$, where $\theta \in \mathcal{M}(\Gamma)$. Let M' be the structure s.t. $M \sim_\Gamma M'$ and $M' \models \theta$. By Proposition 2.3, $M' \models \phi$. ■

Corollary 2.5
forget$(\phi,\Gamma) \Leftrightarrow \bigvee_{\theta\in\mathcal{M}(\Gamma)\text{ and }\phi\wedge\theta\text{ is consistent}} \phi[\theta]$.

Proof: By Proposition 2.3, $\phi \wedge \theta$ entails $\phi[\theta]$.
Thus if $\phi \wedge \theta$ is inconsistent, so is $\phi[\theta]$. ■

Example 2.1 Let $\phi = \forall x.clear(x)$,
and $\Gamma = \{clear(A), clear(B)\}$. Then
$\phi[clear(A) \wedge clear(B)] = \forall x.x = A \wedge true \vee x = B \wedge true \vee x \neq A \wedge x \neq B \wedge clear(x)$, which is equivalent to $\forall x.x = A \vee x = B \vee x \neq A \wedge x \neq B \wedge clear(x)$. Similarly,
$\phi[clear(A) \wedge \neg clear(B)] \Leftrightarrow$
$\forall x.x = A \vee x \neq A \wedge x \neq B \wedge clear(x)$,
$\phi[\neg clear(A) \wedge clear(B)] \Leftrightarrow$
$\forall x.x = B \vee x \neq A \wedge x \neq B \wedge clear(x)$, and
$\phi[\neg clear(A) \wedge \neg clear(B)] \Leftrightarrow$
$\forall x.x \neq A \wedge x \neq B \wedge clear(x)$.
Thus forget$(\phi,\Gamma) \Leftrightarrow$
$\forall x.x = A \vee x = B \vee x \neq A \wedge x \neq B \wedge clear(x)$,
which is equivalent to $\forall x.x \neq A \wedge x \neq B \supset clear(x)$.

We now assume $\mathcal{L}^2$, the second-order extension of $\mathcal{L}$.

Theorem 2.6 *forget*$(\phi, P) \Leftrightarrow \exists R.\phi(P/R)$*, where R is a second-order predicate variable.*

Example 2.2 Let $\phi_1 = \forall x.clear(x) \vee \exists y.on(y,x)$. Then forget$(\phi_1, clear) \Leftrightarrow \exists R\forall x.R(x) \vee \exists y.on(y,x)$, which is equivalent to $true$. Let $\phi_2 = \exists x.clear(x) \wedge \exists y.on(x,y)$. Then forget$(\phi_2, clear) \Leftrightarrow \exists R\exists x.R(x) \wedge \exists y.on(x,y)$, which is equivalent to $\exists x\exists y.on(x,y)$.

In general, forgetting a predicate is not FO definable. Naturally, by Theorem 2.6, second-order quantifier elimination techniques can be used for obtaining FO definability results for forgetting a predicate. In fact, Doherty *et al.* [2001] used such techniques for computing strongest necessary and weakest sufficient conditions of FO formulas, which are concepts closely related to forgetting. As surveyed in [Nonnengart *et al.*, 1999], the following is a classical result on second-order quantifier elimination due to Ackermann (1935). A formula ϕ is positive (resp. negative) wrt a predicate P if $\neg P$ (resp. P) does not occur in the negation normal form of ϕ.

Theorem 2.7 *Let P be a predicate variable, and let ϕ and $\psi(P)$ be FO formulas such that $\psi(P)$ is positive wrt P and ϕ contains no occurrence of P at all. Then*

$$\exists P.\forall\vec{x}(\neg P(\vec{x}) \vee \phi(\vec{x})) \wedge \psi(P)$$

is logically equivalent to $\psi(P(\vec{x}) \leftarrow \phi(\vec{x}))$, denoting the result of replacing each occurrence of $P(\vec{t})$ in ψ with $\phi(\vec{t})$, and similarly if the sign of P is switched and ψ is negative wrt P.

2.2 Basic action theories

The language $\mathcal{L}_{sc}$ of the situation calculus [Reiter, 2001] is a many-sorted first-order language suitable for describing dynamic worlds. There are three disjoint sorts: $action$ for actions, $situation$ for situations, and $object$ for everything else. $\mathcal{L}_{sc}$ has the following components: a constant S_0 denoting the initial situation; a binary function $do(a, s)$ denoting the successor situation to s resulting from performing action a; a binary predicate $Poss(a, s)$ meaning that action a is possible in situation s; action functions, *e.g.*, $move(x, y)$; a finite number of relational fluents, *i.e.*, predicates taking a situation term as their last argument, *e.g.*, $ontable(x, s)$; and a finite number of situation-independent predicates and functions. For simplicity of presentation, we do not consider functional fluents in this paper.

Often, we need to restrict our attention to formulas that refer to a particular situation. For this purpose, we say that a formula ϕ is uniform in a situation term τ, if ϕ does not mention any other situation terms except τ, does not quantify over situation variables, and does not mention $Poss$.

A particular domain of application will be specified by a basic action theory of the following form:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}, \text{ where}$$

1. Σ is the set of the foundational axioms for situations.
2. $\mathcal{D}_{ap}$ is a set of action precondition axioms.
3. $\mathcal{D}_{ss}$ is a set of successor state axioms (SSAs), one for each fluent, of the form

   $$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)),$$

   where $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ are uniform in s.
4. $\mathcal{D}_{una}$ is the set of unique names axioms for actions: $A(\vec{x}) \neq A'(\vec{y})$, and $A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}$, where A and A' are distinct action functions.
5. $\mathcal{D}_{S_0}$, usually called the initial database, is a finite set of sentences uniform in S_0. We call $\mathcal{D}_{S_0}$ the initial KB.

2.3 Progression

Lin and Reiter [1997] formalized the notion of progression. Let $\mathcal{D}$ be a basic action theory, and α a ground action. We denote by S_α the situation term $do(\alpha, S_0)$.

Definition 2.8 Let M and M' be structures with the same domains for sorts *action* and *object*. We write $M \sim_{S_\alpha} M'$ if the following two conditions hold: (1) M and M' interpret all situation-independent predicate and function symbols identically. (2) M and M' agree on all fluents at S_α: For every relational fluent F, and every variable assignment σ, $M, \sigma \models F(\vec{x}, S_\alpha)$ iff $M', \sigma \models F(\vec{x}, S_\alpha)$.

We denote by $\mathcal{L}_{sc}^2$ the second-order extension of $\mathcal{L}_{sc}$. The notion of uniform formulas carries over to $\mathcal{L}_{sc}^2$.

Definition 2.9 Let $\mathcal{D}_{S_\alpha}$ be a set of sentences in $\mathcal{L}_{sc}^2$ uniform in S_α. $\mathcal{D}_{S_\alpha}$ is a progression of the initial KB $\mathcal{D}_{S_0}$ wrt α if for any structure M, M is a model of $\mathcal{D}_{S_\alpha}$ iff there is a model M' of $\mathcal{D}$ such that $M \sim_{S_\alpha} M'$.

Lin and Reiter [1997] proved that progression is always second-order definable. They used an old version of SSAs in their formulation of the result; here we reformulate their result using the current form of SSAs.

We let $\mathcal{D}_{ss}[\alpha, S_0]$ denote the instantiation of $\mathcal{D}_{ss}$ wrt α and S_0, *i.e.*, the set of sentences $F(\vec{x}, do(\alpha, S_0)) \equiv \Phi_F(\vec{x}, \alpha, S_0)$. Let $F_1, \ldots, F_n$ be all the fluents. We introduce n new predicate symbols $P_1, \ldots, P_n$. We use $\phi \uparrow S_0$ to denote the result of replacing every occurrence of $F_i(\vec{t}, S_0)$ in ϕ by $P_i(\vec{t})$. We call P_i the lifting predicate for F_i. When Σ is a finite set of formulas, we denote by $\wedge\Sigma$ the conjunction of its elements.

Theorem 2.10 *The following is a progression of $\mathcal{D}_{S_0}$ wrt α:*

$$\exists \vec{R}.\{\bigwedge(\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\alpha, S_0]) \uparrow S_0\}(\vec{P}/\vec{R}),$$

where $R_1, \ldots, R_n$ are second-order predicate variables.

Therefore, by Theorem 2.6, if ϕ is uniform in S_α and ϕ is a result of forgetting the lifting predicates $\vec{P}$ in $\bigwedge(\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\alpha, S_0]) \uparrow S_0$, then it is a progression of $\mathcal{D}_{S_0}$ wrt α.

3 Progression for local-effect actions

In this section, we show that for local-effect actions, progression is always FO-definable and computable.

We first show an intuitive result concerning forgetting a predicate: if a sentence ϕ entails that the truth values of two predicates P and Q are different at only a finite number of certain instances, then forgetting the predicate Q in ϕ can be obtained from forgetting the Q atoms of these instances in ϕ and then replacing Q by P in the result.

Let $\vec{x}$ be a variable vector, and let $\Delta = \{\vec{t}_1, \ldots, \vec{t}_m\}$ be a set of vectors of ground terms, where all the vectors have the same length. We use $\vec{x} \in \Delta$ to denote the formula $\vec{x} = \vec{t}_1 \vee \ldots \vee \vec{x} = \vec{t}_m$. Let P and Q be two predicates. We let $Q(\Delta)$ denote the set $\{Q(\vec{t}) \mid \vec{t} \in \Delta\}$, and we let $P \approx_\Delta Q$ denote the sentence $\forall \vec{x}. \vec{x} \notin \Delta \supset P(\vec{x}) \equiv Q(\vec{x})$.

Proposition 3.1 *Let ϕ be a formula, $M \models P \approx_\Delta Q$, and $\theta \in \mathcal{M}(Q(\Delta))$. Then for any variable assignment σ, $M, \sigma \models \phi[\theta](Q/P)$ iff $M, \sigma \models \phi[\theta]$.*

Theorem 3.2 *Let P and Q be two predicates, and Δ a finite set of vectors of ground terms. If forget$(\phi, Q(\Delta)) \Leftrightarrow \psi$, then forget$(\phi \wedge (P \approx_\Delta Q), Q) \Leftrightarrow \psi(Q/P)$.*

Proof: Let $\Gamma = Q(\Delta)$. By Theorem 2.4, forget$(\phi, \Gamma) \Leftrightarrow \bigvee_{\theta \in \mathcal{M}(\Gamma)} \phi[\theta]$. Let M be a structure. We show that $M \models \bigvee_{\theta \in \mathcal{M}(\Gamma)} \phi[\theta](Q/P)$ iff there is a model M' of $\phi \wedge (P \approx_\Delta Q)$ s.t. $M \sim_Q M'$. Suppose the latter. Let $\theta \in \mathcal{M}(\Gamma)$ s.t. $M' \models \theta$. Since $M' \models \theta$ and $M' \models \phi$, by Proposition 2.3, $M' \models \phi[\theta]$. Since $M' \models P \approx_\Delta Q$, by Proposition 3.1, $M' \models \phi[\theta](Q/P)$. Since $M \sim_Q M'$, $M \models \phi[\theta](Q/P)$.

Now suppose $M \models \phi[\theta](Q/P)$, where $\theta \in \mathcal{M}(\Gamma)$. Let M' be the structure s.t. $M \sim_Q M'$, $M' \models P \approx_\Delta Q$, and $M' \models \theta$. Since $M \sim_Q M'$ and $M \models \phi[\theta](Q/P)$, $M' \models \phi[\theta](Q/P)$. Since $M' \models P \approx_\Delta Q$, by Proposition 3.1, $M' \models \phi[\theta]$. Since $M' \models \theta$, by Proposition 2.3, $M' \models \phi$. ■

Actions in many dynamic domains have only local effects in the sense that if an action $A(\vec{c})$ changes the truth value

of an atom $F(\vec{d}, s)$, then $\vec{d}$ is contained in $\vec{c}$. This contrasts with actions having non-local effects such as moving a briefcase, which will also move all the objects inside the briefcase without having mentioned them.

Definition 3.3 An SSA is local-effect if both $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ are disjunctions of formulas of the form $\exists \vec{z}[a = A(\vec{u}) \wedge \phi(\vec{u}, s)]$, where A is an action function, $\vec{u}$ contains $\vec{x}$, $\vec{z}$ is the remaining variables of $\vec{u}$, and ϕ is called a context formula. An action theory is local-effect if each SSA is local-effect.

Example 3.1 Consider a simple blocks world. We use a single action, $move(x, y, z)$, moving a block x from block y to block z. We use two fluents: $clear(x, s)$, block x has no blocks on top of it; $on(x, y, s)$, block x is on block y; $eh(x, s)$, the height of block x is even. Clearly, the following SSAs are local-effect:

$$
\begin{aligned}
&clear(x, do(a, s)) \equiv (\exists y, z)a = move(y, x, z) \vee \\
&\quad clear(x, s) \wedge \neg(\exists y, z)a = move(y, z, x), \\
&on(x, y, do(a, s)) \equiv (\exists z)a = move(x, z, y) \vee \\
&\quad on(x, y, s) \wedge \neg(\exists z)a = move(x, y, z), \\
&eh(x, do(a, s)) \equiv (\exists y, z)[a = move(x, y, z) \wedge \neg eh(z, s)] \vee \\
&\quad eh(x, s) \wedge \neg(\exists y, z)[a = move(x, y, z) \wedge eh(z, s)].
\end{aligned}
$$

By using the unique names axioms, the instantiation of a local-effect SSA on a ground action can be simplified. Suppose the SSA for F is local-effect. Let $\alpha = A(\vec{t})$ be a ground action. Then each of $\gamma_F^+(\vec{x}, \alpha, s)$ and $\gamma_F^-(\vec{x}, \alpha, s)$ is equivalent to a formula of the following form:

$$\vec{x} = \vec{t}_1 \wedge \psi_1(s) \vee \ldots \vee \vec{x} = \vec{t}_n \wedge \psi_n(s),$$

where $\vec{t}_i$ is a vector of ground terms contained in $\vec{t}$, and $\psi_i(s)$ is a formula whose only free variable is s. Without loss of generality, we assume that for a local-effect SSA, $\gamma_F^+(\vec{x}, \alpha, s)$ and $\gamma_F^-(\vec{x}, \alpha, s)$ have the above simplified form. In the case of our blocks world, we have:

$$
\begin{aligned}
&clear(x, do(move(c_1, c_2, c_3), s)) \equiv x = c_2 \vee \\
&\quad clear(x, s) \wedge \neg(x = c_3), \\
&on(x, y, do(move(c_1, c_2, c_3), s)) \equiv x = c_1 \wedge y = c_3 \vee \\
&\quad on(x, y, s) \wedge \neg(x = c_1 \wedge y = c_2), \\
&eh(x, do(move(c_1, c_2, c_3), s)) \equiv x = c_1 \wedge \neg eh(c_3, s) \vee \\
&\quad eh(x, s) \wedge \neg(x = c_1 \wedge eh(c_3, s)).
\end{aligned}
$$

Following Vassos *et al.* [2008], we define the concepts of argument set and characteristic set:

Definition 3.4 Let $\mathcal{D}$ be local-effect, and α a ground action. The argument set of fluent F wrt α is the following set:

$$\Delta_F = \{\vec{t} \mid \vec{x} = \vec{t} \text{ appears in } \gamma_F^+(\vec{x}, \alpha, s) \text{ or } \gamma_F^-(\vec{x}, \alpha, s)\}.$$

The characteristic set of α is the following set of atoms:

$$\Omega(s) = \{F(\vec{t}, s) \mid F \text{ is a fluent and } \vec{t} \in \Delta_F\}.$$

We let $\mathcal{D}_{ss}[\Omega]$ denote the instantiation of $\mathcal{D}_{ss}$ wrt Ω, *i.e.*, the set of sentences $F(\vec{t}, S_\alpha) \equiv \Phi_F(\vec{t}, \alpha, S_0)$, where $F(\vec{t}, s) \in \Omega$. We let $\mathcal{D}_{ss}[\overline{\Omega}]$ denote the set of sentences $\vec{x} \notin \Delta_F \supset F(\vec{x}, S_\alpha) \equiv F(\vec{x}, S_0)$. Then we have

Proposition 3.5 *Let $\mathcal{D}$ be local-effect, and α a ground action. Then $\mathcal{D}_{una} \models \mathcal{D}_{ss}[\alpha, S_0] \equiv \mathcal{D}_{ss}[\Omega] \cup \mathcal{D}_{ss}[\overline{\Omega}]$.*

Theorem 3.6 *Let $\mathcal{D}$ be local-effect, and α a ground action. Let $\Omega(s)$ be the characteristic set of α. Then the following is a progression of $\mathcal{D}_{S_0}$ wrt α:*

$$\bigwedge \mathcal{D}_{una} \wedge \bigvee_{\theta \in \mathcal{M}(\Omega(S_0))} (\mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\Omega])[\theta](S_0/S_\alpha).$$

Proof: By Proposition 3.5, Theorems 2.4, 2.10 and 3.2. ■

The size of the progression is $O(2^m n)$, where m is the size of the characteristic set, and n is the size of the action theory. When we do iterative progression wrt a sequence δ of actions, the size of the resulting KB is $O(2^{lm} n)$, where l is the length of δ, and m is the maximum size of the characteristic sets.

Corollary 3.7 *Progression for local-effect actions is always FO definable and computable.*

We remark that this result is strictly more general than the one obtained by Vassos *et al.* [2008], who only showed that progression for local-effect actions is FO definable in a non-constructive way, *i.e.*, they left open the question whether the FO representation is computable or even finite. Moreover, while their proof is quite involved, having to appeal to Compactness of FO logic, ours is actually fairly simple.

Example 3.1 continued. Let $\alpha = move(A, B, C)$. Then $\Omega = \{clear(B, s), clear(C, s), on(A, B, s), on(A, C, s), eh(A, s)\}$. $\mathcal{D}_{ss}[\Omega]$ is simplified to $\{clear(B, S_\alpha), \neg clear(C, S_\alpha), \neg on(A,B,S_\alpha), on(A,C,S_\alpha), eh(A,S_\alpha) \equiv \neg eh(C, S_0)\}$.

Let $\mathcal{D}_{S_0}$ be the following set of sentences:
$A \neq B, A \neq C, B \neq C, clear(A, S_0)$,
$on(A, B, S_0), clear(C, S_0), clear(x, S_0) \supset eh(x, S_0)$,
$on(x, y, S_0) \supset \neg clear(y, S_0)$.
Then $\mathcal{D}_{S_0}$ entails ϑ, denoting $\neg clear(B, S_0) \wedge clear(C, S_0) \wedge on(A, B, S_0) \wedge \neg on(A, C, S_0)$. Thus there are only two $\theta \in \mathcal{M}(\Omega(S_0))$ which are consistent with $\mathcal{D}_{S_0}$:
$\theta_1 = \vartheta \wedge eh(A, S_0)$ and $\theta_2 = \vartheta \wedge \neg eh(A, S_0)$.

For example, let $\phi = clear(x, S_0) \supset eh(x, S_0)$.
Then $\phi[\theta_1] \Leftrightarrow clear(x, S_0)[\vartheta] \supset x = A \vee x \neq A \wedge eh(x, S_0)$,
and $\phi[\theta_2] \Leftrightarrow clear(x, S_0)[\vartheta] \supset x \neq A \wedge eh(x, S_0)$.
Thus $\phi[\theta_1] \vee \phi[\theta_2]$ is equivalent to

$$
\begin{aligned}
&x = C \vee x \neq B \wedge x \neq C \wedge clear(x, S_0) \\
&\qquad \supset x = A \vee x \neq A \wedge eh(x, S_0).
\end{aligned}
$$

By Theorem 3.6 and Corollary 2.5, the following is a progression of $\mathcal{D}_{S_0}$ wrt α:

$$
\begin{aligned}
&move(x_1, y_1, z_1) = move(x_2, y_2, z_2) \\
&\qquad \supset x_1 = x_2 \wedge y_1 = y_2 \wedge z_1 = z_2, \\
&A \neq B, A \neq C, B \neq C, clear(A, S_\alpha), \\
&x = C \vee x \neq B \wedge x \neq C \wedge clear(x, S_\alpha) \supset \\
&\qquad x = A \vee x \neq A \wedge eh(x, S_\alpha), \\
&x = A \wedge y = B \vee \\
&\quad (x \neq A \vee y \neq B) \wedge (x \neq A \vee y \neq C) \wedge on(x, y, S_\alpha) \supset \\
&\qquad \neg(y = C \vee y \neq B \wedge y \neq C \wedge clear(y, S_\alpha)), \\
&clear(B, S_\alpha), \neg clear(C, S_\alpha), \neg on(A, B, S_\alpha), \\
&on(A, C, S_\alpha), eh(A, S_\alpha) \equiv \neg eh(C, S_\alpha).
\end{aligned}
$$

4 Progression for normal actions

In the last section, we showed that for local-effect actions, progression is FO definable and computable. An interesting observation about non-local-effect actions is that their effects often do not depend on the fluents on which they have non-local effects, that is, they normally have local effects on the

fluents that appear in every γ_F^+ and γ_F^-. For example, moving a briefcase will move all the objects in it as well without affecting the fluent *in*. We will call such an action a normal action. In this section, we show that for a normal action α, if the initial KB has the property that for each fluent F on which α has non-local effects, the only appearance of F is in the form of $\phi(\vec{x}) \supset F(\vec{x}, S_0)$ or $\phi(\vec{x}) \supset \neg F(\vec{x}, S_0)$, then progression is FO definable and computable.

Our result is inspired by a result by Lin and Reiter [1997] that for context-free action theories, that is, action theories where every predicate appearing in every γ_F^+ and γ_F^- is situation-independent, if the initial KB has the property that for each fluent F, the only appearance of F is in the form of $\phi(\vec{x}) \supset F(\vec{x}, S_0)$ or $\phi(\vec{x}) \supset \neg F(\vec{x}, S_0)$, then progression is FO definable and computable. Incidentally, their result can be considered as an application of a simple case of the classical result by Ackermann (see Theorem 2.7). To prove our result, we combine the application of this simple case and the proof idea behind our result for local-effect actions.

We first present this simple case of Ackermann's result:

Definition 4.1 We say that a finite theory T is semi-definitional wrt a predicate P if the only appearance of P in T is in the form of $P(\vec{x}) \supset \phi(\vec{x})$, where we call $\phi(\vec{x})$ a necessary condition of P, or $\phi(\vec{x}) \supset P(\vec{x})$, where we call $\phi(\vec{x})$ a sufficient condition of P. We use WSC_P (meaning weakest sufficient condition) to denote the disjunction of $\phi(\vec{x})$ such that $\phi(\vec{x}) \supset P(\vec{x})$ is in T, and we use SNC_P (meaning strongest necessary condition) to denote the conjunction of $\phi(\vec{x})$ such that $P(\vec{x}) \supset \phi(\vec{x})$ is in T.

Theorem 4.2 *Let T be finite and semi-definitional wrt P. Let T' be the set of sentences in T that contains no occurrence of P. Then forget$(T, P) \Leftrightarrow T' \wedge \forall\vec{x}.WSC_P(\vec{x}) \supset SNC_P(\vec{x})$.*

Proof: Clearly, $\exists R.T(P/R) \models T' \wedge \forall\vec{x}.\mathrm{WSC}_P(\vec{x}) \supset \mathrm{SNC}_P(\vec{x})$. To prove the opposite entailment, simply use the definition $\forall\vec{x}.P(\vec{x}) \equiv \mathrm{WSC}_P(\vec{x})$. ■

The following proposition shows that the SSA for a fluent F is semi-definitional wrt the predicate $F(\vec{x}, S_0)$ provided that F does not appear in γ_F^+ or γ_F^-.

Proposition 4.3 *The sentence $F(\vec{x}, S_\alpha) \equiv \gamma_F^+(\vec{x}, \alpha, S_0) \vee F(\vec{x}, S_0) \wedge \neg\gamma_F^-(\vec{x}, \alpha, S_0)$ is equivalent to the following sentences: $\neg\gamma_F^+ \wedge F(\vec{x}, S_\alpha) \supset F(\vec{x}, S_0)$, $F(\vec{x}, S_0) \supset \gamma_F^- \vee F(\vec{x}, S_\alpha)$, $\gamma_F^+ \supset F(\vec{x}, S_\alpha)$, and $\neg\gamma_F^+ \wedge \gamma_F^- \supset \neg F(\vec{x}, S_\alpha)$.*

We now formalize our constraints on the actions and the initial KBs.

Definition 4.4 We say that a ground action α has local effects on a fluent F, if by using $\mathcal{D}_{una}$, each of $\gamma_F^+(\vec{x}, \alpha, s)$ and $\gamma_F^-(\vec{x}, \alpha, s)$ can be simplified to a disjunction of formulas of the form $\vec{x} = \vec{t} \wedge \psi(s)$, where $\vec{t}$ is a vector of ground terms, and $\psi(s)$ is a formula whose only free variable is s. We denote by $\mathrm{LE}(\alpha)$ the set of all fluents on which α has local effects.

Definition 4.5 We say that α is normal if for each fluent F, all the fluents that appear in γ_F^+ and γ_F^- are in $\mathrm{LE}(\alpha)$.

Clearly, both context-free and local-effect actions are normal actions.

Definition 4.6 We say that $\mathcal{D}_{S_0}$ is normal wrt α if for each fluent $F \notin \mathrm{LE}(\alpha)$, $\mathcal{D}_{S_0}$ is semi-definitional wrt F.

Thus any fluent $F \in \mathrm{LE}(\alpha)$ can appear in $\mathcal{D}_{S_0}$ in an arbitrary way. We now have the main result of this section:

Theorem 4.7 *Let $\mathcal{D}_{S_0}$ be normal wrt a normal action α. Then progression of $\mathcal{D}_{S_0}$ wrt α is FO definable and computable.*

Proof: By Theorem 2.10, we need to forget the lifting predicates in $\bigwedge(\mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\alpha, S_0]) \uparrow S_0$. Since α is normal, for each fluent F, all the fluents that appear in γ_F^+ and γ_F^- are in $\mathrm{LE}(\alpha)$. By Proposition 4.3, for each $F \notin \mathrm{LE}(\alpha)$, $\mathcal{D}_{ss}[\alpha, S_0]$ is semi-definitional wrt $F(\vec{x}, S_0)$. Since $\mathcal{D}_{S_0}$ is normal wrt α, for each fluent $F \notin \mathrm{LE}(\alpha)$, $\mathcal{D}_{S_0}$ is semi-definitional wrt F. By applying Theorem 4.2, we forget the lifting predicate for $F \notin \mathrm{LE}(\alpha)$. Now by applying Theorem 3.6, we forget the lifting predicate for $F \in \mathrm{LE}(\alpha)$. ■

Example 4.1 The following is $\mathcal{D}_{ss}$ for the briefcase domain:

$at(x, l, do(a, s)) \equiv (\exists b)[a = move(b, l) \wedge (x = b \vee in(x, b, s))] \vee$
$\quad at(x, l, s) \wedge \neg(\exists b, m)[a = move(b, m) \wedge (x = b \vee in(x, b, s))]$,
$in(x, b, do(a, s)) \equiv a = putin(x, b) \vee$
$\quad in(x, b, s) \wedge \neg a = getout(x, b)$.

For a ground action $\alpha = move(c_1, c_2)$, by using $\mathcal{D}_{una}$, $\mathcal{D}_{ss}[\alpha, S_0]$ can be simplified as follows:

$at(x, l, do(\alpha, S_0)) \equiv l = c_2 \wedge (x = c_1 \vee in(x, c_1, S_0)) \vee$
$\quad at(x, l, S_0) \wedge \neg(x = c_1 \vee in(x, c_1, S_0))$,
$in(x, b, do(\alpha, S_0)) \equiv in(x, b, S_0)$.

Clearly, α has local effects on in, and it is a normal action.

Now let $\mathcal{D}_{S_0}$ be as follows:

$\exists x \forall y \neg in(x, y, S_0)$, $\neg in(A_1, B_1, S_0)$,
$in(A_2, B_2, S_0) \vee in(A_2, B_3, S_0)$,
$at(b, l, S_0) \wedge in(x, b, S_0) \supset at(x, l, S_0)$,
$at(x, l', S_0) \wedge l \neq l' \supset \neg at(x, l, S_0)$,
$b = B_1 \wedge l = L_1 \vee b = B_2 \wedge l = L_2 \supset at(x, l, S_0)$,
$b = B_3 \wedge (l = L_1 \vee l = L_2) \supset \neg at(x, l, S_0)$.

Then $\mathcal{D}_{S_0}$ is normal wrt $\alpha = move(B_1, L_2)$. To progress it wrt α, we first apply Theorem 4.2 to forget the lifting predicate for $at(x, l, s)$, and obtain a set Σ of sentences as follows:

1. If $\phi \in \mathcal{D}_{S_0}$ does not mention fluent at, then $\phi \in \Sigma$.
2. Add to Σ the following sentences:
   $l = L_2 \wedge (x = B_1 \vee in(x, B_1, S_0)) \supset at(x, l, S_\alpha)$,
   $l \neq L_2 \wedge (x = B_1 \vee in(x, B_1, S_0)) \supset \neg at(x, l, S_\alpha)$.
3. If $\phi \supset at(x, l, S_0)$ is in $\mathcal{D}_{S_0}$, then add to Σ the sentence
   $\phi \wedge \neg(x = B_1 \vee in(x, B_1, S_0)) \supset at(x, l, S_\alpha)$.
4. If $\phi \supset \neg at(x, l, S_0)$ is in $\mathcal{D}_{S_0}$, add to Σ the sentence
   $\phi \wedge (l \neq L_2 \vee x \neq B_1 \wedge \neg in(x, B_1, S_0)) \supset \neg at(x, l, S_\alpha)$.

Now since we have $in(x, b, S_\alpha) \equiv in(x, b, S_0)$, we simply replace each occurrence of S_0 in Σ with S_α; the result together with $\mathcal{D}_{una}$ is a progression of $\mathcal{D}_{S_0}$ wrt α.

5 Progression of proper$^+$ KBs

In Sections 3 and 4, we showed that for local-effect and normal actions, progression is FO definable and computable. However, the progression may not be efficiently computable.

In this section, we show that for local-effect and normal actions, progression is not only FO definable but also efficiently computable under the two constraints that the initial KB is in the form of the so-called proper$^+$ KBs, which represent first-order disjunctive information, and the successor state axioms are essentially quantifier-free.

Proper$^+$ KBs were proposed by Lakemeyer and Levesque [2002] as a generalization of proper KBs, which were proposed by Levesque [1998] as an extension of databases. Intuitively, a proper$^+$ KB is equivalent to a (possibly infinite) set of ground clauses. A tractable limited reasoning service has been developed for proper$^+$ KBs [Liu *et al.*, 2004; Liu and Levesque, 2005]. What is particularly interesting about our results here is that progression of proper$^+$ KBs is definable as proper$^+$ KBs, so that we can make use of the available tractable reasoning service.

To formally define proper$^+$ KBs, we use a FO language $\mathcal{L}_c$ with equality, a countably infinite set of constants, which are intended to be unique names, and no other function symbols. We let e range over ewffs, *i.e.*, quantifier-free formulas whose only predicate is equality. We denote by $\mathcal{E}$ the axioms of equality and the set of formulas $\{(c \neq c') \mid c$ and c' are distinct constants$\}$. We let $\forall\phi$ denote the universal closure of ϕ.

Definition 5.1 Let e be an ewff and d a clause. Then a formula of the form $\forall(e \supset d)$ is called a $\forall$-clause. A KB is called *proper*$^+$ if it is a finite non-empty set of $\forall$-clauses.

Example 5.1 Consider our blocks world. The following is a initial KB $\mathcal{D}_{S_0}$ which is proper$^+$:
$on(x, y, S_0) \supset \neg clear(y, S_0)$,
$on(x, y, S_0) \wedge eh(y, S_0) \supset \neg eh(x, S_0)$,
$x = A \vee x = C \supset clear(x, S_0)$,
$x = D \vee x = E \vee x = F \supset \neg eh(x, S_0)$,
$x = A \wedge y = B \vee x = B \wedge y = F \supset on(x, y, S_0)$,
$on(C, D, S_0) \vee on(C, E, S_0)$.

We begin with forgetting in proper$^+$ KBs. We first introduce some definitions and propositions.

Definition 5.2 Let ϕ be a sentence, and p a ground atom. We say that p is irrelevant to ϕ if forget$(\phi, p) \Leftrightarrow \phi$.

Proposition 5.3 *Let p be a ground atom. Let ϕ_1, ϕ_2, ϕ_3 be sentences such that p is irrelevant to them. Then forget$((\phi_1 \supset p) \wedge (p \supset \phi_2) \wedge \phi_3, p) \Leftrightarrow (\phi_1 \supset \phi_2) \wedge \phi_3$.*

Proposition 5.4 *Let $\phi = \forall(e \supset d)$ be a $\forall$-clause, and $P(\vec{c})$ a ground atom. Suppose that for any $P(\vec{t})$ appearing in d, $e \wedge \vec{t} = \vec{c}$ is unsatisfiable. Then $P(\vec{c})$ is irrelevant to ϕ.*

Definition 5.5 Let Σ be a proper$^+$ KB, and $P(\vec{c})$ a ground atom. We say that Σ is in normal form wrt $P(\vec{c})$, if for any $\forall(e \supset d) \in \Sigma$, and for any $P(\vec{t})$ appearing in d, either $\vec{t}$ is $\vec{c}$ or $e \wedge \vec{t} = \vec{c}$ is unsatisfiable.

Proposition 5.6 *Let $P(\vec{c})$ be a ground atom. Then every proper$^+$ KB can be converted into an equivalent one which is in normal form wrt $P(\vec{c})$. This can be done in $O(n + 2^w m)$ time, where n is the size of Σ, m is the size of sentences in Σ where P appears, and w is the maximum number of appearances of P in a sentence of Σ.*

Proof: Let $\phi = \forall(e \supset d)$ be a $\forall$-clause. Let $P(\vec{t_1}), \ldots, P(\vec{t_k})$ be all the appearances of P in ϕ, and let $\Theta = \{\bigwedge_{i=1}^{k} \vec{t_i} \circ_i \vec{c} \mid \circ_i \in \{=, \neq\}\}$. Let $\theta \in \Theta$. We let $d[\theta]$ denote d with each $P(\vec{t_i})$, $1 \leq i \leq k$, replaced by $P(\vec{c})$ if θ contains $\vec{t_i} = \vec{c}$. We use $\phi[\theta]$ to denote $\forall(e \wedge \theta \supset d[\theta])$. Obviously, ϕ is equivalent to the theory $\{\phi[\theta] \mid \theta \in \Theta\}$, which we denote by NF$(\phi, P(\vec{c}))$. For a proper$^+$ KB Σ, we convert it into the union of NF$(\phi, P(\vec{c}))$ where $\phi \in \Sigma$. ■

In the above proof, we can remove those generated $\forall$-clauses $\forall(e \supset d)$ where d contains complemental literals or e is unsatisfiable wrt $\mathcal{E}$. For example, the ewff $x = y \wedge x = A \wedge y = B$ is unsatisfiable wrt $\mathcal{E}$. Also, a $\forall$-clause $\forall(e \wedge t = c \supset d)$ can be simplified to $\forall(e \supset d)(t/c)$. Finally, an ewff can be simplified by use of $\mathcal{E}$.

Definition 5.7 Let $\phi_1 = \forall(e_1 \supset d_1 \vee P(\vec{t}))$ and $\phi_2 = \forall(e_2 \supset d_2 \vee \neg P(\vec{t}))$ be two $\forall$-clauses, where $\vec{t}$ is a vector of constants or a vector of distinct variables. Without loss of generality, we assume that ϕ_1 and ϕ_2 do not share variables other than those contained in $\vec{t}$. We call the $\forall$-clause $\forall(e_1 \wedge e_2 \supset d_1 \vee d_2)$ the $\forall$-resolvent of the two input clauses wrt $P(\vec{t})$.

Theorem 5.8 *Let Σ be a proper$^+$ KB, and $P(\vec{c})$ a ground atom. Then the result of forgetting $P(\vec{c})$ in Σ is definable as a proper$^+$ KB and can be computed in $O(n + 4^w m^2)$ time, where n, w, and m are as above.*

Proof: We first convert Σ into normal form wrt $P(\vec{c})$. Then we compute all $\forall$-resolvents wrt $P(\vec{c})$ and remove all clauses with $P(\vec{c})$. This results in a proper$^+$ KB, which, by Propositions 5.3 and 5.4, is a result of forgetting $P(\vec{c})$ in Σ. ■

Theorem 5.9 *Let Σ be a proper$^+$ KB which is semi-definitional wrt predicate P. Then the result of forgetting P in Σ is definable as a proper$^+$ KB and can be computed in $O(n + m^2)$ time, where n is the size of Σ, and m is the size of sentences in Σ where P appears.*

Proof: We compute all $\forall$-resolvents wrt $P(\vec{x})$ and remove all clauses containing $P(\vec{x})$. This results in a proper$^+$ KB, which, by Theorem 4.2, is a result of forgetting P in Σ. ■

In the above theorems (Theorems 5.8 and 5.9), it is reasonable to assume that $w = O(1)$ and $m^2 = O(n)$. Under this assumption, both forgetting can be computed in $O(n)$ time.

Based on the above theorems, we have the following results concerning progression of proper$^+$ KBs. We first introduce a constraint on successor state axioms.

Definition 5.10 An SSA is essentially quantifier-free if for each ground action α, by using $\mathcal{D}_{una}$, each of $\gamma_F^+(\vec{x}, \alpha, s)$ and $\gamma_F^-(\vec{x}, \alpha, s)$ can be simplified to a quantifier-free formula.

For example, the SSAs for our blocks world and briefcase examples are essentially quantifier-free. For a local-effect SSA, if each context-formula is quantifier-free, then it is essentially quantifier-free. In general, if both $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ are disjunctions of formulas of the form $\exists\vec{z}[a = A(\vec{u}) \wedge \phi(\vec{x}, \vec{z}, s)]$, where $\vec{u}$ contains $\vec{z}$, and ϕ is quantifier-free, then the SSA is essentially quantifier-free.

Proposition 5.11 *Suppose $\mathcal{D}_{ss}$ is essentially quantifier-free. Then $\mathcal{D}_{ss}[\alpha, S_0]$ is definable as a proper$^+$ KB.*

Theorem 5.12 *Suppose that $\mathcal{D}$ is local-effect, $\mathcal{D}_{ss}$ is essentially quantifier-free, and $\mathcal{D}_{S_0}$ is proper$^+$. Then progression of $\mathcal{D}_{S_0}$ wrt any ground action α is definable as a proper$^+$ KB and can be efficiently computed.*

Theorem 5.13 *Suppose that $\mathcal{D}_{ss}$ is essentially quantifier-free, α is a normal action, and $\mathcal{D}_{S_0}$ is a proper$^+$ KB which is normal wrt α. Then progression of $\mathcal{D}_{S_0}$ wrt α is definable as a proper$^+$ KB and can be efficiently computed.*

Example 5.1 continued. We now progress $\mathcal{D}_{S_0}$ wrt $\alpha = move(A, B, C)$. For simplicity, we remove the second sentence from $\mathcal{D}_{S_0}$. The characteristic set of α is $\Omega = \{clear(B, s), clear(C, s), on(A, B, s), on(A, C, s), eh(A, s)\}$. $\mathcal{D}_{ss}[\Omega]$ is simplified to the following proper$^+$ KB:
$\{clear(B, S_\alpha), \neg clear(C, S_\alpha), \neg on(A, B, S_\alpha), on(A, C, S_\alpha), eh(A, S_\alpha) \vee eh(C, S_0), \neg eh(A, S_\alpha) \vee \neg eh(C, S_0)\}$.

We convert $\mathcal{D}_{S_0}$ into normal form wrt $\Omega(S_0)$, and obtain after simplification:

$$y \neq B \wedge y \neq C \wedge (x \neq A \vee y \neq B \wedge y \neq C) \supset (on(x, y, S_0) \supset \neg clear(y, S_0)),$$
$$x \neq A \supset (on(x, B, S_0) \supset \neg clear(B, S_0)),$$
$$x \neq A \supset (on(x, C, S_0) \supset \neg clear(C, S_0)),$$
$$on(A, B, S_0) \supset \neg clear(B, S_0),$$
$$on(A, C, S_0) \supset \neg clear(C, S_0),$$
$$x = D \vee x = E \vee x = F \supset \neg eh(x, S_0),$$
$$clear(A, S_0), clear(C, S_0), on(A, B, S_0), on(B, F, S_0),$$
$$on(C, D, S_0) \vee on(C, E, S_0).$$

We now do resolution on $\mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\Omega]$ wrt atoms in $\Omega(S_0)$, delete all clauses with some atom from $\Omega(S_0)$, and obtain the following set Σ:

$$clear(B, S_\alpha), \neg clear(C, S_\alpha), \neg on(A, B, S_\alpha), on(A, C, S_\alpha),$$
$$eh(A, S_\alpha) \vee eh(C, S_0), \neg eh(A, S_\alpha) \vee \neg eh(C, S_0),$$
$$y \neq B \wedge y \neq C \wedge (x \neq A \vee y \neq B \wedge y \neq C) \supset (on(x, y, S_0) \supset \neg clear(y, S_0)),$$
$$x \neq A \supset \neg on(x, C, S_0),$$
$$x = D \vee x = E \vee x = F \supset \neg eh(x, S_0),$$
$$clear(A, S_0), on(B, F, S_0), on(C, D, S_0) \vee on(C, E, S_0).$$

We replace every occurrence of S_0 in Σ with S_α; the result together with $\mathcal{D}_{una}$ is a progression of $\mathcal{D}_{S_0}$ wrt α.

6 Conclusions

In this paper, we have presented the following results. First, we showed that for local-effect actions, progression is FO definable and computable. This result is stronger than the one obtained by Vassos *et al.* [2008], and our proof is a very simple one via the concept of forgetting. Next, we went beyond local-effect actions, and showed that for normal actions, *i.e.*, actions whose effects do not depend on the fluents on which the actions have non-local effects, if the initial KB is semi-definitional wrt these fluents, progression is FO definable and computable. Third, we showed that for local-effect actions whose successor state axioms are essentially quantifier-free, progression of proper$^+$ KBs is definable as proper$^+$ KBs and can be efficiently computed. Thus we can utilize the available tractable limited reasoning service for proper$^+$ KBs. As an extension of our first result, we have shown that for finite-effect actions, which change the truth values of fluents at only a finite number of instances, progression is FO definable. We have also shown that in the presence of functional fluents, our first and second results still hold. For lack of space, these results will be presented in a longer version of the paper. For the future, we would like to implement a Golog interpreter based on progression of proper$^+$ KBs, which we expect will lead to a more efficient version of Golog compared to the current implementation based on regression.

Acknowledgments

We thank the anonymous reviewers for very helpful comments. This work was supported in part by the European Union Erasmus Mundus programme.

References

[Doherty *et al.*, 2001] P. Doherty, W. Lukaszewicz, and A. Szalas. Computing strongest necessary and weakest sufficient conditions of first-order formulas. In *Proc. IJCAI-01*, 2001.

[Gelfond and Lifschitz, 1993] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2–4):301–323, 1993.

[Lakemeyer and Levesque, 2002] G. Lakemeyer and H. J. Levesque. Evaluation-based reasoning with disjunctive information in first-order knowledge bases. In *Proc. KR*, 2002.

[Levesque *et al.*, 1997] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming*, 31:59–84, 1997.

[Levesque, 1998] H. J. Levesque. A completeness result for reasoning with incomplete first-order knowledge bases. In *Proc. KR-98*, 1998.

[Lin and Reiter, 1994] F. Lin and R. Reiter. Forget it! In *Working Notes of AAAI Fall Symposium on Relevance*, 1994.

[Lin and Reiter, 1997] F. Lin and R. Reiter. How to progress a database. *Artificial Intelligence*, 92(1–2):131–167, 1997.

[Liu and Levesque, 2005] Y. Liu and H. J. Levesque. Tractable reasoning in first-order knowledge bases with disjunctive information. In *Proc. AAAI-05*, 2005.

[Liu *et al.*, 2004] Y. Liu, G. Lakemeyer, and H. J. Levesque. A logic of limited belief for reasoning with disjunctive information. In *Proc. KR-04*, 2004.

[Nonnengart *et al.*, 1999] A. Nonnengart, H. J. Ohlbach, and A. Szalas. Elimination of predicate quantifiers. In *Logic, Language and Reasoning, Part I*, pages 159–181. 1999.

[Reiter, 2001] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. 2001.

[Vassos and Levesque, 2008] S. Vassos and H. J. Levesque. On the progression of situation calculus basic action theories: Resolving a 10-year-old conjecture. In *Proc. AAAI-08*, 2008.

[Vassos *et al.*, 2008] S. Vassos, G. Lakemeyer, and H. J. Levesque. First-order strong progression for local-effect basic action theories. In *Proc. KR-08*, 2008.

golog.lua: Towards a Non-Prolog Implementation of Golog for Embedded Systems

Alexander Ferrein

Robotics and Agents Research Lab
University of Cape Town
Rondebosch 7005
South Africa
alexander.ferrein@uct.ac.za

Abstract. Among many approaches to address the high-level decision making problem for autonomous robots and agents, the robot programming and plan language Golog follows a logic-based deliberative approach, and its successors were successfully deployed in a number of robotics applications over the past ten years. Usually, Golog interpreter are implemented in Prolog, which is not available for our target platform, the bi-ped robot platform Nao. In this paper we sketch our first approach towards a prototype implementation of a Golog interpreter in the scripting language Lua. With the example of the elevator domain we discuss how the basic action theory is specified and how we implemented fluent regression in Lua. One possible advantage of the availability of a Non-Prolog implementation of Golog could be that Golog becomes available on a larger number of platforms, and also becomes more attractive for roboticists outside the Cognitive Robotics community.

1 Introduction

To address the problem of high-level decision making for autonomous robots or agents, a number of different robot programming languages have been developed. Each of these follow a particular paradigm or technique how the problem of decision making could be solved. Among them are for instance the Procedural Reasoning System (PRS) [1], the Saphira architecture [2], Reactive Action Packages (RAP) [3], or the Reactive Plan Language (RPL) [4] and Structured Reactive Controller (SRC) [5]. These approaches mostly follow a reactive paradigm or deploy hierarchical task networks.

The robot programming and plan language Golog, on the other hand, follows a logic-based deliberative approach [6], and its successors were successfully deployed in a number robotics applications over the past ten years. The applications range from service robotics to even robotic soccer applications. Golog was used as the high-level decision-making component on a number of different robot platforms, ranging from the RWI B14/B21 over the Sony Aibo to Lego Mindstorms, and many more tailor-made platforms. During the course of the

last decade it was extended with useful features like integrating sensing and exogenous actions [7], continuous change [8], or decision-theoretic planning [9] to name just a few. It emerged into an expressive robot programming and plan language and is used in the Cognitive Robotics community.

Golog interpreter are in general implemented in Prolog. This is straightforward as the semantic of the language constructs is described in the situation calculus [10], a first order action logic which allows for reasoning about actions and change. The implementation, or better the specification of Golog in Prolog is just a page long, and it was shown that the implementation is correct, having a proper Prolog interpreter [11].

Until now, we did not face any problems to run Golog on our robots [12], though one has to note that it always took some extra computational resources to do the action selection with Golog. However, for our current robot project, it seems that no Prolog interpreter is available. We want to use Golog on the bi-ped robot Nao from French Aldebaran, which is running Open Embedded Linux, for which, to the best of our knowledge, no Prolog system is currently available.

This motivated to start with a re-implementation of Golog in a language different from Prolog. We came up with using the the scripting language Lua [13], which we also used for the Behaviour Engine that we are running on the robot [14]. In this paper, we present our first approach towards a prototype implementation of a Vanilla Golog interpreter in Lua. We first briefly introduce the hardware platform and the software system which we are running on the Nao, as this motivates our decisions to try a re-implementation of Golog, and to use Lua for this purpose. Then, we introduce Golog and Lua, and show in some detail the Prolog implementation of Golog, before we give details of our Lua re-implementation. In particular, we show how the basic action theory is specified or how regression is implemented in our interpreter. As a proof-of-concept we implemented the elevator domain [6, 11]. We conclude with discussing the preliminary state of this work and give an outlook to some future work.

2 Our Embedded System: The Nao Robot

2.1 Hardware Platform

In the past, we used and extended Golog for several robotics applications ranging from service robotics to robotic soccer applications [12]. We learnt to value the flexibility in modelling the application domain and expressing control knowledge in an elegant way. Our Golog implementation was always based on Prolog, and we could run a Prolog engine on our robots so far. It is worth noting that running Golog on a mobile robot platform requires some extra computational resources. For our latest robotics project, however, no Prolog system, to the best of our knowledge, seems to be available. Moreover are the computational resources of our new mobile robot platform quite restricted.

Currently, we are developing robotic soccer applications for the bi-ped humanoid robot Nao built by French Aldebaran [15]. The platform is the successor

of the Sony Aibo in Robocup's Standard Platform League. It is a 21 degree-of-freedom humanoid robot about 58 cm tall and is equipped with two VGA resolution cameras, ultrasonic sensors as well as infrared sensors, an inertial measurement unit, tactile sensors and force resistance sensors in the feet. The robot has microphones, loudspeakers, and it has a number of LEDs with which it can display status information. It is powered by an AMD Geode 500 MHz CPU and equipped with 256 MB of memory. Furthermore, it has 1 GB flash memory for hard disk space.

2.2 Software Framework

The programming framework we are using on the Nao is the Fawkes framework. The Fawkes robot software framework [16] provides the infrastructure to run a number of plug-ins which fulfil specific tasks. Each plug-in consists of one or more threads. The application runs a main loop which is sub-divided into certain stages. Threads can be executed either concurrently or synchronised with a central main loop to operate in one of the stages. All threads registered for the same stage are woken up and run concurrently. The software architecture of Fawkes follows a component-based approach. A component is defined as a binary unit of deployment that implements one or more well-defined interfaces to provide access to an inter-related set of functionality configurable without access to the source code. Components are implemented in Fawkes as a plug-in. For communication between the components we use a blackboard infrastructure which serves well-defined communication interfaces.

Another building block of Fawkes is the use of a Lua-based Behaviour Engine [14]. The idea of this behaviour engine is to provide a behaviour middle-ware between the low-level robot system and high-level decision-making modules. The behaviour engine deploys extended hybrid state machines for monitoring the execution of action patterns or primitive actions. We decided to deploy Lua for the Behaviour Engine, as this scripting language is lightweight with a small memory footprint, though expressive enough for the task. Moreover, Lua showed its potential in a number of successful AI applications so far.[1] The behaviour middle-ware was designed with a high-level decision-making module such as a Golog-based deliberative component in mind. The good experiences with Lua influenced our decision for developing a Lua-based Golog interpreter.

3 Situation Calculus and Golog

3.1 Situation Calculus

The situation calculus is a first order language with equality which allows for reasoning about actions and their effects. The world evolves from an initial situation due to primitive actions. Possible world histories are represented by sequences of actions. The situation calculus distinguishes three sorts: *actions*,

[1] See http://lua.org for a list of applications.

situations, and domain dependent *objects*. A special binary function symbol $do : action \times situation \rightarrow situation$ exists, with $do(a,s)$ denoting the situation which arises after performing action a in the situation s. The constant S_0 denotes the initial situation, i.e. the situation where no actions have yet occurred. The state the world is in is characterised by functions and relations with a situation as their last argument. They are called *functional* and *relational fluents*, respectively.

For each action one has to specify a *precondition axiom* stating under which conditions it is possible to perform the respective action and an *effect axiom* formulating how the action changes the world in terms of the specified fluents. An action precondition axiom has the form $Poss(a(\boldsymbol{x}), s) \equiv \Phi(\boldsymbol{x}, s)$ where the binary predicate $Poss \subseteq action \times situation$ specifies when an action can be executed, and $\boldsymbol{x}$ stands for the arguments of action a. In the situation calculus the effects of actions are formalised by so-called successor state axioms of the form $F(\boldsymbol{x}, do(a,s)) \equiv \varphi_F^+(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\varphi_F^-(\boldsymbol{x}, a, s)$, where F denotes a fluent, φ_F^+ and φ_F^- are formulae describing under which conditions F is true, or false resp. This axiom simply states that F is true after performing action a if φ_F^+ holds, or the fluent keeps its former value if it was not made false. Successor state axioms describe Reiter's solution to the frame problem [11], the problem that all the non-effects of an action have to be formalised as well. Note that free variables in the occurring formulae are meant to be implicitly universally quantified. The background theory (also called basic action theory, or BAT for short) is a set of sentences $\mathcal{D}$ consisting of $\mathcal{D} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$, where $\mathcal{D}_{ssa}$ contains sentences about the successor state axioms, $\mathcal{D}_{ap}$ contains the action precondition axioms, $\mathcal{D}_{una}$ states sentences about unique names for actions, and $\mathcal{D}_{S_0}$ consists of axioms stating what holds in the initial situation. Additionally, Σ contains a number of foundational axioms defining situations. For details we refer to [17, 11].

3.2 Golog

The high-level programming language Golog [6] is based on the situation calculus. As planning is known to be computationally very demanding in general, which makes it impractical for deriving complex behaviours with hundreds of actions, Golog finds a compromise between planning and programming. The robot or agent is equipped with a situation calculus background theory. The programmer can specify the behaviour just like in ordinary imperative programming languages but also has the possibility to project actions into the future. The amount of planning (projection) used is in the hand of the programmer. With this, one has a powerful language for specifying the behaviours of a cognitive robot or agent. While the original Golog is well-suited to reason about actions and their effects, it has the drawback that a program has to be evaluated up to the end before the first action can be performed. It might be that the world changed between plan generation and plan execution so that the plan is not appropriate or is invalid. The original Golog was extended over recent years and has become an expressive robot programming language. Dialects of Golog

feature online execution, sensing facilities [7], continuous change [8], or decision-theoretic planning [9], to name just a few. Golog and it's derivatives were used in a number of successful cognitive robotics applications such as [18–20, 12, 21].

Golog interpreter are usually based on Prolog, as it is straight-forward to implement the logical situation calculus specification of the language in a logic programming framework. In the following, we present the implementation of Vanilla Golog.[2]

1. *Sequence*: each sequence of actions or program statements are evaluated from left to right;
```
do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).
```
2. *Test action*: a test actions evaluates the truth value of a logical formula;
```
do(?(P),S,S) :- holds(P,S).
```
3. *Pick*: a variable `V` is non-deterministically chosen and each occurrence of `V` is substituted in program `E` resulting in `E1`;
```
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
```
4. *Star*: implements the non-deterministic repetition of a program;
```
do(star(E),S,S1) :- S1 = S ; do(E : star(E),S,S1).
```
5. *Conditional*: if the test on the condition holds, the then-branch is evaluated, otherwise, the else-branch is taken into account;
```
do(if(P,E1,E2),S,S1) :- do((?(P) : E1) # (?(-P) : E2),S,S1).
```
6. *Loop*: the star operator is conditioned on `P`;
```
do(while(P,E),S,S1):- do(star(?(P) : E) : ?(-P),S,S1).
```
7. *Non-deterministic choice of actions*: either program `E1` or `E2` is evaluated;
```
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
```
8. *Procedure*: for a procedure, it is simply checked if a declaration of a procedure in Prolog's database with the same name exists, if so, the body of the procedure is further evaluated;
```
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
```
9. *Primitive Action*: similar to procedures, it is checked whether the action is declared. Furthermore, it is checked if the precondition axiom `poss` in the current situation holds.
```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).
```

[2] It is based on a version, which was adopted by S. Sardiña to run under SWI-Prolog and is available at http://www.cs.toronto.edu/cogrobo/main/systems/index.html.

The Elevator Example In the following we restate the elevator example from [6]. First, we need to define the actions in our basic action theory:

```
primitive_action(turnoff(N)).
primitive_action(up(N)).
poss(up(N),S) :- currentFloor(M,S), M < N.
```

Other primitive actions which are required for the elevator application are actions for going one storey down, opening, and closing the elevator doors, and can also be found in [11]. As an example for a control procedure we give the $goFloor(n)$ procedure and the main control procedure *control*.

```
proc(goFloor(N), ?(currentFloor(N)) # up(N) # down(N)).
proc(control, while(some(n, on(n)), serveAfloor) : park).
```

goFloor tests the actual floor, and either chooses the *up* or *down* action. Note that Golog here depends on Prolog's backtracking mechanism to choose either to go up or down. The *control* procedure simply calls the procedure *serveAFloor* (which we omit here) until there is no more clause instance of the fluent $on(n)$ in the clauses database. The fluent $on(n)$ becomes true, if an elevator call button on storey n was pressed. Initially, the call buttons on storey 3 and 5 are pressed, meaning that the facts $on(3, S_0)$ and $on(5, S_0)$ are added to the database as being valid in S_0.

As a final example we want to show the successor state axiom for the fluent *currentFloor*:

```
currentFloor(M,do(A,S)) :- A = up(M) ; A = down(M) ;
    not A = up(N), not A = down(N), currentFloor(M,S).
```

$currentFloor(m, do(a, s))$ is true if either the action performed in situation s was $up(m)$ or $down(m)$, otherwise the fluent value of *currentFloor* remains unchanged. A successful execution of this program leads to the situation, where all buttons are turned off and the elevator is in its parking position, i.e.

$$s^* = ([down(3), turnoff(3), open, close, \\ up(5), turnoff(5), open, close, down(0), open], S_0).$$

4 Lua

Lua [13] is a scripting language designed to be fast, lightweight, and embeddable into other applications. These features make it particularly interesting for the Nao platform. The whole binary package takes less then 200 KB of storage. When loaded, it takes only a very small amount of RAM. This is particularly important on the constrained Nao platform and the reason Lua was chosen for our Behaviour Engine over other scripting languages that are usually more than an order of magnitude larger [22]. In an independent comparison Lua has turned out to be one of the fastest interpreted programming languages [22, 23]. Besides that Lua is an elegant, easy-to-learn language [24] that should allow newcomers to start developing behaviours quickly. Another advantage of Lua is that it can

interact easily with C/C++. As most robot software is written in C/C++, there exists an easy way to make Lua available for a particular control software.

Lua is a dynamically typed language, attaching types to variable values. Eight different types are distinguished: *nil*, *boolean*, *number*, *string*, *table*, *function*, *userdata*, and *thread*. For each variable value, its type can be queried.

The central data structure in Lua are tables. Table entries can be addressed by either indices, thus implementing ordinary arrays, or by string names, implementing associative arrays. Table entries can refer to other tables allowing for implementing recursive data types. For example `t["name"] = value1` stores the key-value pair (name, value1) in table `t`, while `t[9] = value2` stores the value2 at position 9 in array `t`. Special iterators allow access to associative tables and arrays. Note that both index methods can be used for the same table.

Function are first-class types in Lua and can be created at run-time, assigned to a variable, or passed as an argument, or be destroyed. Lua provides proper tail calls and closures to decrease the needed stack size for function calls. Furthermore, Lua offers a special method to modify code at run-time. With the `loadstring()` statement chunks of code (one or more instruction of Lua code is called chunk) can be executed at run-time. This comes in handy to modify code while you are running it.

Lua deploys a register-based virtual machine to run its code. Although it is an interpreted language, a program will be pre-compiled. For code chunks that are created at run-time, the above mentioned `loadstring` function pre-compiles the chunk at run-time. As the virtual machine is register-based the code size is decreased. Furthermore, Lua uses an efficient mark-and-sweep garbage collection which, for example, frees unused values in associative arrays efficiently. Finally, we want to mention the explicit support for threads and co-routines in the Lua specification, which can be particularly useful for robotics applications.

5 Implementing `golog.lua`: A First Approach

In the following we show some details of our prototypical implementation of Golog in Lua. One of the very pleasant features of Prolog is that it is very easy to work with terms and formulae. Creating instances of terms or atoms even at run-time of a program to modify the code is very helpful for dealing with dynamic domains. For example, the initial value of the fluent *on* in our elevator example above was kept by adding the instances $on(3, S_0)$ and $on(5, S_0)$ to the internal clauses data base as atomic formulae. Unification is built in, substituting variable values comes for free and using list structures is comfortable.

In Lua, these concepts are not directly available. As opposed to terms and lists, Lua has its associative table structures and is good in dealing with string values. In our first implementation of Golog in Lua, we mainly use tables and strings to implement a function `Do` which interprets Golog programs. On a technical side, note that our implementation resembles more the transition semantics as proposed in ConGolog [25], as each interpreted statement is consumed from the input program, leaving the rest program to be interpreted. This is however

not a problem as it has been shown that the transition semantics is equivalent to the evaluation semantics that is used by Vanilla Golog, but it requires some special treatment when features such as backtracking are needed. Also, the way we encode action effects is slightly different. We address these topics in the next section.

5.1 Programs and Situation Terms as Nested Tables

In `golog.lua`, a program is a table which is defined in a Lua environment, and the program is run by calling a function `Do(p,s)`

```
prog = {{a_1, {}}, {a_2, {x_1, x_2}},
  {if, {fluent}, {a_3, {}}, {a_4, {}}}}
local s_2, failure = Do(prog, {})
```

The program above consists of an 0-ary action a_1 in sequence with $a_2(x_1, x_2)$ and a conditional which, depending on the truth value of *fluent*, chooses a_3 or a_4, resp. The program is executed with calling the interpreter function `Do` which takes a program and a situation term, and returns the resulting situation after executing the program, or, if the program trace lead to a failure, i.e. the failure variable is true, s_2 contains the last possible action. Assuming that *fluent* holds, the resulting execution trace of the `prog` will be

```
s_2 =  {{"a_1",{}}, {"a_2", {"x_1", "x_2"},{"a_3,{}}}}
```
[3]

We use the empty table or *nil* to represent S_0. Therefore, the above situation term has to be interpreted as $do(a_3, do(a_2(x_1, x_2), do(a_1, S_0)))$. Similarly, we represent logical formulae as tables, with the connectives in prefix notation, i.e. $\{\texttt{and}, \phi, \{\texttt{or}, \psi, \theta\}\}$ represents the formula $\phi \wedge (\psi \vee \theta)$.

5.2 Axioms as Tables and Functions

The domain specification and the basic action theory are defined using special associative arrays. Each fluent name in the domain description has to be inserted into the special table `D_fluents`, which, for the elevator example, means:

```
D_fluents=Set{on, currentFloor}
```

`Set` is one of our auxiliary functions to store the values `on` and `currentFloor` in the associative array `D_fluents`. Similarly, we need to keep track of our primitive actions and procedures:

```
D_act = Set{turnoff, open, close, up, down}
D_proc = Set{proc_goFloor, proc_serve,
  proc_park,proc_control}
```

[3] Note that all program statements, actions, and fluent names must be given as strings. For reasons of readability, we omit the quotation marks throughout this paper. Note also that Lua supports to return multiple value, the situation term and the failure condition in this case.

We need these sets to be able to distinguish user-defined actions, procedures, and fluents from Golog keywords when interpreting a program. Next, we show the definition of fluent *on*.

```
on = {["name"]=on, ["arity"] = 1 }

function on.initially(N)
   return {{"3"}, {"5"}}
end
```

For the fluent *on* from our elevator domain, we define a table called `on`. To refer to it in the Golog program, the field `["name"]` needs to be filled, as well as the arity of the fluent. Next, we specify the initial value, i.e. the value in S_0. We here use Lua's facility to define unnamed tables. The function returns an associative array with the fields `table["3"]=true` and `table["5"]=true`. The intended meaning is that in the initial situation $on(3, S_0) \equiv on(5, S_0) \equiv \top$. For 0-ary fluents, we would simply return the value `true`.

For defining the effects of an action, the user of the Prolog implementation of Vanilla Golog needs to specify successor state axioms. In our Lua implementation, we use effect axioms similar to the way they were implemented in Indigolog [7]:

```
function on.turnoff(N, prev_val)
   local list = Retract(tostring(N),
     prev_val[1])
   return prev_val
end
```

Note that `Retract(value, array)` is one of our helper functions that deletes `value` from `array`. This means, to evaluate the value of fluent f in a particular situation s we apply the effect axioms of those actions that are mentioned in the situation term and that change f's value. For example, consider the elevator domain with $s' = ([down(3), turnoff(3), open, close, up(5), turnoff(5), open, close], S_0)$. To evaluate the value of fluent *on* we have to apply the following effect axioms:

```
on.turnoff(5,on.turnoff(3,on.initially(n)))
```

as the actions *up*, *down*, *open*, *close* do not change the value of the fluent *on*. The above string is generated at run-time by the interpreter and Lua's facility to apply code at run-time using the `loadstring()` command and is executed to evaluate the effects of an action. Similarly, we use `loadstring` to check whether precondition axioms or effects axioms are defined. For example, the code fragment

```
local action = "turnoff"
if loadstring("return type(" ..
  action .. ".Poss)")() == "function" then
  ...
else error("Precondition axiom for action
    \%s undefined\\n", action) end
```

checks at run-time whether the precondition axiom for action `turnoff` is defined.[4] In the above example our evaluation routine for checking the effects of action `turnoff` returns the value *nil*, meaning that no instances of *on* are currently valid. To be able to evaluate fluent values this way, we follow the convention that the last argument of an effect axiom always takes the value from a successor situation. Another requirement is that all action effects changing a fluent value are defined per action and that the closed world assumption holds. We therefore require that effect axioms are defined as part of the fluent definition. (`on.turnoff` means that the function *turnoff* is defined in the namespace of `on` and can only be used in this namespace.) An example for a precondition axiom is:

```
function turnoff.Poss(N, s)
  return has_fval({on, {N}}, s)
end
```

The action `turnoff(n)` is only possible, if the call button on the respective storey is pressed, i.e iff `(on(n), s) = true`. To access the fluent value the user can apply the function `holds` or `has_fval`, which we address below.

5.3 Holds, Pick, and Some

To evaluate logical formulae, we provide a function `holds(f,s)`, which evaluates if `f` holds in situation `s`. As stated above, we use an prefix notation for logical connectives. We evaluate sub-formulae recursively, just as Golog's Prolog implementation does.

```
function holds(f, s)
  if type(f) == "table" then
    if Member(f[1], binop)  --binary op
    then return holds_binop(f, s)
5   ...
end

function holds_binop(f, s)
  local op = table.remove(f, 1)
10  local result
  -- traverse formula and evaluate each element
  if op == "and" then result = true
    while f[1] do
     local eval=holds(table.remove(f,1),s)
15   result = result and eval end
   ...
  return result
end
```

[4] In our current naive and not optimised proof-of-concept implementation, we check for the axiom each time the action is called. A more clever way would be to check these things once before executing the program. Note that ".." is the string concatenation operator in Lua.

We call the respective evaluation function depending on the operator type. The example above shows the evaluation for the operator `and`. More complicated is the implementation of quantifiers. The current reasoning engine in our prototype implementation is somewhat restricted. Existential quantifiers are only allowed in fluent formulae. To this end, we introduce a function `has_fval`, to query fluent formulae.

```
has_fval({"on", {"3"}}, {}) → true
has_fval({"on", {nil}}, {}) → {{"3", "5"}}
```

With the help of `has_fval` it is straight-forward to the operator $\mathit{some}(n, \mathit{fluent}(n))$, which refers to the logical formula $\exists x.\mathit{fluent}(x)$. In the Prolog implementation, the evaluation via the predicate *holds* is successful, if $\mathit{fluent}(n, s)$ follows from subsequently applying the fluent's successor state axiom on s given its initial value. Similarly, we check with `has_fval(f,s)` if there is an instantiation of `f` in `s` by subsequently applying the effect axioms on `f`. Vanilla Golog also offers the *pi* operator, which binds the variable in the formula $\exists x.\mathit{fluent}(x)$: $\mathit{pi}(n, ?(\mathit{fluent}(n)))$. We omitted the *pi* operator in our current prototype implementation. We achieve the variable binding with applying *some* to a fluent, whose arguments are void, i.e. the arguments of the fluent contain nil values (second case of `has_fval(f,s)` above). A more general reasoning engine is subject to future work. One possibility might be to use the constraint system CLIPS [26], for which also a Lua interface is available.

Finally we need to address argument substitution to implement call-by-value functionality. This is needed for procedure arguments, but also for the aforementioned case of substitutions for quantifiers. The substitution algorithm for procedures is quite simple, for each argument value, we get the variable name from the procedure prototype, and substitute each occurrence of the variable reference in the procedure body with the value as given in the procedure call. To allow nested procedure calls, we hold the substitutions on each call level on a stack. Similarly, we substitute each occurrence of a variable in a *some* statement in the subsequent program.

5.4 Executing Actions

As primitive actions need to be declared first, it is easy to distinguish them from other constructs. As we have mentioned above follows our implementation the transition semantics idea of ConGolog. The current program statement is consumed while being interpreted. Hence, it is particularly easy to execute actions immediately by using the `loadstring()` functionality.[5]

One complication of Lua comes with how Lua handles variables. Lua supports in general call-by-reference, meaning that you alter the original data object given as an argument, and not a local copy of it. If you need a local copy of a Golog subprogram such as a procedure, you have to iterate through the table representing

[5] With adding sensing and exogenous actions to the Lua specifications of the interpreter together with guarded action theories [7], it should be quite straight-forward to extend our current implementation to an on-line interpreter.

the sub-program and copy each sub-table to a new table. Although, the programs are not large and Lua is fast with accessing tables, it seems to be overhead. Here, we might need to find a different way to deal with this. This means also that for backtracking as needed for Golog's "#" operator, we need to copy not only the different branches of the non-deterministic choice, but also the situation terms, so that we can determine the correct situation term for the successful branch.

Finally, we sketch the implementation of our function `Do`:

```
function Do(program, s1)
  local s2, failure, instr
  repeat
    instr = table.remove(program, 1)
    -- process next instruction
    if type(instr) == "table" then
      -- pop first statement from program
      local statement=table.remove(instr, 1)
      -- process the first instruction
      if statement == nil then return
         s1, true
       -- non-det. choice
      elseif statement == "#" then
        local ndet_1=table.remove(instr, 1)
        local ndet_2=table.remove(instr, 1)
        s2,failure=Do_ndet(ndet_1,ndet_2,s1)
      ...  -- other statements ...
      else       -- unknown action
        error("Unknown statemet\n"))
        failure = true
      end
    else
      error("Program invalid\n")
      s2 = {}; failure = true
    end
    s1 = s2
    if failure then break end
  until not program[1]
  return s2, failure
end
```

The main loop iterates over the program, popping the first statement from the program and processing it. If there are no more statements in the input program, and no failure occurred, the execution of the program was successful, if at any point during the interpretation of the program a failure occurs, the further execution is immediately terminated.

5.5 The Elevator in Lua

The main control loop for the elevator in Lua looks like:

```
proc_control={["name"]=control,{},
```

```
  {{while,{some,{n},{on,{n}}},
    {{proc_serve,{n}}}},{proc_park}}}

5 proc_serve ={["name"]=proc_serve,{N},
  {{proc_goFloor,{N}},{turnoff,{N}},
    {open,{}}, {close,{}}}}

proc_goFloor={["name"]=proc_goFloor,{N},
10  {{#,{#,{?,{{currentFloor,{N}}}},
        {up,{N}}},{down,{N}}}}}
```

As long as there are still instances of the fluent on, the procedure `proc_serve` is executed. As we discussed above note that we get a value for the argument n of $some(n)$. The argument for the procedure $proc_serve(n)$ is also substituted by this value as the procedure is the body of the *while* instruction. The execution trace of the elevator program in Lua is

```
*** SUCCESS! No (more) solution (lol):
s2={{down, {3}}, {turnoff, {3}}, {open, {}}, {close,{}},{up,{5}},
  {turnoff, {5}},  {open,{}},{close,{}},{down,{0}},{open,{}}}
```

leading to the same solution as the Prolog implementation of Vanilla Golog.

6 Discussion

Why do we believe this work is useful? Our motivation to begin with this work was the unavailability of a Prolog system on our target platform. The Open Embedded Linux system, to the best of our knowledge, does not offer a Prolog system so far. As we still want to make use of Golog for the high-level decision making of the robot, we need to provide an interpreter by other means. However, as mentioned several times throughout this paper and also the title suggests is the state of this work preliminary. We yet have to show, for the general applicability of this work, that our implementation is competitive with known implementations in Prolog. As for our application on the Nao, we seem to have no other choice than to re-implement Golog. Besides our first results and the fact that the elevator examples works with our interpreter, we have to show that our implementation is correct. Also note that this implementation is naive, and a first quick approach to develop a Golog interpreter in Lua. In particular, we did not yet consider to use meta tables or Lua's closure mechanism for defining the BAT. In future implementations, these features may be taken into account as well as the possibility of integrating Golog language features directly into the Lua specification using Metalua [27], a meta language based on Lua.

As already mentioned, for our future work we need to enhance the reasoning engine and develop an interpreter for online Golog, incorporating features that have proved useful (e.g. cf. [12]). Furthermore, we have to show that our implementation is competitive with the Prolog implementation. Another important issue for the usability of Golog is an easy and neat syntax. The syntax presented here is contributed to the syntax of the associative arrays as provided by Lua.

We think that this representation resembles rather an abstracted syntax tree, and should become the representation for the back-end of our new interpreter. The front-end should make use of a regular programming syntax without "lots of silly curly brackets". Here we aim at using the LPEG library which is available for Lua [28]. This package provides interpreting *parsing expression grammars* (PEGs), which could be used to generate the intermediate code, which we presented in this paper. Also, in this step several optimisations can be undertaken to speed up the execution time of a Golog program such as pre-processing loop invariants, or generating tables for often used fluent values, to speed up regression. Another advantage of Lua is the availability of a fast C/C++ interface. It is rather easy to connect Lua with the rest of your robot system. Finally, we aim at using Golog as the standard high-level control language in the Fawkes framework which was recently released (`www.fawkesrobotics.org`). The idea with that is to find a larger robotics community that might be using Golog for encoding control programs.

References

1. Ingrand, F., Chatila, R., Alami, R., Rober, F.: PRS: A high level supervision and control language for autonomous mobile robots. In: Proc. ICRA-96. (1996)
2. Konolige, K., Myers, K., Ruspini, E., Saffiotti, A.: The Saphira architecture: A design for autonomy. JETAI **9**(1) (1997) 215–235
3. Bonasso, R., Firby, R., Gat, E., Kortenkamp, D., Miller, D., Slack, M.: Experiences with an architecture for intelligent, reactive agents. Journal of Experimental and Theoretical Artifical Intelligence **9**(2-3) (1997) 237–256
4. McDermott, D.: A reactive plan language. Technical Report YALEU/DCS-RR-864, Yale University, Department of Computer Science (1991)
5. Beetz, M.: Structured reactive controllers. Journal of Autonomous Agents and Multi-Agent Systems **2**(4) (2001) 25–55
6. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A Logic Programming Language for Dynamic Domains. J. of Logic Programming **31** (1997) 59–84
7. De Giacomo, G., Levesque, H., Sardiña, S.: Incremental execution of guarded theories. Computational Logic **2**(4) (2001) 495–525
8. Grosskreutz, H., Lakemeyer, G.: ccgolog – A logical language dealing with continuous change. Logic Journal of the IGPL **11**(2) (2003) 179–221
9. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00), AAAI Press (2000) 355–362
10. McCarthy, J.: Situations, Actions and Causal Laws. Technical report, Stanford University (1963)
11. Reiter, R.: Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)
12. Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains. Robotics and Autonomous Systems, Special Issue on Semantic Knowledge in Robotics **56**(11) (2008) 980–991

13. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua - An Extensible Extension Language. Software: Practice and Experience **26**(6) (Jan 1999) 635 – 652
14. Niemüller, T., Ferrein, A., Lakemeyer, G.: A lua-based behavior engine for controlling the humanoid robot nao. In: 2009 RoboCup Symposium. (2009)
15. Aldebaran Robotics: Website. http://www.aldebaran-robotics.com/ (2008)
16. Niemüller, T.: Developing A Behavior Engine for the Fawkes Robot-Control Software and its Adaptation to the Humanoid Platform Nao. Master's thesis, Knowledge-Based Systems Group, RWTH Aachen University (2009)
17. Pirri, F., Reiter, R.: Some contributions to the metatheory of the situation calculus. Journal of the ACM **46**(3) (1999) 325–361
18. Hähnel, D., Burgard, W., Lakemeyer, G.: GOLEX - bridging the gap between logic Golog and a real robot. In Herzog, O., Günter, A., eds.: KI-98: Advances in Artificial Intelligence. Volume 1504 of Lecture Notes in Computer Science., Springer (1998) 165–176
19. Levesque, H.J., Pagnucco, M.: Legolog: Inexpensive experiments in cognitive robotics. In: Proceedings of CogRob-00. (2000)
20. Soutchanski, M., Pham, H., Mylopoulos, J.: Decision making in uncertain real-world domains using dt-golog. In: Proc. AAAI-06. (2006)
21. Eyerich, P., Nebel, B., Lakemeyer, G., Claßen, J.: Golog and pddl: what is the relative expressiveness? In: PCAR '06: Proceedings of the 2006 international symposium on Practical cognitive agents and robots, ACM (2006)
22. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: The Evolution of Lua. In: Proceedings of History of Programming Languages III, ACM (2007) 2–1 – 2–26
23. The Debian Project: The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/ retrieved Jan 30th 2009.
24. Hirschi, A.: Traveling Light, the Lua Way. IEEE Software **24**(5) (2007) 31–38
25. De Giacomo, G., Lesperance, Y., Levesque, H.J.: Congolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121**(1-2) (2000) 109–169
26. Giarratano, J., Riley, G.: Expert Systems: Principles and Programming. fourth edition edn. Course Technology (2004)
27. Fleutot, F., Tratt, L.: Contrasting compile-time meta-programming in metalua and converge. In: Workshop on Dynamic Languages and Applications. (July 2007)
28. Medeiro, S., Ierusalimschy, R.: A parsing machine for PEGs. In: Proceedings of the 2008 Symposium on Dynamic Languages, ACM (2008) 1–12

Challenges for domestic service robots

Sven Behnke
University of Bonn, Germany

Benchmarking robotic systems is difficult. Robot competitions, such as RoboCup and the DARPA Grand and Urban Challenges provide a standardized test environment and allow for the direct comparison of different systems. For domestic service robots, in recent years, the RoboCup@Home league has been established. Domestic service tasks require three main skills from autonomous robots: robust navigation, mobile manipulation, and intuitive communication with the users. My team NimbRo developed the robot Dynamaid for these competitions. For robust navigation, Dynamaid has a base with four individually steerable differential wheel pairs, which allow omnidirectional motion. For mobile manipulation, Dynamaid is additionally equipped with two anthropomorphic arms that include a gripper, and with a trunk that can be lifted as well as twisted. For intuitive multimodal communication, the robot has a microphone, stereo cameras, and a movable head. Dynamaid can perceive persons and objects in its environment, recognize and synthesize speech. Together with our communication robot Robotinho, it won the Innovation award at RoboCup 2009.

Robust and efficient visual SLAM for spatial cognition

Calway, Andrew
University of Bristol

It has long been understood that autonomous exploration of previously unseen environments to gain spatial awareness and understanding will be a fundamental capability of future robotic systems. The development of techniques for simultaneous localisation and mapping (SLAM) - determining world structure and topology whilst simultaneously estimating position - has therefore been at the centre of robotics research for many years. This has been based on a variety of different sensors, including lasers, vision, GPS, odometry and ultrasonics. Of these, vision based techniques have perhaps the greatest potential given the low cost and small form factor of the sensor and the richness of the data. Recent years have seen significant advances in visual SLAM, driven in the main by the possibility of a highly portable position sensing device, especially in applications such as wearable computing. Several systems now exist which are capable of highly robust and efficient localisation and mapping, operating at rates in excess of standard video frame rates, giving genuine real-time capability. As such, they provide huge potential for advancing sensing capability in robotic systems and further as a catalyst for advancing spatial cognition, especially when coupled with other advances in computer vision, such as object and activity recognition. In this talk I will summarise some of these advances in visual SLAM, focusing on work carried out at Bristol on robustness to erratic motion, relocalisation in previously built maps, extracting higher order map structure and mechanisms for efficient map representation.

A Constraint-Based Approach for Plan Management in Intelligent Environments*

Federico Pecora and **Marcello Cirillo**
Center for Applied Autonomous Sensor Systems
Örebro University, SE-70182 Sweden
<name>.<surname>@oru.se

Abstract

In this paper we address the problem of realizing a service-providing reasoning infrastructure for proactive human assistance in intelligent environments. We propose SAM, an architecture which leverages temporal knowledge represented as relations in Allen's interval algebra and constraint-based temporal planning techniques. SAM seamlessly combines two key capabilities for contextualized service provision, namely human activity recognition and planning for controlling pervasive actuation devices.

Introduction

The problem we tackle in this paper is that of realizing a service-providing reasoning infrastructure for proactive human assistance in intelligent environments. Two key capabilities that are often desirable in a service-providing intelligent environment are (1) the ability to recognize activities performed by the human user, and (2) the ability to plan and execute the behavior of pervasive service-providing devices according to the indications of activity recognition.

Activity recognition has received much attention in the literature and the term has been employed to indicate a variety of capabilities. In this paper we take activity recognition to mean the ability of the intelligent system to deduce temporally contextualized knowledge regarding the state of the user on the basis of a set of heterogeneous sensor readings. Equipped with such a capability, an intelligent environment could be capable of proactively planning for and executing services that provide contextualized assistance. This requires a way to model the temporal and causal dependencies that exist between these tasks and the state of the human user. For instance, if a smart home could recognize that the human user is cooking, it could instruct a cleaning robot to avoid navigating to the the dining room until the subsequent dining activity is over.

This paper presents SAM, an Activity Management architecture[1] for service providing intelligent environments which achieves the two key capabilities mentioned above. SAM is built on top of the Multi-component Planning and Scheduling framework (OMPS) (Fratini, Pecora, and Cesta 2008). Specifically, in conjunction with an intelligent environment equipped with pervasive sensors and actuators, SAM provides the means to monitor the daily activities of a human being and to proactively assist the human through the environment's actuators. The architecture realizes an on-line abductive reasoning process on patterns of sensor observations provided by the intelligent environment, and is capable of synthesizing action plans for the environment's actuators in reaction to recognized human activities. As a direct result of the underlying framework, SAM retains three important properties: (1) the component-based domain description language provides a common formalism for expressing the activity recognition and proactive controller functionalities of the domain; (2) the constraint-based nature of the architecture allows to perform concurrent activity recognition, planning and execution; (3) the component-based nature of the framework allows to implement modular interfaces to the intelligent environment, thus supporting the incremental integration of new sensory/actuation elements.

Related Work

Current approaches to the problem of recognizing human activities can be roughly categorized as *data-driven* or *knowledge-driven*. In data-driven approaches, models of human behavior are learned from large volumes of data over time. Notable examples of this approach employ Hidden Markov Models (HMMs) for learning sequences of sensor observations with given transition probabilities, e.g., (Wu et al. 2007). Knowledge-driven approaches follow a complementary approach in which patterns of observations are modeled from first principles rather than learned. Such approaches typically employ an abductive processes, whereby sensor data is explained by hypothesizing the occurrence of specific human activities. Examples include reasoning approaches in which rich temporal representations are employed to model the conditions under which patterns of human activities occur (Jakkula, Cook, and Crandall 2007).

Data- and Knowledge-driven approaches have complementary strengths: the former provide an effective way to recognize elementary activities from large amounts of continuous data; conversely, knowledge-driven approaches are

*This paper appears in the *Proceedings of the Scheduling and Planning Applications Workshop (SPARK)*, held in conjunction with the 19th International Conference on Planning and Scheduling, 2009.

[1]SAM stands for "SAM the Activity Manager".

useful when the criteria for recognizing human activities are given by crisp rules that are clearly identifiable. In SAM, we follow the latter approach.

Also relevant to our work are various uses of schedule execution monitoring techniques for domestic activity monitoring presented in the literature, e.g., (Cesta et al. 2007; Pollack et al. 2003). An important difference with the above works lies in the fact that they employ pre-compiled (albeit highly flexible) schedules as models for human behavior. In the present work, we employ a planning process to actually instantiate such candidate schedules on-line.

SAM leverages the capability of OMPS to plan for state variables, a feature typical of several continuous planning approaches (Knight et al. 2001). In addition, SAM leverages the ability of OMPS to employ custom variable types. This has allowed us to build the sensing and actuation capabilities directly into new variable types which extend the state variable. In SAM, variables are not only used to represent elements of the domain, but also to implement active processes which operate concurrently with the continuous planning process, providing it with real world data obtained form the intelligent environment.

Lastly, SAM is related to the situation recognition approach described in (Dousson, Gaborit, and Ghallab 1993), which also employs temporal reasoning techniques to perform on-line recognition of temporal patterns of sensory events. Like SAM, the requirements for recognition are modeled as temporal relations in Allen's interval algebra, and both recognition and actuation are modeled within the same formalism. However, in SAM these two types of reasoning are integrated at the reasoning level in addition to being described by the same formalism. Also, while the former approach is limited to "triggering" events as a result of recognized situations, SAM allows to trigger the generation of a contingent plan whose elements are flexibly constrained to sensory events or recognized activities as they evolve in time.

Domain Representation

SAM is implemented within the OMPS temporal reasoning framework (Fratini, Pecora, and Cesta 2008). OMPS is a constraint-based planning and scheduling software API for developing temporal planning and scheduling applications, and has been used to develop a variety of decision support tools, ranging from highly-specialized space mission planning software to classical planning frameworks.

SAM leverages the domain description language provided by OMPS to model the dependencies that exist between sensor readings, the state of the human user, and tasks to be performed in the environment. In this section we describe how domains expressed in this formalism can be used to represent both requirements on sensor readings and on actuation devices. The following section will describe the actual implementation of SAM, i.e., how such domain descriptions are employed to infer the state of the user and to contextually synthesize action plans for actuators in the intelligent environment.

OMPS's domain description language is grounded on the notion of *component*. A component is an element of a domain theory which represents a logical or physical entity. Components model parts of the real world that are relevant for a specific decisional process, such as complex physical systems or their parts. Components can be used to represent, for example, a robot which can navigate the environment and grasp objects, or an autonomous refrigerator which can autonomously open and close its door.

An automated reasoning functionality developed in OMPS consists in a procedure for taking *decisions* on components. Decisions describe an assertion on the possible evolutions in time of a component. For instance, a decision on the fridge component described above could be to open its door no earlier than time instant 30 and no later than time instant 40. More precisely, a decision is an assertion on the value of a component in a given flexible time interval, i.e., a pair $\langle v, [I_s, I_e] \rangle$, where the nature of the value v depends on the specific component and I_s, I_e represent, respectively, an interval of admissibility of the start and end times of the decision. In the fridge example, assuming the door takes five seconds to open, the flexible interval is $[I_s = [30, 40], I_e = [34, 44]]$.

OMPS provides a number of built-in component types, among which consumable and re-usable multi-capacity resources, and state variables. The built-in state variable type of component instead models elements whose state in time is represented by a symbol. OMPS supports disjunctive values for state variables, e.g., a decision on a state variable that models a mobile robot could be $\langle \mathbf{navigate} \vee \mathbf{grasp}, [I_s, I_e] \rangle$, representing that the robot should be in the process of either navigating or grasping an object during the flexible interval $[I_s, I_e]$. For the purposes of this work, we focus on state variable type components and two custom components that have been developed in SAM to accommodate the needs of the physically instantiated nature of our application domain.

The core intuition behind OMPS is the fact that decisions on certain components may entail the need to assert decisions on other components. For instance, the decision to dock the robot to the fridge may *require* that the fridge door has already been opened. Such dependencies among component decisions are captured in a domain theory through what are called *synchronizations*. A synchronization is a set of requirements expressed in the form of temporal constraints. Such constraints in OMPS are bounded variants of the relations in the restricted Allen's Interval Algebra (Allen 1984; Vilain, Kautz, and van Beek 1989). Specifically, temporal constraints in OMPS enrich Allen's relations with bounds through which it is possible to fine-tune the relative temporal placement of constrained decisions. For instance, the constraint **A** DURING $[3, 5][0, \infty)$ **B** states that **A** should be temporally contained in **B**, that the start time of **A** must occur between 3 and 5 units of time after the beginning of **B**, and that the end time of **A** should occur some time before the end of **B**.

Figure 1(a) shows an example of how temporal constraints can be used to model requirements among actuators in an intelligent environment. The three synchronizations involve two components: a robotic table and an intelligent fridge (represented, respectively, by state variables

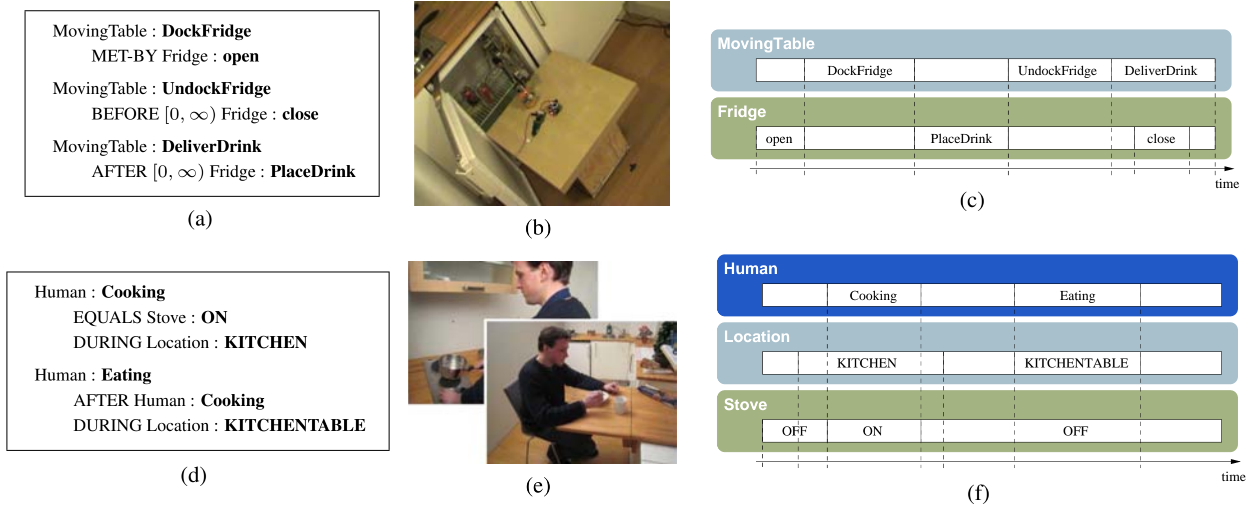


Figure 1: *Top row:* three synchronizations in a possible domestic robot planning domain (a), the corresponding real components available in our intelligent environment (b), and a possible timeline for the two components (c). *Bottom row:* two synchronizations in a possible domestic activity recognition domain (d), the corresponding situations as enacted by a test subject in a test environment (e), and a possible timeline for the three components (f).

MovingTable and Fridge). The MovingTable can dock and undock the Fridge, and navigate to the human user to deliver a drink. The Fridge component can open and close its door, as well as grasp a drink inside it and place it on a docked table. The above three synchronizations model three simple requirements of this domain, namely: (1) since the Fridge's door cannot open if it is obstructed by the MovingTable (see figure 1(b)), and we would like the door to be kept open only when necessary, docking the fridge must occur directly after the fridge door is opened (MET-BY constraint); (2) for the same reasons, the fridge door should close only after the MovingTable has completed the undocking procedure (BEFORE constraint); and (3) delivering a drink to the human is possible only after the drink has been placed on the table (AFTER constraint).

While temporal constraints express requirements on the temporal intervals of decisions, value constraints express requirements on the value of decisions. OMPS provides the VALUE-EQUALS constraint to model that two decisions should have equal value. For instance, asserting d_1 VALUE-EQUALS d_2 where the two decisions' values are, respectively, $\mathbf{v_1} = \mathbf{A} \vee \mathbf{B}$ and $\mathbf{v_2} = \mathbf{B} \vee \mathbf{C}$, will constrain the value of both decisions to be $\mathbf{B}$ (the intersection of possible values). As for temporal constraints, OMPS provides built-in propagation for value constraints.

Decisions and temporal constraints asserted on components are maintained in a *decision network* (DN), that is at all times kept consistent through *temporal propagation*. This ensures that the temporal intervals underlying the decisions are kept consistent with respect to the temporal constraints, while decisions are anchored flexibly in time. In other words, adding a temporal constraint to the DN will either result in the calculation of updated *bounds* for the intervals I_s, I_e for all decisions, or in a propagation failure, indicating that the added constraint or decision is not admissible. Temporal constraint propagation is a polynomial time operation, as it is based on a Simple Temporal Network (Dechter, Meiri, and Pearl 1991).

For each component in the domain, OMPS provides built-in methods to extract the *timeline* of the component. A timeline represents the behavior of a component in time as it is determined by the decisions and constraints imposed on this component in the DN. Figure 1(c) shows a possible timeline for the two components Fridge and MovingTable of the previous example. Notice that, in general, it is possible to extract many timelines for a component, as constraints bound decision start and end times flexibly. In the remainder of this paper we will always employ the earliest start time timeline, i.e., the timeline obtained by choosing the lower bound for all decisions' temporal intervals I_s, I_e.

In the previous example temporal constraints are used to model the requirements that exist between two "actuator components" (modeled as state variables) in carrying out the task of retrieving a drink from the fridge. In addition to actuators, however, state variables can be used to represent sensors in an intelligent environment, their values thus representing *sensor readings* rather than commands to be executed. Consequently, while temporal constraints among the values of actuator components represent temporal dependencies among commands to be executed that should be upheld in proactive service enactment, temporal constraints among "sensor components" represent temporal dependencies among sensor readings that are the result of specific human activities. For instance, the synchronizations in figure 1(d) describe possible conditions under which the human activities of **Cooking** and **Eating** can be inferred (where omitted, temporal bounds are assumed to be $[0, \infty)$). The synchronizations involve three components, namely a state variable representing the human inhabitant of the intelligent environment, a state variable representing a stove state sen-

sor, and another state variable representing the location of the human as it is determined by a person localization sensor in the environment. The synchronizations model how the relative occurrence of specific values of these components in time can be used as evidence of the human cooking or eating: the former is deduced as a result of the user being located in the **KITCHEN** (DURING constraint) and is temporally equal to the sensed activity of the Stove sensor; similarly, the requirement for asserting the **Eating** activity consists in the human being having already performed the **Cooking** activity (AFTER constraint) and his being seated at the **KITCHENTABLE**.

A unique feature of SAM is that the same formalism can be employed to express requirements both for enactment and for activity recognition. This is enabled by two specializations of the state variable component type, namely *sensor components* and *actuator components*. As we will see, a single inference algorithm based on temporal constraint reasoning provides a means to concurrently deduce context from sensor components and to plan for actuator components.

Recognizing Activities and Executing Proactive Services in SAM

SAM employs three types of components: *state variables*, *sensors* and *actuators*. State variables are employed to model one or more aspects of the user's activities of daily living. For instance, in the examples that follow we will use a state variable whose values are $\{\mathbf{Cooking}, \mathbf{Eating}, \mathbf{InBed}, \mathbf{WatchingTV}, \mathbf{Out}\}$ to model the human user's domestic activities. Sensors and actuators are specialized variants of the built-in state variable type which implement an interface between the real-world sensing and actuation modules and the DN. Sensor components interpret data obtained from the physical sensors deployed in the intelligent environment and represent this information as decisions and constraints in the DN. Actuators are components that trigger the execution on a real actuator of a planned decision. Actuators also have a sensing capability which allows to update the DN with relations that model the temporal bounds of execution of the executed operations.

In SAM, the DN acts as a "blackboard" where decisions and constraints re-construct the reality observed by sensor components as well as the current hypothesis on what the human being is doing. This hypothesis is deduced by a continuous re-planning process which attempts to infer new possible states of the human being and any necessary actuator plans.

SAM is implemented as a multitude of concurrent processes (described in detail in the following sections), each operating continuously on the DN:

Sensing processes: each sensor is a process that adds decisions and constraints to represent the real-world observations provided by the intelligent environment.

Inference process: the current DN is manipulated by the continuous inference process, which adds decisions and constraints that model the current activity performed by the user and any proactive support operations to be executed by the actuators.

Actuator processes: actuators ensure that decisions in the DN that represent operations to be executed are dispatched as commands to the real actuators and that termination of actuation operations are reflected in the DN as they are observed in reality.

These processes add decisions and constraints to the DN in real-time, and access to the DN is scheduled by an overall process scheduler. Each process modifies the DN, thus triggering constraint propagation.

Continuous Inference Process

SAM's continuous inference process relies on the fact that the DN represents at all times the current situation in the real world, possesses two key capabilities: (1) to assess whether the DN contains evidence of sensed values in a given time interval; and (2) to assess whether the DN contains the requirements described in a particular synchronization. Both capabilities can be viewed as ways to *support* candidate decisions. Supporting a decision means performing one of the two following steps:

Unification. A decision is supported by unification if it is possible to impose a temporal EQUALS constraint and a VALUE-EQUALS constraint between it and another decision which is already supported. If the result of imposing these two constraints is successful, then this is an indication that indeed there is an interval of time in which the value of the decision to support has been sensed in the real environment. SAM can therefore "query" the DN to assess whether a value $\mathbf{v}$ has been sensed in a certain interval of time $[I_s, I_e]$ by attempting to support through unification a decision $\langle \mathbf{v}, [I_s, I_e] \rangle$.

Expansion. Expanding a synchronization entails that new (unsupported) decisions and constraints are added to the DN as prescribed by the requirements of the synchronization. Support for these new decisions is sought by recursively expanding other synchronizations or unifying the new decisions with others already present in the DN. Overall, expansion is how SAM assesses whether the current situation of sensor readings in the DN can support a particular hypothesis: it adds an unsupported decision representing the current hypothesis (e.g., that the human being is cooking), and tries to support it through the domain theory and existing sensed values in the DN.

The continuous re-planning process implemented in SAM is shown in procedure `Replan`. The procedure leverages unification and expansion to continuously attempt to support decisions which represent hypotheses on the state of a number of *monitored* components. These components are all those components for which we wish SAM to deduce their current state. In our specific application domain, all these components are state variables which model some aspect of the human user's state. For each monitored component, the procedure adds to the DN a decision whose value is a disjunction of all its possible values (lines 2–3). For instance, if the component in question is the state variable Human described previously, then the new decision to be added will be $d_{hyp}^{\text{Human}} = \langle (\mathbf{Cooking} \vee \mathbf{Eating} \vee \mathbf{InBed} \vee$

$\mathbf{WatchingTV} \vee \mathbf{Out}), [[0, \infty), [0, \infty)]\rangle$. This decision is marked as un-supported (line 4), i.e., it constitutes a hypothesis on the current activity in which the human user is engage in. The procedure then constrains this decision to occur after any other decisions on the same component (lines 5–6). This is done in order to avoid that the new decision is trivially supported by unification with a decision supported in a previous call to the `Replan` procedure. Finally, the procedure triggers a decision supporting algorithm which attempts to support the newly added decisions by recursively expanding synchronizations and unifying the resulting requirements (line 7). In the process of supporting new decisions, their values will be constrained (by VALUE-EQUALS constraints) to take on a specific value. For instance, if the domain theory contains a synchronization stating that the requirements for **Eating** on component Human are a certain set of values on some sensor components, then the unsupported decision is marked as supported, the unary constraint d_{hyp}^{Human} VALUE-EQUALS **Eating** is imposed, and new (un-supported) decisions on the sensor components are added to the DN.

Procedure `Replan` (*DN*)

1 **foreach** $c \in$ *MonitoredComponents* **do**
2 $\quad \mathbf{v} \leftarrow \bigvee_{v_i \in \text{possibleValues}(c)} v_i$
3 $\quad \text{DN} \leftarrow \text{DN} \cup d_{hyp}^{c} = \langle \mathbf{v}, [I_s, I_e] \rangle$
4 $\quad$ mark d_{hyp}^{c} as not supported
5 $\quad$ **foreach** d *on component* c **do**
6 $\quad\quad \text{DN} \leftarrow \text{DN} \cup d_{hyp}^{c}$ AFTER $[0, \infty)$ d
7 `SupportDecisions` (*DN*)

If the decision supporting algorithm terminates successfully, the DN contains the new decisions that have been added by the re-planning procedure, plus all those decisions and constraints that implement support for these decisions. The value of each newly supported decision on monitored components has been constrained to be that required by the synchronization that was used by the `SupportDecisions` procedure. Since these decisions are linked by temporal constraints to decisions on sensor components, their placement in time will follow the evolution of the DN's decisions on sensor components as time progresses.

If `SupportDecisions` fails, the resulting DN is identical to before the re-planning procedure was started, therefore reflecting the fact that no new information was deduced.

Note that the continuous `SupportDecisions` procedure is greedy, in that the first successfully applicable synchronization is selected in support of current sensor readings.

Sensing Processes

In order to realize the interface between OMPS and real-world sensors in the intelligent environment, a new component, the *sensor*, was developed in SAM. A sensor is modeled in the domain for each physical sensor in the intelligent environment. Each sensor component is provided with an interface to the physical sensor, as well as the capability to periodically update the DN with decisions and constraints that model the state of the physical sensor. The process for updating the DN is described in procedure `UpdateSensorValues`. Specifically, each sensor component's sensing procedure obtains from the DN the decision that represents the value of the sensor at the previous iteration (line 1). This decision, if it exists, is the decision whose end time has an infinite upper bound (u_e). No such decision exists if at the previous iteration the sensor readings were undetermined (d is null, i.e., there is no information on the current sensor value in the DN). The procedure then obtains the current sensor reading from its interface to the physical sensor (line 2). Notice that this could also be undetermined (null in the procedure), as a sensor may not provide a reading at all. At this point, three situations may occur.

Procedure `UpdateSensorValues` (DN,t_{now})

1 $d \leftarrow \langle \mathbf{v}, [[l_s, u_s], [l_e, u_e]] \rangle \in \text{DN}\ \ s.t.\ \ u_e = \infty$
2 $\mathbf{v_s} \leftarrow$ `ReadSensor()`
3 **if** $d = \text{null}\ \wedge\ \mathbf{v_s} \neq \text{null}$ **then**
4 $\quad \text{DN} \leftarrow \text{DN} \cup d' = \langle \mathbf{v_s}, [[0, \infty), [0, \infty)] \rangle$
5 $\quad \text{DN} \leftarrow \text{DN} \cup d'$ RELEASE $[t_{\text{now}}, t_{\text{now}}]$
6 $\quad \text{DN} \leftarrow \text{DN} \cup d'$ DEADLINE $[t_{\text{now}} + 1, \infty]$
7 **else if** $d \neq \text{null}\ \wedge\ \mathbf{v_s} = \text{null}$ **then**
8 $\quad \text{DN} \leftarrow \text{DN} \cup d$ DEADLINE $[t_{\text{now}}, t_{\text{now}}]$
9 **else if** $d \neq \text{null}\ \wedge\ \mathbf{v_s} \neq \text{null}$ **then**
10 $\quad$ **if** $\mathbf{v_s} = \mathbf{v}$ **then**
11 $\quad\quad \text{DN} \leftarrow \text{DN} \cup d$ DEADLINE $[t_{\text{now}} + 1, \infty]$
12 $\quad$ **else**
13 $\quad\quad \text{DN} \leftarrow \text{DN} \cup d$ DEADLINE $[t_{\text{now}}, t_{\text{now}}]$
14 $\quad\quad \text{DN} \leftarrow \text{DN} \cup d' = \langle \mathbf{v_s}, [[0, \infty), [0, \infty)] \rangle$
15 $\quad\quad \text{DN} \leftarrow \text{DN} \cup d'$ RELEASE $[t_{\text{now}}, t_{\text{now}}]$
16 $\quad\quad \text{DN} \leftarrow \text{DN} \cup d'$ DEADLINE $[t_{\text{now}} + 1, \infty]$

New sensor reading. If the DN does not contain an unbounded decision and the physical sensor returns a value, then a decision is added to the DN representing this (new) sensor reading. The start time of this decision is anchored to the current time t_{now} by means of a RELEASE constraint and made to have an unbounded end time (lines 3–6). If the DN contains an unbounded decision that differs from the sensor reading, then the procedure models this fact in the DN as above, and in addition "stops" the previous decision by imposing a DEADLINE constraint, i.e., anchoring the decision's end time to t_{now} (lines 9, 12–16).

Continued sensor reading. If the DN contains an unbounded decision and the physical sensor returns the same value as that of this decision, then the procedure ensures that the increased duration of this decision is reflected in the DN. It does so by updating the lower bound of the decision's end time to beyond the current time by means of a new DEADLINE constraint (lines 9–11). Notice that this ensures that at the next iteration the DN will contain an unbounded decision.

Interrupted sensor reading. If the DN contains an unbounded decision and the physical sensor returns no read-

ing ($\mathbf{v_s}$ is null), then the procedure simply interrupts the unbounded decision by bounding its end time to the current time with a DEADLINE constraint (lines 7–8).

Actuation Processes

The inference procedure implemented in SAM continuously assesses the applicability of given synchronizations in the current DN by asserting and attempting to support new decisions on monitored components, such as the Human state variable presented earlier. This same mechanism allows to obtain contextualized plan synthesis capabilities through the addition of synchronizations that model how actions carried out by actuators should be temporally related to recognized activities. For instance, in addition to requiring that **Cooking** should be supported by requirements such as "being in the kitchen" and "using the stove", a requirement involving an actuator component can be added, such as "turn on the ventilation over the stove". More in general, for each actuation-capable device in the intelligent environment, an actuator component is modeled in the domain. This component's values represent the possible commands that can be performed by the device. In the domain, these values are added as requirements to the synchronizations of monitored components. As sensor components interface the real world to represent sensor readings in the DN, actuator components interface the real world to trigger commands to real actuators when decisions involving them appear in the DN.

However, it should be noticed that robotic devices are only partially controllable, in that we do not have strict guarantees on when and for how long given commands will be executed. For this reason, actuator components also possess a sensory capability that is employed to feed information on the status of command execution back into the DN. As sensor components, actuator components write this information directly into the DN, thus allowing the re-planning process to take into account the current state of execution of the actions.

Procedure `UpdateExecutionState`(DN,t_{now})

1 $D \leftarrow \{\langle \mathbf{v}, [[l_s, u_s], [l_e, u_e]] \rangle \in \text{DN} : l_s \leq t_{\text{now}}, u_e = \infty\}$
2 **foreach** $d \in D$ **do**
3   **if** `IsExecuting`($\mathbf{v}$) **then**
4   DN $\leftarrow$ DN $\cup$ d DEADLINE $[t_{\text{now}} + 1, \infty]$
5   **else if** $l_s = l_e$ **then**
6   `StartExecuting`($\mathbf{v}$)
7   DN $\leftarrow$ DN $\cup$ d RELEASE $[t_{\text{now}}, t_{\text{now}}]$
8   **else** DN $\leftarrow$ DN $\cup$ d DEADLINE $[t_{\text{now}}, t_{\text{now}}]$

Actuators execute concurrently with the re-planning and sensing operations described above. The operations performed by actuators are shown in procedure `UpdateExecutionState`. Each actuator component first identifies all decisions that have an unbounded end time and whose earliest start time falls before or at the current time (line 1). The fact that these decisions are unbounded indicates that they have been planned for execution and their execution has not yet terminated. The fact that their start time lies before or at the current time indicates that they are scheduled to start or have already begun. For each of these decisions, the physical actuator is queried to ascertain whether the corresponding command is being executed. If so, then the decision is constrained to end at least one time unit beyond the current time (lines 3–4). If the command is not currently in execution, the procedure checks whether the command still needs to be issued to the physical actuator. This is the case if the earliest start and end times of the decision coincide (because the decision's end time was never updated at previous iterations). The procedure dispatches the command to the actuator and anchors the start time of the decision to the current time (lines 5–7). Conversely, if the decision's start and end times do not coincide, then the decision is assumed to be ended, and the procedure imposes the current time as its earliest and latest end time (line 8).

Case Studies in the PEIS-Home

We illustrate the use of sensor components in SAM with four runs performed in the PEIS-Home, a prototypical intelligent environment deployed at the at Örebro University (see `aass.oru.se/~peis`). The environment provides ubiquitous sensing and actuation devices, including the robotic table and intelligent fridge described in earlier examples.

In the first run our aim is to assess the sleep quality of a person by tracking how many times and for how long the user turns on his night light when he lies in bed. For this purpose, we employ three physical sensors: a pressure sensor, placed beneath the bed, a luminosity sensor placed close to the night light, and a person tracker based on stereo vision. We then define a domain with three sensor components and the two synchronizations shown in figure 2. Note

1) Human : **InBed**	2) HumanAbstract : **Awake**
DURING Location : **NOPOS**	DURING Human : **InBed**
EQUALS Bed : **ON**	EQUALS NightLight : **ON**

Figure 2: Synchronizations defined in our domain for the Human and HumanAbstract components to assess quality of sleep.

that the human user is modeled by means of two distinct components, Human and HumanAbstract. This allows us to reason at different levels of abstraction on the user: while the decisions taken on component Human are always a direct consequence of sensor readings, synchronizations on values of HumanAbstract describe knowledge that can be inferred from sensor data as well as previously recognized Human and HumanAbstract activities. The first synchronization models two requirements for recognizing that the user has gone to bed: first, the user should not be observable by the tracking system, since the bedroom is a private area of the apartment and, therefore, outside the field of view of the cameras; second, the pressure sensor beneath the bed should be activated. The resulting **InBed** decision has a duration EQUAL to the one of the positive reading of the bed sensor. The second synchronization grasps the situation in which, although lying in bed, the user is not sleeping. The decision **Awake** on the component HumanAbstract depends therefore on the decision **InBed** of the Human and on the sensor readings of NightLight.

This simple domain was employed to test SAM in our intelligent home environment with a human subject. The overall duration of the experiment was 500 seconds, with the concurrent inference and sensing processes operating at a rate of about 1 Hz. Figure 5 (a) is a snapshot of the five components' timelines at the end of the run (from top to bottom, the three sensors and the two monitored components).

1) HumanAbstract : **Lunch**
STARTED-BY Human : **Cooking**
FINISHED-BY Human : **Eating**
DURING Time : **afternoon**

2) HumanAbstract : **Nap**
AFTER HumanAbstract : **Lunch**
EQUALS Human : **WatchingTV**

3) Human : **Cooking**
DURING Location : **KITCHEN**
EQUALS Stove : **ON**

4) Human : **WatchingTV**
EQUALS Location : **COUCH**

5) Human : **Eating**
DURING Location : **KITCHENTABLE**
EQUALS KTRfid : **DISH**

Figure 3: Synchronizations modeling afternoon activities of the human user.

The outcome of a more complex example is shown in figure 5 (b). In this case the scenario contains four instantiated sensors. Our goal is to determine the afternoon activities of the user living in the apartment, detecting activities like **Cooking**, **Eating** and the more abstract **Lunch**. To realize this example, we define five new synchronizations (figure 3), three for the Human component and two for the HumanAbstract component. Synchronization (3) identifies the human activity **Cooking**: the user should be in the kitchen and its duration is EQUAL to the activation of the Stove. Synchronization (5) models the **Eating** activity, using both the Location sensor and an RFID reader placed beneath the kitchen table (component KTRfid). A number of objects have been tagged to be recognized by the reader, among which dishes whose presence on the table is required to assert the decision **Eating**. The last synchronization for the Human component (4) correlates the presence of the user on the couch with the activity of **WatchingTV**.

Synchronizations (1) and (2) work at a higher level of abstraction. The decisions asserted on HumanAbstract are inferred from sensor readings (Time), from the Human component and from the HumanAbstract component itself. This way we can identify complex activities such as **Lunch**, which encompasses both **Cooking** and the subsequent **Eating**, and we can capture the fact that after lunch the user, sitting in front of the TV, will most probably fall asleep.

Also this example was executed in the PEIS-Home. It is worth mentioning that the decision corresponding to the **Lunch** activity on the HumanAbstract component was identified only when both **Cooking** and **Eating** were asserted on the Human component. Also it can be noted that **Nap** is identified as the current HumanAbstract activity only after the lunch is over and that on the first occurrence of **WatchingTV**, **Nap** was not asserted because it lacked support form the **Lunch** activity.

As an example of how the domain can include actuation as synchronization requirements on monitored components, let us consider the following run of SAM in a setup which includes the robotic table and autonomous fridge devices described earlier.

1) Human : **WatchingTV**
EQUALS Location : **COUCH**
START MovingTable : **DeliverDrink**

2) MovingTable : **DockFridge**
MET-BY Fridge : **OpenDoor**

3) MovingTable : **DeliverDrink**
AFTER Fridge : **PlaceDrink**

4) MovingTable : **UndockFridge**
BEFORE Fridge : **CloseDoor**

5) Fridge : **PlaceDrink**
MET-BY MovingTable : **DockFridge**
MEETS MovingTable : **UndockFridge**

6) OpenDoor : **OpenDoor**
MET-BY Fridge : **GraspDrink**

Figure 4: Synchronizations defining temporal relations between human activities and proactive services.

As shown in figure 4, we use abductive reasoning to infer when the user is watching TV. In this case, however, we modify the synchronization (4) presented figure 3 to include the actuators in the loop. The new synchronization (figure 4, (1)), not only recognizes the **WatchingTV** activity, but also asserts the decision **DeliverDrink** on the MovingTable component. This decision can be supported only if it comes AFTER another decision, namely **PlaceDrink** on component Fridge (synchronization (3)). When SAM's replanning procedure attempts to support **WatchingTV**, synchronization (5) is called into play, stating that **PlaceDrink** should occur right after (MET-BY) the MovingTable has docked the Fridge and right before the undocking maneuver (MEETS). The remaining three synchronizations — (2), (4) and (6) — are attempted to complete the chain of support, that is, the Fridge should first grasp the drink with its robotic arm, then open the door before the MovingTable is allowed to dock to it, and finally it should close the door right after the MovingTable has left the docking position.

This chain of synchronizations leads to the presence in the DN of a plan to retrieve a drink from the fridge and deliver it to the human who is watching TV. Notice that when the planned decisions on the actuator components are first added to the DN, their duration is minimal. Through the actuators' `UpdateExecutionState` procedure, these durations are updated at every re-planning period until the devices that are executing the tasks signal that execution has completed. Also, thanks to the continuous propagation of the constraints underlying the plan, decisions are appropriately delayed until their earliest start time coincides with the current time. A complete run of this scenario was performed in our intelligent environment and a snapshot of the final timelines is shown in figure 5 (c).

Conclusions and Future Work

In this paper we have presented SAM, an architecture for concurrent activity recognition, planning and execution. The architecture builds on the OMPS temporal reasoning framework, and leverages its component-based approach to realize a decisional framework that operates in a closed loop with physical sensing and actuation components in an intelligent environment. We have demonstrated the feasibility of the approach with a number of experimental runs in a real environment with a human test subject.

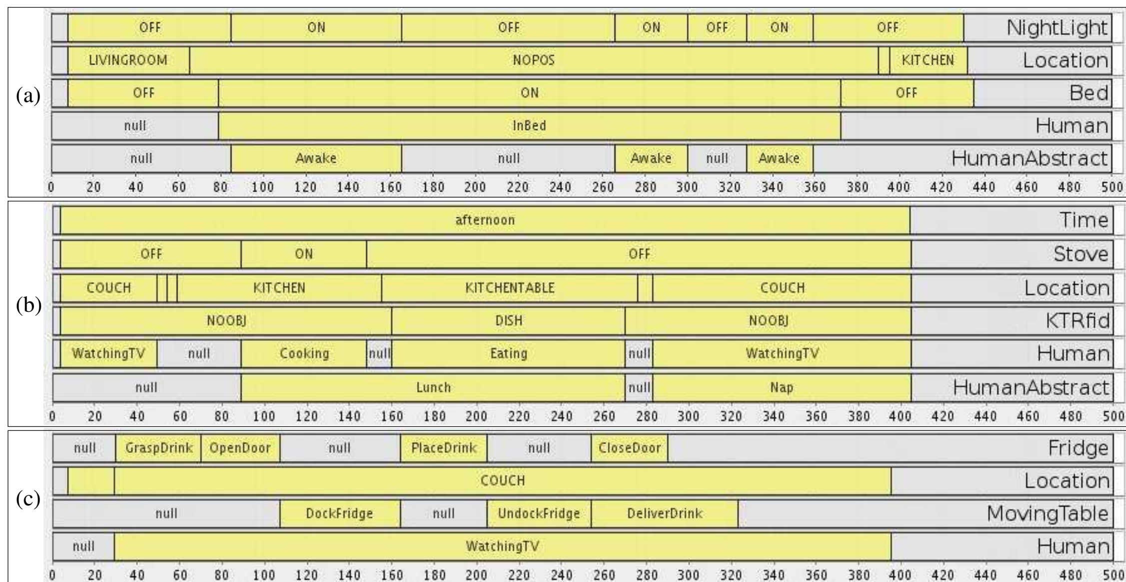


Figure 5: Timelines resulting from the runs performed in our intelligent home using the sleep monitoring (a), afternoon activities (b) and proactive service (c) domains.

One of SAM's current limitations is its relatively simple depth-first search strategy. A more sophisticated re-planning strategy would allow to take into account domains in which more than one synchronization is applicable to support a hypothesis, thus leading to different timelines for the same component. These synchronizations could model, for instance, alternative "explanations" for patterns of sensor readings, or alternative plans that realize different forms of support. Alternative synchronizations on the same values could also enable the synthesis of contingency plans for dealing with actuator execution failures. However, this would inevitably affect the performance of the re-planning procedure, which we have purposefully kept simple in order to maintain re-planning time within the limit of acceptable sampling rates. A first step in the direction of obtaining a performant re-planning procedure is presented in (Ullberg, Loutfi, and Pecora 2009), which details the performance as well as completeness and correctness proofs of SAM's activity recognition functionality.

Acknowledgements. The Authors wish to thank Alessandro Saffiotti for his support as well as the anonymous reviewers for their helpful comments.

References

Allen, J. 1984. Towards a general theory of action and time. *Artificial Intelligence* 23(2):123–154.

Cesta, A.; Cortellessa, G.; Giuliani, M.; Pecora, F.; Scopelliti, M.; and Tiberio, L. 2007. Caring About the User's View: The Joys and Sorrows of Experiments with People. In *ICAPS07 Workshop on Moving Planning and Scheduling Systems into the Real World*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artif. Intell.* 49(1-3):61–95.

Dousson, C.; Gaborit, P.; and Ghallab, M. 1993. Situation recognition: Representation and algorithms. In *Proc. of 13th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 166–174.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):231–271.

Jakkula, V.; Cook, D.; and Crandall, A. 2007. Temporal pattern discovery for anomaly detection in a smart home. In *Proc. of the 3rd IET Conf. on Intelligent Environments(IE)*, 339–345.

Knight, R.; Rabideau, G.; Chien, S.; Engelhardt, B.; and Sherwood, R. 2001. Casper: Space exploration through continuous planning. *IEEE Intelligent Systems* 16(5):70–75.

Pollack, M.; Brown, L.; Colbry, D.; McCarthy, C.; Orosz, C.; Peintner, B.; Ramakrishnan, S.; and Tsamardinos, I. 2003. Autominder: an intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems* 44(3-4):273–282.

Ullberg, J.; Loutfi, A.; and Pecora, F. 2009. Towards Continuous Activity Monitoring with Temporal Constraints. In *Proc. of the 4th Workshop on Planning and Plan Execution for Real-World Systems at ICAPS09*. (to appear).

Vilain, M.; Kautz, H.; and van Beek, P. 1989. Constraint propagation algorithms for temporal reasoning: A revised report. In Weld, D., and de Kleer, J., eds., *Readings in Qualitative Reasoning about Physical Systems*, 373–381. Morgan Kaufmann.

Wu, J.; Osuntogun, A.; Choudhury, T.; Philipose, M.; and Rehg, J. 2007. A Scalable Approach to Activity Recognition Based on Object Use. In *Proceedings of ICCV 2007. Rio de Janeiro, Brazil*.

Detecting humans activities in video and still images

Hlavac, Vaclav
Czech Technical University

The method for recognizing human actions based on pose primitives will be presented. In learning mode, the parameters representing poses and activities are estimated from videos. In run mode, the method can be used both for videos or still images. For recognizing pose primitives, we extend a histogram of Oriented Gradient (HOG) based descriptor to better cope with articulated poses and cluttered background. Action classes are represented by histograms of poses primitives. For sequences, we incorporate the local temporal context by means of n-gram expressions. Action recognition is based on a simple histogram comparison. Unlike the mainstream video surveillance approaches, the proposed method does not rely on background subtraction or dynamic features and thus allows for action recognition in still images. (Joint work with Chritian Thurau)

Context and place categorization for assistive robotics

Jim Little
Computer Science
University of British Columbia

Online annotated image databases are increasingly common. We develop a spatial-semantic modeling system that learns object-place relations from such databases, and then apply these relations to real-world tasks. In the home such a system can label novel scenes with place information, as we demonstrate on test scenes drawn from the same source as our training set. We have designed our system for future enhancement of a robot platform that performs state-of-the-art object recognition and creates object maps of realistic environments. In this context, we demonstrate the use of spatial-semantic information to perform clustering and place labeling of object maps obtained from real homes and offices. Then place information feeds back into the robot system to inform an object search planner about likely locations of a query object.

Modeling the Observed Behavior of a Robot through Machine Learning

INRIA

Artificial systems are becoming more and more complex, almost as complex in some cases as natural systems. Up to now, the typical engineering question was "*how do I design my system to behave according to some specifications*". However, the incremental design process is leading to so complex artifacts that engineers are more and more addressing a quite different issue of "*how do I model the observed behavior of my system*". Engineers are faced with the same problem as scientists studying natural phenomena. It may sound strange for an engineer to engage in observing and modeling what a system is doing, since this should be inferable from the models used in the system's design stage. However, a modular design of a complex artifact develops only local models that are combined on the basis of some composition principle of these models; it seldom provides global behavior models.

These general remarks hold in computer sciences throughout several examples of complex systems, ranging from multi-core processors to internet networks. This talk will illustrate the global approach of observation and modeling on the problem of understanding and predicting the behavior of a mobile robot.

Robots are becoming very complex, with a large number of sensory-motor functions combining dozens of actuators and sensors, offering the capabilities of many navigation and manipulation skills, and allowing the execution of sophisticated tasks. The design of these robots usually relies on some reasonable assumptions about the environment and does not model explicitly changing, open-ended environments with human interaction. Hence, a precise observation model of a given robot behavior in a varying and open environment can be essential for understanding how the robot operates within that environment, for predicting its behavior and for improving it.

Machine learning techniques are developed for acquiring the behavior models we are seeking. Three different approaches will be illustrated. In the first approach we learn from experience very robust ways of performing a high-level task such as "*navigate to*". The designer specifies a collection of *skills* represented as *hierarchical tasks networks*, whose primitives are sensory-motor functions. The skills provide different ways of combining these sensory-motor functions to achieve the desired task. The specified skills are assumed to be complementary and to cover different situations. The relationship between control states, defined through a set of task-dependent features, and the appropriate skills for pursuing the task is learned as a finite observable Markov decision process. This MDP provides a general policy for the task; it is independent of the environment and characterizes the abilities of the robot for the task.

In the second and third approaches, we learn from observations and we model as stochastic automata the behavior of the robot in performing a given task. We use two different techniques:

- Hidden Markov models, where part of the learning problems are how to acquire the finite

observation space and the finite state space;

- Dynamic Bayes networks, that can be less readable from a user's point of view, but that are used to improve online the robot behavior.

The talk will survey these approach, the tradeoffs, advantages and complexity of each approach, how the robotics experiments have been carried out, and the obtained results. The details of this research pursued jointly with several colleagues and PhD students can be found out in particular in the following publications.

M.FOX, M.GHALLAB, G.INFANTES, D.LONG. Robot introspection through learned hidden Markov models. *Artificial Intelligence*, 170(2): 59-113, Feb. 2006

B.MORISSET, M.GHALLAB, Learning how to Combine Sensory-Motor Functions into a Robust Behavior. *Artificial Intelligence*, 172(4-5): 392-412, March 2008

G. INFANTES, F. INGRAND, M. GHALLAB, Learning Behaviors Models for Robot Execution Control. Proc. 17th European Conference on Artificial Intelligence ECAI 2006, Aug. 2006

G.INFANTES , F.INGRAND , M.GHALLAB. Learning behavior models for robot execution control. 16th International Conference on Automated Planning and Scheduling (ICAPS), Anableside (GB), 6-10 June 2006, pp.394-397

Combining qualitative modelling and trial-and-error learning for skill acquisition

Sammut, Claude
Univ. of New South Wales

Pure reinforcement learning does not scale well to domains with many degrees of freedom and particularly to continuous domains. We introduce a hybrid method in which a symbolic planner constructs an approximate solution to a control problem. Subsequently, a numerical optimisation algorithm is used to refine the qualitative plan into an operational policy. The method is demonstrated on the problem of learning a stable walking gait for a bipedal robot.

Attentive Monitoring and Adaptive Control in Cognitive Robotics

E. Burattini, A. Finzi, S. Rossi and **M. Staffa**
Universita' degli Studi di Napoli "Federico II" - Italy.
email: {ernb, finzi, srossi}@na.infn.it, mariacarla.staffa@unina.it

Abstract

In this work, we present an attentional system for a robotic agent capable of adapting its emergent behavior to the surrounding environment and to its internal state. In this framework, the agent is endowed with simple attentional mechanisms regulating the frequencies of sensory readings and behavior activations. The process of changing the frequency of sensory readings is interpreted as an increase or decrease of attention towards relevant behaviors and particular aspects of the external environment. In this paper, we present our framework discussing several case studies considering incrementally complex behaviors and tasks.

Introduction

An autonomous robotic agent is expected to operate in complex dynamic environments by continuously monitoring the internal processes and the external environment. The robot executive system is to coordinate different low-level strategies (such as obstacles avoidance, walls follow, gates crossing, etc.) with high-level activities (such as achieving a goal, picking up an object, etc.), giving them, from time to time, different priority values both for allocation of resources and for action selection processes. The low-level activities are usually safety critical and are managed in a reactive way. On the other hand, high-level activities are generally achieved by processing more complex tasks, and, therefore, require high computational costs for both the inputs processing and data acquisition from the environment.

In this context, attentional mechanisms balancing sensory elaboration and actions execution can play a crucial role. In particular, attentional mechanisms have two main roles: direct sensors towards the most salient sources of information; filter the available sensory data to prevent unnecessary information processing. As a result of the application of these mechanisms, the robot behavior should be enhanced: the robot is to react faster to task-related or safety critical stimuli because processing resources are focused on not relevant stimuli.

Attentional mechanisms applied to autonomous robotic systems have been proposed elsewhere (e.g. (Mitsunaga and Asada 2002; Carbone et al. 2008; Frintrop, Jensfelt, and Christensen 2006)), mainly for vision-based robotics. In contrast, in our work, we are interested in artificial attentional processes suitable for the executive control. In particular, our aim is to provide a kind of supervisory attentional system (Norman and Shallice 1986; Cooper and Shallice 2000) capable of monitoring and regulating multiple concurrent behaviors at different level of abstraction. The notion of *divided attention* (Kahneman 1973) suggests that a limited amount of attention is allocated to tasks, with the resources involved in multi-task performances, and can be available in graded quantity. In an artificial setting, this can be obtained by introducing suitable scheduling mechanisms.

In this work, we present a behavior-based control architecture endowed with attentional mechanisms which are based on periodic releasing mechanisms of activations (Burattini and Rossi 2007; 2008). In this context, each behavior is equipped with an adaptive internal clock that regulates the sensing rate and the resulting action activations. The process of changing the frequency of sensory readings is interpreted as an increase or decrease of attention towards relevant behaviors and particular aspects of the external environment: the higher is the frequency, the higher is the resolution at which a process is monitored and controlled. Here, we present our framework providing several case studies where we discuss the effectiveness of the approach considering its scalability and the adaptivity with respect to different environments and tasks.

Attentive Executive Control

Our goal is to develop a behavior-based control system endowed with attentional mechanisms which focus sensory acquisitions and processing and modulates behaviors activations. The executive system should be enhanced with a supervisory attentional system (Norman and Shallice 1986) to suitably combine delibertive and reactive activities, monitoring and regulating multiple concurrent behaviors (Kahneman 1973). Our working hypothesis is that attentional behaviors are affected by internal self-regulating mechanisms and external sources of salience. The attentional global behavior should emerge from the interrelation of the attentional mechanisms associated with each single behavior.

Design Principles

The attentional control system we consider in this work combines the following design principles:

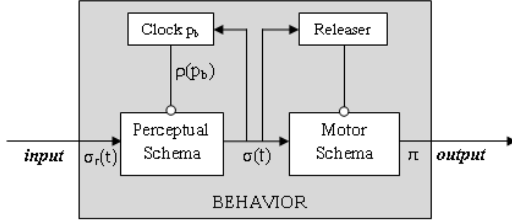


Figure 1: Each behavior is composed of an adaptive clock, a releasing function, a perceptual schema and a motor schema.

- *Behavior-based control system*. The attentional control is obtained from the interaction of a set of multiple parallel attentional behaviors working at different levels of abstraction.
- *Attentional monitoring*. Attentional mechanisms are to focus monitoring and control activities on relevant internal behaviors and external stimuli.
- *Internal and external sources of salience*. The sources of salience are behavior and task dependent; these can dependent by either internal states (e.g. hunger, fear) or external stimuli (e.g. obstacles, unexpected variations of the environment).
- *Adaptive sensory readings*. For each behavior, the process of changing the rate of sensory readings is interpreted as an increase or decrease of attention towards a particular aspect of the environment the robotic system is interacting with: the higher is the frequency, the higher the resolution at which an activity is monitored and regulated.
- *Emergent attentive behavior*. The overall attentional behavior should emerge from the interrelations of the attentive mechanisms associated with the behaviors.

Attentive Monitoring in the AIRM Architecture

In (Burattini and Rossi 2007; 2008), we connected the concept of IRM (Innate Releasing Mechanisms) (Lorenz 1991; Tinbergen 1951) to the concept of periodical activations of behaviors (Pezzulo and Calvi 2006; Stoytchev and Arkin 2001; 2004) introducing the Adaptive Innate Releasing Mechanisms (AIRMs). An AIRM is a *releasing mechanism* endowed with an internal *adaptive clock*.

In Figure 1 the AIRM is represented through a Schema Theory representation (Arbib 1998). Each behavior is characterised by a schema composed of a Perceptual Schema (PS), which elaborates sensor data, a Motor Schema (MS), producing the pattern of motor actions, and a control mechanism, based on a combination of a clock and a releaser. In particular, the releaser enables/disables the activation of the MS, according to the sensor data (t). For example, the presence of a predator releases the motor schema of an escape behavior. Instead, the adaptive clock is active with a base period and it enables/disables data flow $_r(t)$ from sensors to PS. Therefore, when the activation is disabled, sensor data are not processed (yielding to sensory readings reduction). Furthermore, the clock regulates its period, hence the frequency of data processing, using a feedback mechanism on the sensor data (t).

We assume a discrete time model - with the machine cycle as the time unit - where each behavior is endowed with a clock regulating its own activations. This regulation mechanism, that we call *monitoring strategy*, is characterized by:

- An initial period p_b called base period, ranging in an interval $[p_{min}, p_{max}]$,
- An *updating function* $f(t) : \mathbb{R}^n \rightarrow \mathbb{R}$ that changes the clock period p, according to the parameters the behavior depends on (sensors used, internal state, special features of the environment, and the behavior goal).
- A trigger function (t, p_{t-1}), which enables/disables the data flow $_r(t)$ from sensors to PS, at each p time unit. More formally:

$$(t, p_{t-1}) = \begin{cases} 1, & \text{if } t \bmod p_{t-1} = 0 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

- Finally, a support function $(f(t)) : \mathbb{R} \rightarrow \mathbb{N}$ maps the values generated by the updating function $f(t)$ in a range of allowed values for the period $[p_{min}, p_{max}]$. More precisely:

$$(x) = \begin{cases} p_{max}, & \text{if } x \geq p_{max} \\ \lfloor x \rfloor, & \text{if } p_{min} < x < p_{max} \\ p_{min}, & \text{if } x \leq p_{min} \end{cases} \quad (2)$$

Now, starting from the clock period at time 0,

$$p_0 = p_b \qquad \text{with } t = 0 \text{ and } p_b \in [p_{min}, p_{max}]$$

The clock period at time t is regulated as follows:

$$p_t = (t, p_{t-1}) * (f(t)) + (1 - (t, p_{t-1})) * p_{t-1} \quad (3)$$

That is, if the behavior is disabled, the value of the period calculated at time t remains unchanged at the last computed value p_{t-1}. Instead, when the value of trigger function is equal to 1, the behavior is activated and, subsequently, its activation period changes according to the $(f(t))$ function.

Attentive Monitoring and Control. The *monitoring strategy*, i.e. the process of changing the clock sampling rate, can be associated with the increase or decrease of attention towards a particular behavior. Namely, the more salient is the behavior, the higher is the clock frequency and the resolution at which a behavior is monitored and regulated. Notice that, the frequencies of the adaptive clocks provide also a divided attention mechanism: the monitoring activity is distributed over the concurrent behaviors depending on the frequencies of their associated clocks.

Following this approach, we can obtain different attentional mechanisms associated with each behavior once we define the associated *monitoring strategy*. Therefore an attentive behavior will result from the combination of the the initial period p_b, the permitted values range $[p_{min}, p_{max}]$ and the updating policy $f(t)$. In order to obtain a good

monitoring strategies, it is necessary to balance the cost of monitoring a behavior against the risk of acquiring inaccurate/degraded information about the environment.

These attentive monitoring strategies are introduced to provide the following main benefits:

- the periodical activation can reduce the number of activations of the perceptive system causing a relative decrease in the computational burden, and improving performance of the entire system;
- the use of adaptive activation mechanisms allows us to obtain a behavior that adapts itself to the specific environmental conditions (e.g. the robot reads sensors more often if there is a dangerous situation and less often in cases of a safe operational situation).

Example. Consider the example of a person who is crossing a street. Depending on the traffic intensity, this person has to pay more or less attention while crossing the street, turning his head left and right. Here, the monitoring frequency should be regulated according to the speed of the passing cars. The pedestrian has to react according not only to the environmental change (a car passing on the street) but also to the speed at which this happens (fast or slow cars). Intuitively, we can associate the speed of the pedestrian and his monitoring activity to the speed of the passing car. Following this approach, in (Burattini and Rossi 2008), we provided an example of a robot whose task was to cross a street avoiding moving obstacles. In this case, the updating policy has a frequency that is directly proportional to the speed of the moving obstacles: the higher the speed, the smaller the sampling period.

Case Studies Overview

In this section, we present and discuss our framework deployed in different scenarios and setting, both in simulation and in the real world, from simple scenarios to more complex settings. Our aim is to discuss our approach considering its effectiveness efficiency, adaptability (in different scenarios), and scalability (considering increasingly complex behaviors and tasks).

For the simulated experiments we used the *Stage* tool of the *Player* project (Gerkey, Vaughan, and Howard 2003), while for the real one we used he *PIONEER* 3*DX* robotic platform *Active Media Robotics*, endowed with a blobfinder camera, and sonars.

Conflicting tasks

In previous work (Burattini and Rossi 2010; Burattini et al. 2010), we investigated the application of the AIRM attentional mechanisms in simple cases of conflicting behaviors. In the following we provide an overview of two scenarios presented in (Burattini and Rossi 2010) and (Burattini et al. 2010) respectively.

Emergent Action Selection in Conflicting Tasks. In (Burattini and Rossi 2010), we describe a simple case study involving two conflicting attentive behaviors: `ESCAPE`, representing predator avoidance, `FIND_FOOD`, representing the search for food. `FIND_FOOD` has an updating policy that depends on the risk of starvation, and it is regulated by a linear time-dependent function representing hunger: the higher the hunger, the higher the attention towards the food. `ESCAPE` changes its clock period following the Weber-Fechner law of perception which is used to represent fear: the higher the fear, the higher the attention towards the predators. When `FIND_FOOD` is enabled and the robot perceives a green object (representing food), it activates a movement towards the food. When the `ESCAPE` is enabled and the robot perceives a red object (representing a possible threat), it activates a movement opposite to the threat and a velocity inversely proportional to the clock period.

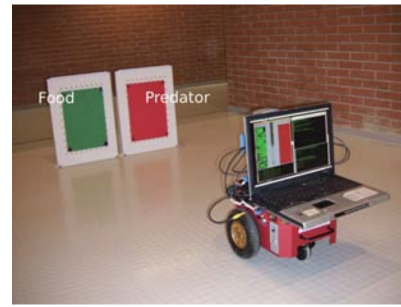


Figure 2: Conflicting tasks: food and predator.

When the robot encounters a red object close to a green object (see Figure 2) we have a conflicting behavior. In this case, the attentional mechanisms implemented with the adaptive clocks allow to balance the trade off between the risk of predation and the risk of starvation. This can be obtained avoiding the introduction of explicit action selection mechanisms. Indeed, if the threat stand still, as soon as the risk of starvation increases more than the value of the `ESCAPE` clock, the robot starts moving towards the food, escaping in the case an abrupt movement of the threat. The combination of these two behaviors, elicited by the risk of starvation and the risk of predation, is an oscillating movement that will lead, eventually, to reach the position of the food.

Parallel Execution of Conflicting Task. Inspired by studies (Patten et al. 2004; Harbluk, Noy, and Eizenmann 2002) on cognitive distraction while driving (i.e., talking over a mobile phone), (Burattini et al. 2010) considers a case study including two behaviors that, although conflicting, can be simultaneously carried on. (Harbluk, Noy, and Eizenmann 2002) shows that drivers, under a high cognitive load, execute less saccadic movements consistently with an increase of fixation time and a smaller exploration of the visual field. These results suggest that parallel tasks can be accomplished, but the resources allocated to each task are dynamically distributed according to environmental conditions and to cognitive and physical capabilities.

To investigate our framework in an analogous setting, in (Burattini et al. 2010), we designed the case study of a mo-

Figure 3: The hallway domain.

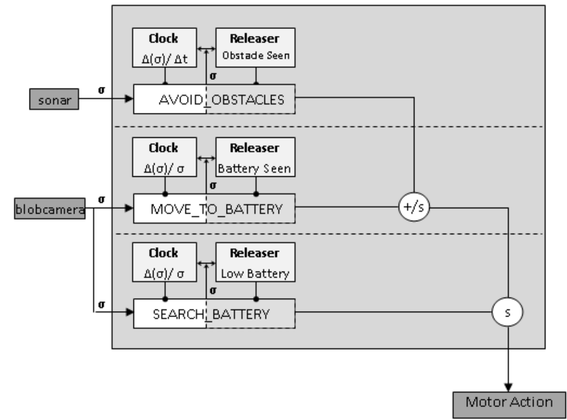


Figure 4: Control architecture.

bile robot that is to run across a hallway in the shortest time possible, while counting green blobs distributed on walls and arranged into clusters (Burattini et al. 2010) (see Figure 3). The two tasks of running and counting conflict on the speed of the robot. Indeed, the first task requires a high speed, while the second require a slow speed to effectively count all the blobs.

In order to accomplish the two tasks we implemented a robotic system endowed with three behaviors: `RUN`, `SEARCH`, and `SCAN`. `SEARCH` looks for green blobs on the left and right wall. This behavior works with a maximal frequency until at least one green blob is detected, then the period is increased proportionally to the amount of green blobs. `SCAN` counts the blobs once a salient area is identified and the clock period is proportional to the `SEARCH` activation frequency. `RUN` sets the speed of the robot that is in inversely proportional to the period. The clock period of the `RUN` is directly proportional to the period of `SEARCH`.

The observed system behavior is the following. The robot starts running with a medium speed, looking for green objects on the walls of the corridor. When the system detects a cluster of blobs, the period of `SCAN` decreases, allowing the robot to slow down its speed and to count the objects it detects. Similarly, if no green objects are detected, the period of the `RUN` become smaller, allowing a more accurate exploration (moving several times the camera looking for objects), and increasing the system speed to reach the end of the corridor as soon as possible.

In (Burattini et al. 2010), we compared the system performances with respect to an analogous system with non adaptive clocks (i.e. activation at each machine cycle). The experiments show that the proposed architecture performs better compared to the non attentional setting in terms of: number of detected blobs (effectiveness); tradeoff between time and counted blobs (cost/benefit); error of detection (precision); less activations of the perceptual schema (efficiency). The attentive system is effective in counting blobs because it can coordinate and modulate speed and pan-tilt control, focusing the visual exploration on the region of interest. The overall attentional coordination increases the time needed to accomplish the task, but this additional time is spent in the counting phase, effectively trading off between time and precision. Basically, the system can modulate the activation frequencies on the basis of the available resources and external conditions. Indeed, using the adaptive clocks, the number of behaviors activations substantially decreases compared to the case where each behavior is enabled at each machine cycle, and this results in a substantial gain in performances.

Foraging domain

In this scenario, we consider a robot whose aim is to explore a dynamic and unknown environment avoiding obstacles and seeking sources of energy to recharge its batteries in a fixed amount of time.

In this scenario, we evaluated the performances of our attentive system with respect to the performances of analogous behavior-based systems not equipped with adaptive clocks. In particular, to better assess the gain due to the attentional mechanisms, we compared the system with respect to two different versions of the control system:

(a) a *cautious* version of the system, that we call *without clocks* (STD): where each behavior can be activated at each machine cycle, depending on the releasing function (as in a standard non adaptive behavior-based architecture);

(b) a *brave* version of the system with *periodic clocks*: where behaviors are associated with periodic, but fixed, activation periods. In this case, each behavior has its own clock without attentional adaptivity.

In this setting, we want to prove the scalability of the attentive mechanism with respect to the system complexity (system with more behaviors) and different environmental conditions (obstacles configurations). Our aim here is to demonstrate the ability of our attentive monitoring strategies to regulate the resources distribution among the different behaviors.

Behavior-based control. The robot behavior is obtained as the combination of the following primitive behaviors (see Figure 4): AVOID, SEARCH_BATTERY, MOVE_TO_BATTERY.

The AVOID behavior is responsible for obstacle avoidance. This behavior is safety critical and needs an updating policy for its adaptive clock which is able to timely react to dangerous situations. In this case, the AVOID clock period changes according to the first derivative of the input percept. More formally, the AVOID clock period is updated according to (3), with the following updating function:

$$f(t) = \left(\frac{{}_{avoid} * p_{t-1}}{\Delta \quad (t) + k_{avoid}} \right)$$

where $\Delta \quad (t)$ is equal to $\quad (t) - \quad (t - p_{t-1})$ that is the difference between the actual data perceived by the sensor $\quad (t)$ and date received at the previous sampling time $\quad (t - p_{t-1})$. In this way, the AVOID activations frequency adapts itself not only to the environmental changes, but also to the speed at which these changes take place. ${}_{avoid}$ and k_{avoid} are two attenuation parameters. These two parameters are context dependent and can be tuned by a suitable learning algorithm.

The AVOID behavior is responsible not only for the robot orientation, but also for its speed variations. In particular, speed is related to the period according to the following relation:

$$speed_t = \frac{max_speed \times p_t}{p_{max}}$$

where $speed_t$ is the current speed, max_speed is the maximum value allowed for the robot speed. The range of values for the speed is $[0, 0.3] m/s$. In this way, if the period is relaxed, the robot moves at a maximum speed, otherwise, it slows proportionally to the decrease of the period. This allows the agent to avoid obstacles in a smooth way (see the next paragraphs for details).

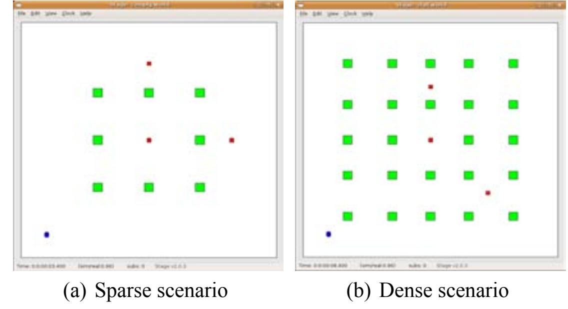
The SEARCH_BATTERY behavior provides a random search of sources of energy in the environment. The frequency of this behavior activation is related to the level of charge of the robot's battery. The lower the battery, the greater the activity of the search behavior. Since we assume that the energy need is represented by a function $e(t)$ that grows with time, the updating function can be defined as follows:

$$f(t) = \frac{k_{search}}{e(t) + h_{search}}$$

where k_{search} and h_{search} are two context dependent parameter to be suitably tuned by a learning algorithm. The output of this behavior is a random pattern of orientations for the motor action.

The MOVE_TO_BATTERY behavior guides the agent towards the battery when this has been identified. So the releaser is activated by battery detection using the blob camera. Analogously, to the previous updating function, the period of this behavior activation also depends on the level of the battery charge and can be defined as:

$$f(t) = \frac{k_{move}}{e(t) + h_{move}}$$



(a) Sparse scenario (b) Dense scenario

Figure 5: Map of the environment in two scenarios. Energy sources and obstacles are, respectively, the small red boxes and the big green boxes. The robot is the blue rounded square at the left bottom.

Namely, the adaptive clock period is regulated by a time-dependent function that represents the agent need of energy.

If the releaser is on and the agent can perceive the battery, the output will be a movement towards it, otherwise the agent will relies on the SEARCH_BATTERY behavior. The trajectory towards the battery is calculated with respect to the centroid of the red blob, which identifies the battery charge in the scene.

Testing Scenarios. The robotic system works in a foraging domain characterized by an area of 20m x 20m (400 m^2). In this environment we considered two possible configurations: (1) a sparse scenario with few obstacles (Figure -(a)) and (2) a dense scenario with many obstacles (Figure -(b)). The size of the robot with respect to the environment is $0.2\ m \times 0.1\ m$ ($0.2\ m^2$). Obstacles are represented by a green square ($0.7 m \times 0.7\ m$), while the energy source by red square in size $0.3\ m \times 0.3\ m$. We performed the experiments in simulation using the *Stage* tool.

In these scenarios, we considered the system performances by incrementally adding behaviors and tasks. From only one behavior (AVOID) to a combination of three behaviors (AVOID, SEARCH_BATTERY, and MOVE_TO_BATTERY). For each setting, we collected the data of 10 runs.

Avoiding Obstacles. First of all, we considered a robot equipped with the AVOID behavior, whose task is to safety navigate into the environment with obstacles, for a fixed interval of time (5 minutes). This test has been performed in both the sparse and dense scenarios.

In Table 1, the results of the attentive system are compared with respect to the caution version (case (a)), *without clocks*, and brave version (case (b)), with *periodical clocks*.

The collected parameters are: the number of activations of the avoid behavior, the number of possible dangerous situation (minimum distance from the obstacle detected by sonar less then $0.3\,m$), and the average speed of each run.

In Table 1, we see that both in the case of sparse and dense obstacles, the number of the different behaviors activations is radically reduced in the case of the attentive system. Less behavior activations determines a reduction in the computa-

	AVOID		dangers		speed (m/s)	
	avg	st.dev	avg	st.dev	avg	st.dev
SPARSE adaptive	403	18	6,8	6,7	0,2874	0,0045
SPARSE periodic	621	14	24,3	27,4	0,2886	0,0053
SPARSE without clock	1203	4	0	0	0,2078	0,0312
DENSE adaptive	476	30	3,6	5,5	0,2696	0,0075
DENSE periodic	625	3	45,3	49,8	0,2748	0,0136
DENSE without clock	1279	25	0	0	0,1704	0,0118

Table 1: Attentive, Periodic and STD architectures endowed with the AVOID behavior and compared in the sparsity and density scenario.

tional time spent for sensory data acquisition and processing.

Furthermore, these results show that by improving the complexity of the environment we do not lose the benefits of the attentive setting in term of low activations and high average speed.

The results obtained with periodic clocks represent a medium case. Indeed, the periodic setting reduces the behavior activations with respect to the setting without clocks, however, without adaptability we cannot ensure the robot safety (note the increment of possible dangerous situations in the case of periodic clocks).

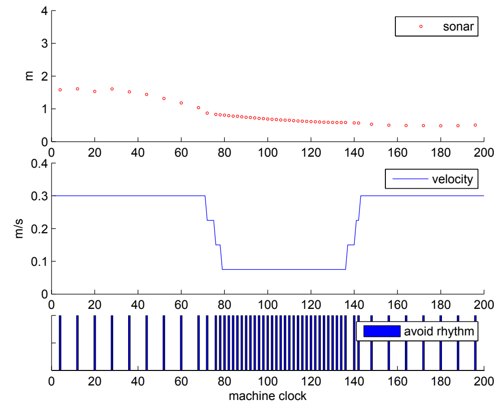
Avoiding Obstacles and Reaching a Source of Energy. In a second set of tests, we enhanced the functionality of the control system by adding the SEARCH_BATTERY and MOVE_TO_BATTERY behaviors. Here, the robot task is to safety navigate the environment trying to reach the source of energy according to its needs in a fixed amount of time. The amount of time chosen for the experiments is 3 minutes. As before, we compared the performances of the three architectures both in the sparse and in the dense environment.

In Table 2, in addition to the data presented in the previous test, we show also the average number of sources of energy reached.

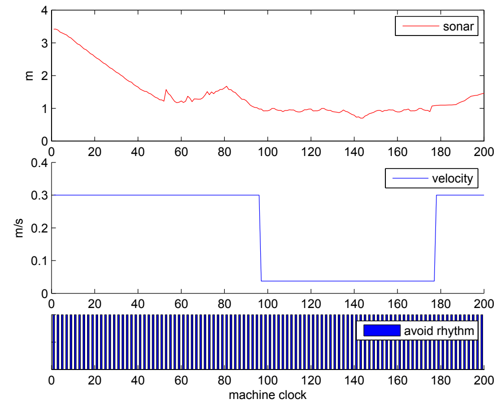
Differently from previous case, the number of MOVE_TO_BATTERY activations is minimal with a periodic system, however, here we have also a decrease in the average number of batteries reached. This happens because the MOVE_TO_BATTERY behavior is responsible of directing the robot toward the source of energy, hence, the smaller the number of the activations the lower the chance of finding battery and the precision of the robot maneuvers during the battery approach. Moreover, in the periodic setting, the number of possible dangers grows dramatically with respect to the attentive one, where we find more sources of energy and less dangerous situations.

If we compare the attentive architecture with respect to the one without clocks, we see less activations (Table 2), more energy found, and less crashes despite the average speed the robot remains high. This means that the attentive robot can reach its goals earlier with less effort.

Moreover, the attentive behavior appears smoother and more natural then the one of the non adaptive versions; this also affects safety. For example, in the attentive case the agent can avoid obstacles in a smooth way (Figure 6-(a)), because the AVOID behavior, responsible for the speed vari-



(a) Attentive avoidance



(b) Non-adaptive avoidance

Figure 6: Comparing avoidance behaviors.

	AVOID		MOVE_TO_B		SEARCH_B		dangers		speed (m/s)		energy	
(S=Sparse/D=Dense)	avg	st.dev	avg	st.dev	avg	st.dev	avg	st.dev	avg	st.dev	avg	st.dev
S adaptive	310,7	10,4	132,6	67,0	45,2	5,1	39,4	25,8	0,282	0,003	1,1	0,6
S periodic	560,2	62,4	17,5	37,0	113,6	22,8	136,9	77,1	0,167	0,003	0,2	0,4
S without clock	968,8	69,8	417,8	197,0	302	101,7	87,7	40,9	0,175	0,012	1,9	0,7
D adaptive	330,3	6,8	250,5	114,4	32,6	7,0	15,2	9,9	0,238	0,018	0,8	0,4
D periodic	605,2	175,1	28,7	59,4	93,9	29,1	72	111,4	0,162	0,007	0,4	0,8
D without clock	1054	43,9	106,5	50,5	408,2	124,9	49,5	15,2	0,169	0,021	1,1	0,6

Table 2: Attentive, Periodic and STD architectures endowed with three behaviors and compared in the sparsity and in the dense scenarios.

ations, can modify the robot speed proportionally to the relevance of the situation. Instead, in the non-adaptive case, the speed is very high if there is no danger, very low otherwise; this produces drastic speed variations (Figure 6-(b)) that can determine unsafe behaviors.

Related Works

Attention-based control is an emerging issue, in particular for vision-guided mobile robots. Several approaches in literature address the problem of feature extraction to support task execution (Minato and Asada 2001), localization, mapping, and navigation (Mitsunaga and Asada 2002; Frintrop, Jensfelt, and Christensen 2006; Carbone et al. 2008). For instance, in (Minato and Asada 2001) an attentive behavior is learned by pairing actions and image features.

Mechanisms for executive and divided attention in robot execution monitoring are less explored. In (Garforth, McHale, and Meehan 2006), the authors investigate executive attention in mobile robotics tasks proposing the deployment of a supervisory attentional system inspired by (Norman and Shallice 1986). Concurrent tasks interacting with the attentive processes are considered in (Wasson, Kortenkamp, and Huber 1999) where we find a robot architecture integrating active vision and tasks execution. However, here divided attention is not considered while attentive and goal-directed behaviors are integrated and coordinated using a perceptual memory.

Closely related to our system, in (Stoytchev and Arkin 2001) Stoytchev and Arkin propose an hybrid architecture combining deliberative planning, reactive control, and motivational drives. In this context, the internal state is represented by motivational variables affecting action and perception. Analogously to our framework, periodic activations of behaviors as cicardian rhythms and time-dependent motivational processes are deployed, however, here internal clocks are not directly used for attention selection and behavior modulation.

Other authors dealt with flexible/adaptive behavior realized through timed activations. For example, (Pezzulo and Calvi 2006) presented a parallel architecture focused on the concept of activity level of each schema which determines the priority of its thread of execution. A more active perceptual schema can process the visual input more quickly and a more active motor schema can send more commands to the motor controller. However, while in our approach such effects are obtained through periodic activation of behaviors, in (Pezzulo and Calvi 2006) the variables are elaborated through a fuzzy based command fusion mechanism.

Our attentive sampling can be also related to flexible scheduling for periodic tasks in real-time systems (Buttazzo et al. 2002; Beccari, Caselli, and Zanichelli 2005). Here, analogously to our system, period modulation is exploited to degrade computation and keep balanced the system load. For example in (Buttazzo et al. 2002), the authors propose an elastic model to decide how to change the sampling period associated with a task. Similar techniques can be incorporated in our framework, however, in our case sampling rate depends not only on the computational load, but also on salience due to environmental changes, motivations, and goals.

Conclusions and Future Work

In this paper, we illustrate an attention-based control architecture for a robotic system capable of adapting its emergent behavior to the surrounding environment and to its internal state. While attention-based robot control has been already considered in literature, mainly for vision-based robots, mechanisms for executive and divided attention in robot execution monitoring are less explored. In the context of a behavior-based executive system, we introduced simple attentional mechanisms which are based on the periodic releasing mechanisms of activations introduced by (Burattini and Rossi 2007; 2008).

In the proposed attentional system, each behavior is equipped with an adaptive clock and the process of changing the frequency of sensory readings is interpreted as an increase or decrease of attention towards relevant behaviors and particular aspects of the external environment.

To validate our approach, we experimented the control architecture in different case studies. In particular, we tested the scalability and the adaptivity of the approach with respect to different and heterogeneous environments and tasks. Furthermore, we evaluated the performances of the attentional system with respect to the performances of other behavior-based systems not provided with attentive and adaptive mechanisms. The collected results show that attentional mechanisms permit a smooth and natural emergent behavior in all the considered scenarios trading off between adaptivity and performances. We are currently investigating suitable learning mechanisms to set the parameters associated with monitoring strategies and attentional mechanisms to combine deliberative and reactive processes.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no.216239.

References

Arbib, M. A. 1998. Schema theory. In *The handbook of brain theory and neural networks*. Cambridge, MA, USA: MIT Press. 830–834.

Arkin, R. C.; Ali, K.; Weitzenfeld, A.; and Cervantes-Pérez, F. 2000. Behavioral models of the praying mantis as a basis for robotic behavior. *Robotics and Autonomous Systems* 32(1):39–60.

Beccari, G.; Caselli, S.; and Zanichelli, F. 2005. A technique for adaptive scheduling of soft real-time tasks. *Real-Time Syst.* 30(3):187–215.

Burattini, E., and Rossi, S. 2007. A robotic architecture with innate releasing mechanism. In *2nd International Symposium on Brain, Vision and Artificial Intelligence*, volume 4729 of *Lecture Notes in Computer Science*, 576–585. Springer.

Burattini, E., and Rossi, S. 2008. Periodic adaptive activation of behaviors in robotic system. *Int. J. Pattern Recognition and Artificial Intelligence - Special Issue on Brain, Vision and Artificial Intelligence* 22(5):987–999.

Burattini, E., and Rossi, S. 2010. Periodic activations of behaviours and emotional adaptation in behaviour-based robotics. *Connection Science* 22(2):(in press).

Burattini, E.; Rossi, S.; Finzi, A.; and Staffa, M. 2010. Attentional modulation of mutually dependent behaviors. In *Proceedings of the 11th Interntional Conference on Simulation of Adaptive Behavior*, (in press).

Buttazzo, G. C.; Lipari, G.; Caccamo, M.; and Abeni, L. 2002. Elastic scheduling for flexible workload management. *IEEE Trans. Computers* 51(3):289–302.

Carbone, A.; Finzi, A.; Orlandini, A.; and Pirri, F. 2008. Model-based control architecture for attentive robots in rescue scenarios. *Auton. Robots* 24(1):87–120.

Cooper, R., and Shallice, T. 2000. Contention scheduling and the control of routine activities. *Cognitive Neuropsychology* 17:297–338.

Frintrop, S.; Jensfelt, P.; and Christensen, H. I. 2006. Attentional landmark selection for visual slam. In *Proc. of IROS 2006*.

Garforth, J.; McHale, S. L.; and Meehan, A. 2006. Executive attention, task selection and attention-based learning in a neurally controlled simulated robot. *Neurocomputing* 69(16-18):1923–1945.

Gerkey, B.; Vaughan, R.; and Howard, A. 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proc. ICAR 2003*, 317–323.

Harbluk, J. L.; Noy, Y. I.; and Eizenmann, M. 2002. Impact of cognitive distraction on driver visual behavior and vehicle control. Technical report, 81st annual meeting of the Transportation Research Board, Washington, DC.

Kahneman, D. 1973. *Attention and Effort*. Englewood Cliffs, NJ: Prentice-Hall.

Lorenz, K. 1991. *King solomon's ring*. Penguin.

Minato, T., and Asada, M. 2001. Image feature generation by visio-motor map learning towards selective attention. In *Proc. of IROS-2001*, 1422–1427.

Mitsunaga, N., and Asada, M. 2002. Visual attention control for a legged mobile robot based on information criterion. In *Proc. of IROS-2002*, 244–249.

Norman, D., and Shallice, T. 1986. Attention in action: willed and automatic control of behaviour. *Consciousness and Self-regulation: advances in research and theory* 4:1–18.

Patten, C. J. D.; Kircher, A.; Ostlund, J.; and Nilsson, L. 2004. Using mobile telephones: cognitive workload and attention resource allocation. *Accid Anal Prev* 36(3):341–350.

Pezzulo, G., and Calvi, G. 2006. A schema based model of the praying mantis. In *SAB – 9th International Conference on Simulation of Adaptive Behavior*, volume 4095 of *Lecture Notes in Computer Science*, 211–223. Springer.

Stoytchev, A., and Arkin, R. C. 2001. Combining deliberation, reactivity, and motivation in the context of a behavior-based robot architecture. In *Proc. 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, 290–295.

Stoytchev, A., and Arkin, R. C. 2004. Incorporating motivation in a hybrid robot architecture. *JACIII* 8(3):269–274.

Tinbergen, N. 1951. The study of instinct. Oxford University press.

Wasson, G.; Kortenkamp, D.; and Huber, E. 1999. Integrating active perception with an autonomous robot architecture. *Robotics and Autonomous Systems* 26:325–331.

Combining Planning and Motion Planning

Jaesik Choi and Eyal Amir
Department of Computer Science
University of Illinois at Urbana Champaign
Urbana, IL, 61801 USA
{jaesi,eyal}@illinois.edu

Abstract

Robotic manipulation is important for real, physical world applications. General Purpose manipulation with a robot (eg. delivering dishes, opening doors with a key, etc.) is demanding. It is hard because (1) objects are constrained in position and orientation, (2) many non-spatial constraints interact (or interfere) with each other, and (3) robots may have multi-degree of freedoms (DOF). In this paper we solve the problem of general purpose robotic manipulation using a novel combination of planning and motion planning. Our approach integrates motions of a robot with other (non-physical or external-to-robot) actions to achieve a goal while manipulating objects. It differs from previous, hierarchical approaches in that (a) it considers kinematic constraints in configuration space (C-space) together with constraints over object manipulations; (b) it automatically generates high-level (logical) actions from a C-space based motion planning algorithm; and (c) it decomposes a planning problem into small segments, thus reducing the complexity of planning.

1. Introduction

Algorithms for general purpose manipulations of daily-life objects are still demanding (e.g. keys of doors, dishes in a dish washer and buttons in elevators). However, the complexity of such planning algorithm is exponentially proportional to the dimension of the space (the degree-of-freedom (DOF) of the robot and the number of objects) (Canny 1987). It was shown that planning with movable objects is P-SPACE hard (Chen and Hwang 1991; Dacre-Wright, Laumond, and Alami 1992; Stilman and Kuffner 2005). Nonetheless, previous works examined such planning in depth (Likhachev, Gordon, and Thrun 2003; Kuffner and LaValle 2000; Kavraki et al. 1996; Brock and Khatib 2000; Alami et al. 1998; Stilman and Kuffner 2005) because of the importance of manipulating objects. The theoretical analysis gave rise to some practical applications (Alami et al. 1998; Cortés 2003; Stilman and Kuffner 2005; Conner et al. 2007), but general purpose manipulation remains out of reach for real-world-scale applications.

Motion planning algorithms have difficulty to represent non-kinematic constraints despite of its strength in planning with kinematic constraints. Suppose that we want to let a robot push a button to turn a light on. CSpace[1] can represent such constraints. However, the CSpace representation could be (1) redundant and (2) computationally inefficient because CSpace is not appropriate for compact representations. It could be redundant, because it always considers the configurations of all objects beside our interests (i.e. a button and a light). Moreover, mapping such constraints into CSpace would be computationally inefficient, because mapping a constraint among n objects could take $O(2^n)$ evaluations in worst case. Thus, most of motion planning algorithms assume that such mappings in CSpace are encoded.

AI planning algorithms and description languages (e.g. PDDL (McDermott 1998)) have difficulty to execute real-world robots despite of its strength in planing with logical constraints. Suppose that we have a PDDL action for 'push the button' which makes a button pushed and a light turned on. However, the PDDL description could be (1) ambiguous and (2) incomplete (require details). Given a robot with m joints, it is ambiguous how to execute the robot to push the button, because such execution is not given in the description. Instead, it assumes that there is a predefined action which makes some conditions (e.g. a button pushed) satisfied whenever precondition is hold and the action is done.

Both methods solve this problem in different ways. Motion planning algorithms use abstractions to solve this problem. AI plannings use manual encodings. Although abstraction provides solutions in a reasonable amount of time in many applications, abstraction lose completeness. Thus, it has no computational benefit in worst cases. Although AI plannings have no need to search the huge CSpace, it requires manual encodings which are not only error-prone but also computationally inefficient.

We minimize manual encodings using the reachability of objects. That is, logical actions are extracted from a tree (planned by a motion planning algorithm), if the actions change the reachability of objects (i.e. a switch can be reachable by opening a door).

Our algorithm provides a path of a robot given following inputs: configurations of a robot and objects; constraints between objects; an initial state; and a goal condition.[2] We use logical expressions to represent both spatial constraints

[1] CSpace is the set of all possible configurations

[2] For each object, we provide a function which maps from a configuration to discrete states (labels) of objects, if discrete states

Dagstuhl Seminar Proceedings 10081
Cognitive Robotics

in C-space (e.g. collision) and constraints in state space (we define them formally in section 4). We automatically build a set of actions from a motion planner, while it was done by hands in previous works.

In detail, our algorithm unifies a general purpose (logical) planner and a motion planner in one algorithm. Our algorithm is composed of three subroutines: (1) extracting logical actions from a motion planner, (2) finding an abstract plan from the logical domain, and (3) decoding it into C-space. It extracts PDDL actions (McDermott 1998) from a tree constructed by a motion planner in C-space. Then, it combines extracted actions with a given KB_{object} (Knowledge Base) that has propositions, axioms (propositional formulas) and abstract PDDL actions. To find an abstract plan efficiently, we automatically partitioned the domain by a graph decomposition algorithm before planning. In the planning step, an abstract plan is found by a factored planning algorithms (Amir and Engelhardt 2003; Brafman and Domshlak 2006) which are designed for the decomposed domain. In decoding, a motion plan is found from the abstract plan.

We argue that the complexity of a planning problem is bounded by the treewidth of the encoded KB. One may think some analogy between the treewidth of KB in this paper and the number of mutually-interfering objects in the motion planning literature. However, the treewidth is more general expression because KB has more expressive power than the conventional C-space. In addition, this work proposes two improvements in terms of efficiency. One improvement is to use a factored planning algorithm for the decomposed domain. The other is to encode actions on behalf of workspace which is much smaller than C-space.

This approach is a unique decomposition-based path planning algorithm. We minimize manual encodings which are required to manipulate objects. Both (kinematic) constraints of the robot, and constraints of manipulating object are considered in our planning. It is efficient because its efficiency depends only on the workspace (2D or 3D), when appropriate conditions are met. Moreover, our method calculates actions of a robot once and can reuse them for other tasks.

Section 2presents related works. Section 3provides a motivational example. Section 4explains our encoding to build a KB. Sections 5and 6show our algorithm. Finally, section 7. provides experimental results followed by the conclusion in section 8.

2. Related Works

Here, we review the related works in two aspects: (1) using logical representation in robot planning; and (2) modifying the motion planning algorithm to achieve complex task (eg. manipulating objects). One may see the former way as top-down and the latter way as bottom-up.

(Alami et al. 1998) presents a well-integrated robot architecture which controls multiple robots. It uses logical representations in higher level planners and C-space based motion planners in lower-level planning. However, the combination of two planners is rather naive (manual).

are required for the provided constraints of objects (KB_{object}).

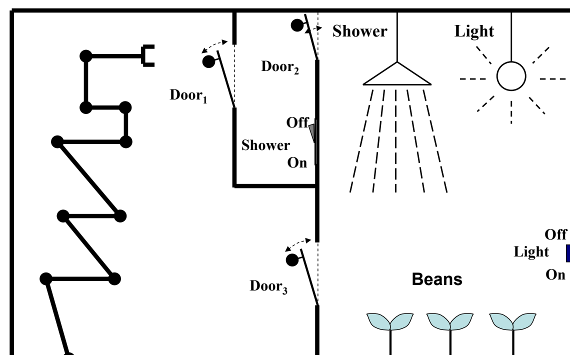


Figure 1: This figure shows an example of manipulating objects with a robotic arm. The goal is to take care of beans in a glasshouse. Beans require water and light everyday. The robot will provide water and light for beans. To accomplish this goal, the arm needs to manipulate objects such as doors and switches.

Recently, (Conner et al. 2007) provides an improved way to combine the *Linear Temporal Logic (LTL)* to control continuously moving cars in the simulated environment.[3] However, their model is a nondeterministic automata, while our model is deterministic. Due to the intractability of nondeterministic model, their representation is restricted to a subset of LTL to achieve a tractable (polynomial time) algorithm. Experiments are focused on controlling cars instead of manipulating objects.

Motion planning research has a long-term goal of building a motion planning algorithm that finds plans for complex tasks (eg. manipulating objects). (Stilman and Kuffner 2005) suggests such a planning algorithm based on a heuristic planner (Chen and Hwang 1991) which efficiently relocates obstacles to reach a goal location. Recently, it was extended to embed constraints over object into the *C-space* (Stilman 2007). In fact, the probabilistic roadmap method (Kuffner and LaValle 2000) of the algorithm is highly effective in manipulating objects in detail. However, we argue that our algorithm (factored planning) is more appropriate in terms of generality and efficiency than a search-based (with backtracks) heuristic planner.

Other works also present efforts in this direction to build a motion planning algorithm for complex tasks. (Plaku, Kavraki, and Vardi 2008) solves a motion planning problem focused on safety with logical constraints represented with LTL . (M. Pardowitz 2007) focuses on learning actions for manipulating objects based on the explanation based learning (Dejong and Mooney 1986). They use a classical hierarchical planner in planning. (J. Van den Berg 2007) provides an idea that extracts the propositional symbols from a motion planner. The symbols are used to check the satisfiability of the planning problems. (S. Hart 2007) uses a potential field method to achieve complex tasks with two arms. However, the main interests of these works are not planning algorithm, or are limited to the rather simpler tasks.

[3] Any *First Order Logic (FOL)* sentences can be reduced to *Linear Temporal Logic (LTL)*. Thus, LPL is a superset of FOL.

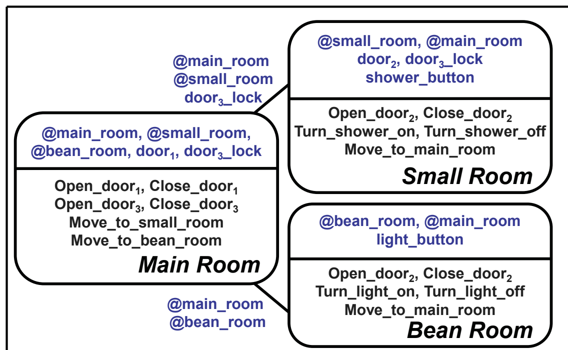


Figure 2: This is a possible tree decomposition for the toy problem of figure 1. The shared propositions appear on edges between subgroups. For example, a proposition ('$@door_3_lock$') is shared by two subgroups ('Main Room' and 'Small Room') because the proposition is used by actions of two subgroups (respectively '$Open(Close)_door_3$' and '$Turn_shower_on(off)$'). The KB is decomposed into small groups based on the geometric information (eg. the configurations of the room).

3. A Motivating Example

Figure 1 shows a planning problem. The goal is to provide water and light to beans. The robotic arm should be able to manipulate buttons in the spatial space to provide water and light. There are also non-spatial constraints. At any time either the $shower$ is off or $door_3$ is closed or both.

The planner requires both a general purpose (logical) planner and a motion planner. It requires general purpose planner because the arm needs to revisit some points of C-space several times in a possible solution. The way points may include '$Open_door_1$', '$Close_door_1$', and '$Turn_light_on$'. The state space can be different, whenever the robot revisits the same point in the C-space. It is certainly motion planning problem because the kinematic constraints of the arm should be considered. For example, the arm should not collide with obstacles, although the hand of the arm may contact objects.

Hierarchical planners have been classical solutions for these problems. A hierarchical planner takes in charge of high level planning. A motion planner takes in charge of low level planning. However, researchers (or engineers) need to define actions of the robot in addition to axioms among propositions for objects. Without the manual encodings, the hierarchical planner may need to play with the large number of propositions ($O(exp(DOF_{robot}))$=|*discretized C-Space*|) , when DOF_{robot} is the DOF of the robot. With such naive encoding, computational complexity of planning become ($O(exp(exp(DOFs)))$).

Moreover, naive hierarchical planners often have difficulty to find solutions for the following reason. Firstly, it requires interactions between subgoals. For example, the arm must go into the "Bean room" and turn the "light" on (subgoal) before it goes into the "small room" and turn the "shower" on (subgoal). This is essentially the *'Susman anomaly'* which means that the planner dose one thing (being in the Bean room) and then it has to retract it in order to

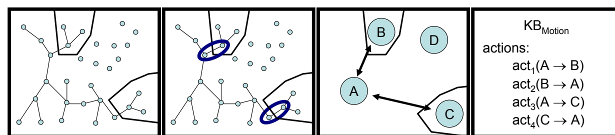


Figure 3: This figure illustrates a process to encode a motion plan into KB_M. The process is follows: (1) a motion plan (a tree) is built by a motion planning algorithm; (2) actions which changes the states of objects are found; (3) propositions are generated (and grouped) based on the found actions; and (4) a KB_M is created. Here, we assume that we have a function which provides discrete states of objects given the configuration of an object in finding actions (2). In this figure, the $door_1$ in figure 1 and 2 is closed in a set of states (A). The $door_1$ is moved little in B. However, the $door_1$ is not fully opened. Thus, configurations in the area D is not connected. The area C corresponds to the pushed light button on figure 1 and 2.

achieve other goal (turning the shower on). Thus, it may require several backtracking in planning. Secondly, there are two ways of (in principle) achieving "on(light)": (1) going through the small room; and (2) opening door to the Bean room from the Arm-base room. Unless manual encoding is given by an engineer, The latter way (going through the small room) is fine from the perspective of hierarchical planning. However, it will not work in practice because the arm is not long enough (kinematics). Formally, there is no *downward solution*.

Thus, this toy problem shows that (1) hierarchical planning does not work with a naive (simple) encoding, and (2) a complete encoding is too complex to engineer directly. We are interested in general principles that underlie a solution to this problem.

In motion planning literature, hybrid planners are used to address these problem (Alami, Siméon, and Laumond 1989; Alami, Laumond, and Siméon 1997; Alami et al. 1998; Conner et al. 2007; Plaku, Kavraki, and Vardi 2008). However, these are either hard to engineer due to manual encodings, or infeasible to conduct complex tasks due to the curse of dimensionality of expanded C-space. The size of C-space of a hybrid planner exponentially increases with additional movable objects and given propositions. Thus, solving a complex problem may require extensive search.

Here, we seamlessly combine the general purpose planning and the motion planning. Our planner finds all reachable locations and possible actions that change states of object, states of propositions, or the reachable set of objects.[4] Thus, high-level planner can start to plan based on the actions extracted by a motion planner.[5] The number of actions and states can be different according to constraints of the robot.

However, the number of actions and states can be still intractable. To solve this problem, we partition the domain into the smaller groups of actions and states. For

[4]Here, we assume that we know states of objects without uncertainty as in (Conner et al. 2007).

[5]Our planner may have more actions and states than the hand-encoded case.

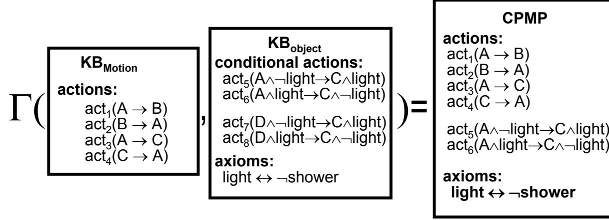


Figure 4: This shows an operation (or algorithm) to combine the extracted KB_M with pre-existing KB_O. KB_O is independently given in a general form to a robot. Thus, KB_O can be reusable for robots with different configurations space. Meanwhile, KB_{MP} is specific to a robot. Thus, some actions (e.g. act_7 and act_8) in KB_O are invalidated by the KB_M.

example, the domain can be partitioned as shown in figure 2. It is composed of three parts: (1) operating the shower switch; (2) operating the light switch; and (3) operating in between. The partition can be automatically done with approximate tight bound (Becker and Geiger 1996; Amir 2001).

A factored planner(Amir and Engelhardt 2003) efficiently finds a plan with the partitioned domain. The partitioned groups are connected as a tree shape. In the factored domain, our factored planner finds all the possible effects of the set of actions in each factored domain. Then, the planner passes the planned results into the parents of the domain in the tree. In the root node, all the valid actions and effects are gathered. The planner finds a plan for the task, if it exists.

Then, we use a local planner to find a concrete path in C-space at the final step. However, there is no manual (explicit) encoding (eg. 'turning the *switch A*') between two layers, except logical constraints and mapping functions provided as input.

4. Problem Formulation

Combining C-space and State Space

Here we suggest new problem formulation to combine C-space of an object-manipulating robot and KB (defined in the next paragraph) of objects and propositions. An object, located in a specific workspace, generates propositions into KB. Other axioms (propositional formulas) and actions (PDDL(McDermott 1998)) are given for the propositions. We will call this KB as **CPMP** (Combining Planning and Motion Planning).

Definition CPMP (Combining Planning and Motion Planning) is composed of propositions for states of a robot and objects, logical axioms over a robot and objects, and PDDL actions of a robot. It groups a set of points in C-space into a proposition (p_c) in the *CPMP*. Actions of a robot are translated into actions of the *CPMP*. A set of propositions and actions are constrained each other by logical axioms (propositional formulas).

A CPMP is composed of propositional symbols, logical axioms, and PDDL actions. The propositional symbols (P) represent states in binary values. The axioms ($Axiom$) are

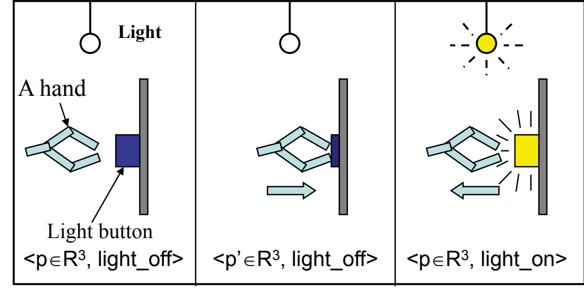


Figure 5: This example shows a situation in which one position in the workspace can correspond to two different states in the combined space (CPMP). Although the physical locations of the arm and button are the same in the workspace, the state (eg. light is on) is different. The situation can be represented when the C-space and state space in KB are combined (CPMP), and it is not possible to represent in the classical C-space alone.

propositional formulas. The actions ($Action$) represent the pair of preconditions and effects of a robot motion. It has a set of propositions that represents states of a robot and objects. *External states* are propositions in KB_M extracted from C-space. *Internal states* are propositions explicitly given in KB_O.

It also include a set of axioms. The axioms (logical formulas) represent relations among states of objects. When a state of an object (o^i) is o^i_1 (e.g. light), the state of another object (o^j) is constraints o^j_1 (e.g. ¬shower).[6] It is represented as follows.

$$o^i_1 \leftrightarrow o^j_1$$

In CPMP, a set of actions, KB_M, is generated from a tree (or network) in C-space built by a motion planning algorithm as shown in figure 3. In detail, two points (p_1 and p_2) in the network are connected by a line (an action of the robot). This can be simply encoded as follows.

$$\begin{aligned} Action: \quad & Move(p_1, p_2) \\ & \text{Precond}: p_1 \\ & \text{Effect}: p_2 \wedge \neg p_1 \end{aligned}$$

When the action changes the state of an object (o) from o_1 to o_2, the action can be encoded as follows.

$$\begin{aligned} Action: \quad & MoveObject(p_1, p_2, o_1, o_2) \\ & \text{Precond}: p_1 \wedge o_1 \\ & \text{Effect}: p_2 \wedge o_2 \wedge \neg p_1 \wedge \neg o_1 \end{aligned}$$

Figure 5 represents the expressive power of $CPMP$. It represents a situation which can not be described by a C-space but $CPMP$. The same physical locations are different states in $CPMP$ because the state of the light is changed.

A $CPMP$ has following properties.

[6]Such axioms are manually encoded. However, the encodings are independent of a specific robot. Thus, the encodings can be reusable to other types of robot. Moreover, there are algorithms (Amir and Russell 2003; Shahaf and Amir 2007) which can generate such axioms with a sensor-mounted robot.

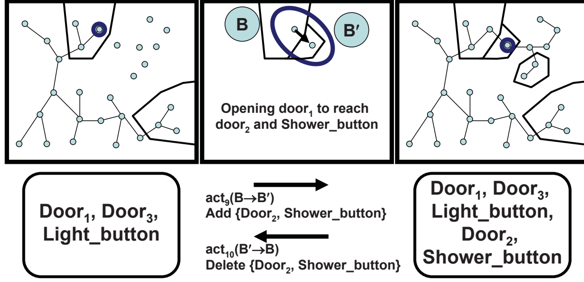


Figure 6: This figure represents an action which changes states of the object ($Door_1$) to change the reachable set of objects. Before doing the action ($act_9 B \rightarrow B'$), the set of reachable objects are { $Door_1$, $Door_3$, and $Light_button$ }. After the action, { $Door_2$, $Shower_button$ } are also included in the reachable set.

- A $CPMP$ has more expressive power than a C-space, if no two configurations in C-space can distinguish the two internal states.[7]
- It may reduce the number of propositions in $CPMP$, if spatial locations of end-effector are well-defined into disjoint sets. In each disjoint set, all spatial locations of end-effector have an identical internal state. Thus, any edge between the two disjoint sets changes some of the internal state.

Lemma 1. *The complexity of planning problem in the $CPMP$ is as hard as P-SPACE.*

Proof. Any motion planning problem (P-SPASE hard) with movable objects can be reduced to a planning problem in $CPMP$. Suppose that $CPMP$ includes only external propositions which are extracted from the motion planning algorithm. □

Encoding with Mapping Functions and Reachability

Here, we suggest an automatic encoding for moving objects while maintaining states given mapping functions[8] and reachability of objects. When a robot manipulates movable objects, it changes C-space of the robot. Hybrid systems(Alami, Siméon, and Laumond 1989; Alami, Laumond, and Siméon 1997; Alami et al. 1998) consider each C-space as a mode. Then, each manipulation connects two distinct modes. However, the size of the space is exponentially proportional to the number of objects and the number of joints. To address this issue, we group a set of modes based on the states of propositions and reachability of objects as shown in figure 3 and 6.

There are two cases to register an action (an edge between two points extracted from a motion planner) into CPMP. Firstly, we register an action into CPMP, if two points have different states in CPMP with a mapping function as shown in figure 3. We validate abstract PDDL actions which are realized by the action. Secondly, we also register an action into CPMP, if an edge changes a set of reachable objects as shown in figure 6.[9] Thus, we build a hypergraph whose nodes are sets of modes (C-spaces) which have the same states (in terms of mapping functions) and the same set of reachable object. Our algorithm extensively searches actions with a *resolution complete* motion planner until no new action is found in the hypergraph given a specific resolution.

Lemma 2. *The size of the discretized C-space for a robot manipulating n objects with given propositions in CPMP is bounded by $O(exp(|objects| + n + p))$, when $|objects|$ is the number of objects, n is the DOF (Degree of Freedom) of the robot, and p is the number of propositions.*

Lemma 3. *The number of possible actions (edges) in the discretized C-space for objects is only bounded by $O((|objects|) \cdot exp(|objects|))$, when the robot moves one object with an action.*

Proof. From a point in C-space of object $O(exp(|objects|))$, we can choose an object $O(|objects|)$ to change states. □

5. Finding a Solution in $CPMP$

To solve a task in $CPMP$, we provide a naive algorithm followed by two improvements: (1) it solves the problem in the (smaller) factored KBs; and (2) it reduces the number of propositions in $CPMP$ using workspace.

A Naive Solution

Given a CPMP, algorithm *NaiveSolution* finds a solution for a task. It may use a general purpose planner ($GeneralPlanner$) to find an abstract solution. Then, ($LocalMotionPlan$) encodes a path in C-space.

Input: r(a robot), KB_O(KB of objects), s_{start}(initial state), and s_{goal}(goal condition)
Output: $path_{concrete}$(Solution)
$KB_M \leftarrow$ FindActionFromMP(r)
$CPMP = \Gamma(KB_M, KB_O)$
$path_{abstract} \leftarrow$ GeneralPlanner($CPMP$, s_{start}, s_{goal})
$path_{concrete} \leftarrow$ LocalMotionPlan($path_{abstract}$)

Algorithm 1: *NaiveSolution* provides a path for a robot. It uses a general planner ($GeneralPlanner$) to find an abstract solution. Then, it is encoded into the path in the C-space by a motion plan ($LocalMotionPlan$).

Tree Decomposition of KB with Objects

Given a KB, finding a tree-decomposition of the minimum treewidth is a NP-hard problem. However, the complexity is only bounded by the treewidth of CPMP, if a tree-decomposition is found once by an efficient heuristic(Becker and Geiger 1996; Amir 2001).

[7] C-space normally takes into account configurations which only consider spartial locations of a robot or objects.

[8] A mapping function provides a state of a proposition (eg. object) given a configuration of objects and a robot.

[9] The reachable objects are added to preconditions and effects respectively.

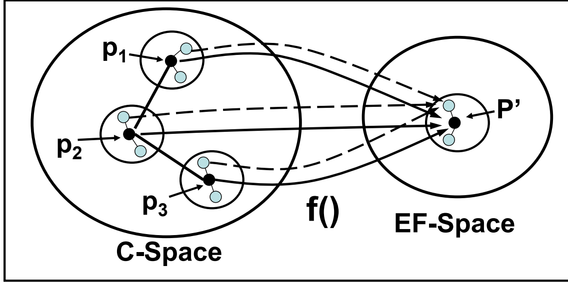


Figure 7: This figure shows a mapping function (f()) from a C-space to an EF-Space. p_1, p_2, and p_3 in C-space are mapped into p' in EF-Space. The connected lines ((p_1, p_2) and (p_2, p_3)) represent the first condition of Theorem 3. The circles represent the second condition.

Theorem 4. *The complexity of planning in CPMP is bounded by $O(exp(tw(CPMP)))$ if the tree-decomposition is given.*[10]

Proof. Proofs in (Brafman and Domshlak 2006; Amir 2001) can be easily modified to prove this theorem. □

From Exponential C-space to Polynomial EF-Space

In this section, we provide an improvement for a previous approach(Choi and Amir 2007) which uses workspace instead of exponential C-space. Although it is not always applicable, it efficiently finds a solution when applicable. Here, we want to transform C-space into a smaller space, EF-Space, using a mapping function $f()$. The function ($f()$) maps each point (p) in C-space into a point (p') in EF-Space with satisfying following conditions.

1. Suppose that P is a set of points whose image are p' in EF-Space ($f(p) = p'$). Any pair of two elements (p_1, p_2 $\in P$) is connected each other in C-space;
2. Suppose that two points (p and q) are mapped into two points (p' and q') in EF-Space. p and q are connected neighbor if and only if p' and q' are connected neighbor.

The connected neighbor means that they are directly connected in the space.

Theorem 5. *The complexity of motion planning in EF-Space is bounded by following*

$$O(\textit{EF-Space}) \cdot O(max_{ep \in \textit{EF-Space}}(ball(P_{ep}))).$$

P_{ep} is a set of points whose image is ep. (That is, $P_{ep} = \{p|f(p) = ep\}$) The $ball(P)$ is volume of the ball which includes P.

Proof. Given a motion planning problem (an initial configuration and goal one), a path in EF-Space can be found in O(EF-Space) with a graph search algorithm. Given the path in EF-Space, one needs to search the whole ball in worst case. □

One simple example of EF-Space is the workspace of end-effector. Suppose that the points in C-space are mapped into the points of end-effector in workspace. One can build an algorithm that finds all the neighboring points from the innermost joint (or wheel) to the outermost joint with dynamic programming. If points of the previous joint are connected to all the neighboring points, the neighboring points of the current joint are found by a movement of current joint (current step) or a movement of any previous joint (previous steps). The found connected points in workspace satisfy the second conditions, if the first condition holds in the workspace.

In worst case, the first condition is hard to satisfy. In the environment, the mapping function (f) should be bijective. Thus, the EF-Space is nothing but the C-space. However, the first condition holds in many environment where the distance between the obstacles (or object) and the robot is far enough. That is the theoretical reason why the planning problem in the spare environment is easy even in C-space.

Moreover, one can find another EF-Space considering topological shape of robot (Choi and Amir 2007). In the space, two points (p_1 and p_2) are mapped into the same point p'_1 if two configurations (p_1 and p_2) are homotopic, and they indicate the same end point. Otherwise, another point p'_2 is generated in the EF-Space. In 2D, two groups of configurations are divided by an island in right and left slides. Thus, the EF-Space is exponential to the number of island obstacles. However, the EF-Space itself is bounded by the workspace which is polynomial to the number of joints. Thus, it is much smaller than the C-space.

6. A Unified Motion Plan

We present our algorithms in this section. The main algorithm , *UnifiedMotionPlanner* (Algorithm 2), is composed of three parts: *FindActionFromMP* (Algorithm 3); *FactoredPlan* (Algorithm 4); and *LocalPlanner*. The goal of *UnifiedMotionPlanner* is to find a solution to achieve a goal situation.

```
Input: r(a robot), KB_O(KB of objects), s_start(initial state),
       s_goal(goal condition)
Output: path_concrete(Solution)
KB_M ← FindActionFromMP(r)
CPMP = Γ(KB_M, KB_O)
KB_Tree ← PartitionKBtoTree(CPMP)
path_abstract ← FactoredPlan( KB_Tree, s_start, s_goal )
path_concrete ← LocalPlan( path_abstract )
return path_concrete
```

Algorithm 2: *UnifiedMotionPlaner* finds all the reachable locations and actions in each location with FindActionFromMP. A motion planner is embedded in FindActionFromMP to extract abstracted actions in C-space. Then, PartitionKBtoTree partitions the $CPMP$ into a tree. FactoredPlan finds a solution given the pair of initial and goal condition in the partitioned tree domain. The LocalPlan finds a concrete path for the robot.

FindActoinFromMP

FindActionFromMP searches all the reachable locations and actions in C-space or EF-Space. In both cases, it has a dra-

[10] tw(KB) is the treewidth of KB.

Input: r(a robot)
Output: KB_M(extracted actions)
$MP_{Tree} \leftarrow$ a random tree in C-space built by a motion planner (e.g. Probabilistic Roadmap, Factored-Guided Motion Planning)
for *each edge* $(e_{ij}) \in MP_{Tree}$ **do**
  if $state(p_i) \neq state(p_j)$ **then**
    $KB_M \leftarrow KB_M \bigcup \{$
    $act_{ij}(state(p_i) \wedge p_i \rightarrow state(p_j) \wedge p_j \wedge \neg p_i) \}$
    $KB_M \leftarrow KB_M \bigcup \{$
    $act_{ji}(state(p_j) \wedge p_j \rightarrow state(p_i) \wedge p_i \wedge \neg p_j) \}$
return KB_M

Algorithm 3: . *FindActionFromMP* finds all abstract actions for a robot. A motion planner (eg. *FactorGuidedPlan* or *RoadmapMethod*) recursively finds all the reachable locations and actions. Then, the algorithm insert actions of each configuration (c_{ij}) of objects in the workspace. It assume that the object is in the configuration (c_{ij}). Thus, the condition (configuration of objects) is combined into the actions (act_{ij}). The union of all actions become the KB_M.

matically reduced space.

FactoredPlan

FactoredPlan finds a solution after factoring the domain (the space of end-effector in workspace) into small domains. It decomposes the domain into a tree in which each partitioned group becomes nodes, and shared axioms appear on a link between nodes. Then, it finds partial plans for a node and its children nodes with assuming that the parents nodes may change any shared states in between. After all, it finds a global solution in the root node.

Input: KB_{Tree} (partitioned KB as a tree), s_{start} (initial states), s_{goal} (goal condition)
Output: $path_{abstract}$ (An abstract plan)
depth $\leftarrow$ (predefined) number of interaction between domains.
for *each node*(KB_{part}) *in* KB_{Tree} *from leaves to a root* **do**
  $Act_{ab} \leftarrow$ PartPlan(KB_{part}, depth) .
  SendMessage(Act_{ab}, the parent node of KB_{part})
$path_{ab} \leftarrow$ a solution from s_{init} to s_{goal} in the root node of KB_{tree}
return $path_{ab}$

Algorithm 4: *FactoredPlanning* algorithm automatically partitions the domain to solve the planning problem (from s_{init} to s_{goal}). It iterates domains from leaves to the root node without backtracks. In each node, *PartPlan* finds all possible actions that change shared states in the parents node. *PartPlan* assumes that the parent node may change any states in the shared states in between. The planned actions in the subdomain become an abstract action in the parent node. They are passed by *SendMessage*.

7. An Experiment in Simulation

In this preliminary simulation, we build our algorithm for a task that pushes buttons to call numbers. There are 8 buttons in total. 4 buttons ($key1(P1)$, $key2(P2)$, $unlock(P3)$, and $lock(P4)$) are used to lock (and unlock) the buttons. Other 4 buttons ($\#A(P5)$, $\#B(P6)$, $\#C(P7)$ and $Call(P8)$) are used to make phone calls. Initially, the button is locked, the robot needs to push unlock buttons after pushing both key buttons ($P1$ and $P2$). Then, the robot can make a phone call with pushing the $Call$ button ($P8$) after selecting an appropriate number among $\#A(P5)$, $\#B(P6)$, and $\#C(P7)$. After a call, the buttons are automatically unlocked. We encode such constraints and action in KB_O.

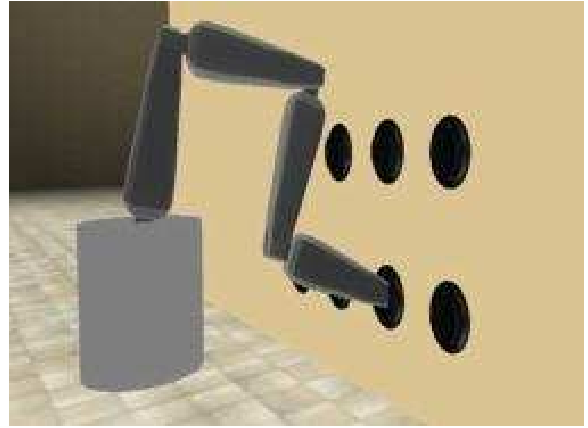


Figure 8: This is a capture of the motion of push button in the wall in experiments. The robot has 5 DOFs (rotational joints on the base and 4 revolute joints on the arm). We do experiment with increasing the number of joints from 2 to 9.

To build KB_M, we build a tree from a randomized algorithm with 80000 points in C-space. With a labeling function that returned the states of buttons, we found 33 edges in the tree[11]. They are encoded into 8 actions in KB_M for 8 buttons. Then, the combined KB ($CPMP$) is used to find a goal (calling all numbers ($\#A$, $\#B$, and $\#C$). The returned abstract actions are decoded into a path on the tree of motion plan. Figure 8 is a snapshot of the simulation.[12]

In this experiment, we focus on extracting actions from a motion planning algorithm, because factored planer itself is not a contribution of this paper. Theoretical and experimental benefits of *FactoredPlan* is shown in the previous papers (Amir and Engelhardt 2003; Brafman and Domshlak 2006). We run our simulation on a general purposed planner (Fourman 2007). Thus, the *NaiveSolution* algorithm is used in this simulation.

8. Conclusions and Future Research

We present an algorithm that combines the general purpose (logical) planner and a motion planner. Our planner is designed to manipulate objects with robot. To solve the problem, previous works used a hierarchical planner (high-level) and a motion planner (low-level). Most of them used manual encodings between two layers. That was one of technical

[11]We simplify the manipulations for attaching and detaching buttons

[12]The details of encoded actions and movies are available at http://reason.cs.uiuc.edu/jaesik/cpmp/supplementary/.

hardness of this problem.

Theoretically, the combination of such planner is hard for the following reasons: (1) hierarchical planner is hard and not feasible sometime; and (2) direct combination of C-space and state space gives an doubly exponential search problem. Moreover, we can miss the geometric motion planning information, if we translate everything to PDDL (McDermott 1998) without a motion planner.

We combine the C-space and state space in a KB, CPMP (Combining Planning and Motion Planning). Moreover, we provide the computational complexity of the problem. We also argue that the treewidth of CPMP determines the hardness of a manipulation task.

The suggested algorithm still has some limitations that need to be improved in future research. First, mapping function in Section 4needs manual encodings. Our algorithms assume that there is a mapping function which provides the value of shared propositions given a configuration of C-Space. Thus, an algorithm which can detect the change of shared propositions with sensors would be promising. Second, the exploration steps in *FindActionFromMP* may take long time due to the large cardinality of state space ($O(n + |objects| + p)$ as in lemma 2. Third, assumptions of EF-space would inappropriate for cluttered environments where $O(max_{ep \in \text{EF-Space}}(ball(P_{ep}))$ of theorem 5 are intractable.

The combining planning and motion planning is a generalized framework. However, there are many research problems to be solved in the future research. First, an algorithm which learns the mapping function between two spaces is necessary. Our algorithm assumes that there is a mapping function which provides the value of shared propositions given a configuration of C-Space. Thus, an algorithm which can detect the change of shared propositions with sensors would be promising. Second, the exploration step may take long time due to the large cardinality of state space. Thus, an adaptive exploration algorithm which builds a tree or a graph in CSpace based on the constraints of stats space would be useful.

9. Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 05-46663. We also thank UIUC/NCSA Adaptive Environmental Sensing and Information Systems (AESIS) initiative for funding part of the work.

References

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. 1998. An architecture for autonomy. *International Journal of Robotics Research* 17(4):315–337.

Alami, R.; Laumond, J.-P.; and Siméon, T. 1997. Two manipulation planning algorithms. In Laumond, J.-P., and Overmars, M., eds., *Algorithms for Robotic Motion and Manipulation*. Wellesley, MA: A.K. Peters.

Alami, R.; Siméon, T.; and Laumond, J.-P. 1989. A geometrical approach to planning manipulation tasks. In *Proceedings International Symposium on Robotics Research*, 113–119.

Amir, E., and Engelhardt, B. 2003. Factored planning. In *IJCAI*, 929–935.

Amir, E., and Russell, S. J. 2003. Logical filtering. In *IJCAI*, 75–82.

Amir, E. 2001. Efficient approximation for triangulation of minimum treewidth. In *UAI*, 7–15.

Becker, A., and Geiger, D. 1996. A sufficiently fast algorithm for finding close to optimal junction trees. In *UAI*, 81–89.

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *AAAI*.

Brock, O., and Khatib, O. 2000. Real-time replanning in high-dimensional configuration spaces using sets of homotopic paths. In *ICRA'00*, 550–555.

Canny, J. 1987. *The Complexity of Robot Motion Planning*. Cambridge, MA: MIT Press.

Chen, P., and Hwang, Y. 1991. Motion planning for a robot and a movable object amidst polygonal obstacles. In *ICRA'91*, 444–449.

Choi, J., and Amir, E. 2007. Factor-guided motion planning for a robot arm. In *IROS'07*, 27–32.

Conner, D. C.; Kress-Gazit, H.; Choset, H.; Rizzi, A.; and Pappas, G. J. 2007. Valet parking without a valet. In *IROS'07*.

Cortés, J. 2003. *Motion Planning Algorithms for General Closed-Chain Mechanisms*. Ph.D. Dissertation, Institut National Polytechnique de Toulouse, Toulouse, France.

Dacre-Wright, B.; Laumond, J.-P.; and Alami, R. 1992. Motion planning for a robot and a movable object amidst polygonal obstacles. In *ICRA'92*, volume 3, 2474–2480.

Dejong, G., and Mooney, R. 1986. Explanation-based learning: An alternative view. *Mach. Learn.* 1(2):145–176.

Fourman, M. 2007. Propplan. Software.

J. Van den Berg, M. O. 2007. Kinodynamic motion planning on roadmaps in dynamic environments. In *IROS'07*.

Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Trans. on Rob. and Auto.* 12(4):566–580.

Kuffner, J. J., and LaValle, S. M. 2000. RRT-connect: An efficient approach to single-query path planning. In *ICRA'00*.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* search with provable bounds on sub-optimality. In *NIPS'03*.

M. Pardowitz, R. Zollner, R. D. 2007. Incremental acquisition of task knowledge applying heuristic relevance estimation. In *IROS'07*.

McDermott, D. 1998. The planning domain definition language manual.

Plaku, E.; Kavraki, L. E.; and Vardi, M. Y. 2008. Hybrid systems: From verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design*.

S. Hart, R. G. 2007. Natural task decomposition with intrinsic potential fields. In *IROS'07*.

Shahaf, D., and Amir, E. 2007. Logical circuit filtering. In *IJCAI*, 2611–2618.

Stilman, M., and Kuffner, J. 2005. Navigation among movable obstacles: Real-time reasoning in complex environments. *International Journal of Humanoid Robotics* 2(4):479–504.

Stilman, M. 2007. Task constrained motion planning in robot joint space. In *IROS'07*.

Fast replanning

Koenig, Sven
USC, Los Angeles

Planning systems for mobile robots often need to operate in domains that are only incompletely known or change dynamically. In this case, they need to replan quickly as their knowledge changes. Replanning from scratch is often very time consuming. In this talk, I will discuss ongoing research by us and others on incremental heuristic search. Incremental search methods reuse information from previous searches to find solutions to series of similar search tasks potentially much faster than is possible by solving each search task from scratch, while heuristic search methods use heuristic knowledge in form of approximations of the goal distances to focus the search and solve search problems potentially much faster than uninformed search methods. I will discuss the theory behind incremental heuristic search and several of its applications. This is joint work with my students and ex-students C. Bauer, D. Furcy, M. Likhachev, Y. Liu, A. Ranganathan and X. Sun.

Spatial computing - or how to design a right brain hemisphere

Freksa, Christian
Universitat Bremen

Spatial problems come in a variety of forms: as physical tasks as in the piano movers problem; or more abstractly as in natural language or in a formal geometric specification; or in some form in between as in a diagram that exhibits aspects of physical space and aspects of abstractions. Similarly, we have a variety of options for solving spatial problems ranging from concrete to abstract. In my talk, I will present different ways in which we can solve spatial problems and I will discuss some of the features of the different approaches. I will emphasize cognitive aspects of problem solving and I propose selecting reasoning methods with regards to the contexts of their respective task environments. From a cognitive perspective, a particularly relevant method is to use spatial structures for solving spatial problems. I will call this approach Spatial Computing. Spatial Computing enables us solving problems of a certain type in a particularly efficient way. I will address the question whether the principles underlying Spatial Computing can be exploited for the development of special spatial computers; and if so, whether these computers will be restricted to solving spatial problems or whether they may be of more general use. I will suggest that humans employ principles of Spatial Computing for solving problems that go far beyond spatial problems.

Cognitive robotics, embodied cognition and human-robot interaction

J. Gregory Trafton
Naval Research Lab
Washington, DC

In the past few years, we have been building embodied cognitive models. We use embodied representations in both the online and the offline sense (Wilson, 200Z). We use online cognition for "here and now" interactions (Brooks, 1990) while we use offline cognition by using body based representations for thinking even when decoupled from the environment (e.g., Tucker and Ellis, 2001). We build our models to be as cognitively plausible as possible, using cognitive representations, strategies, and procedures with the belief that this makes human-robot interaction easier and stronger.

We have a number of online models, including gaze-following (Trafton, Harrison, Fransen, and Bugajska, 2009) and level 1 visual perspective taking (Trafton and Harrison, under review). We also have a model of offline cognition based on Tucker and Ellis' 2001 data (Harrison and Trafton, in press).

We believe that our process models of human cognition not only help us understand how people perform different tasks but also provide a different approach to human-robot interaction.

Harrison, A. M., and Trafton, J. G. (in press). Cognition for action: an architectural account for grounded interaction. Proceedings of the Cognitive Science Society, 2010.

Trafton and Harrison (under review). Embodied Spatial Cognition.

Trafton, J.G., Harrison, A.M., Fransen, B.R., and Bugajska, M. (2009) An embodied model of infant gaze- following. In A. Howes, D. Peebles, R. Cooper (Eds.), 9th International conference on cognitive modeling - ICCM2009, Manchester, UK.

Tucker, M., and Ellis, R. (2001) The potentiation of grasp types during visual object categorization. V isual cognition, 8, 769-800.

Wilson, M. (2002). Six views of embodied cognition. Psychonomic Bulletin and Review.

Biologically Inspired Cognitive Architectures II:
Papers from the AAAI Fall Symposium (FS-09-01)

The GLAIR Cognitive Architecture

Stuart C. Shapiro and **Jonathan P. Bona**
Department of Computer Science and Engineering and Center for Cognitive Science
State University of New York at Buffalo
{shapiro|jpbona}@buffalo.edu

Abstract

GLAIR (Grounded Layered Architecture with Integrated Reasoning) is a multi-layered cognitive architecture for embodied agents operating in real, virtual, or simulated environments containing other agents. The highest layer of the GLAIR Architecture, the Knowledge Layer (KL), contains the beliefs of the agent, and is the layer in which conscious reasoning, planning, and act selection is performed. The lowest layer of the GLAIR Architecture, the Sensori-Actuator Layer (SAL), contains the controllers of the sensors and effectors of the hardware or software robot. Between the KL and the SAL is the Perceptuo-Motor Layer (PML), which grounds the KL symbols in perceptual structures and subconscious actions, contains various registers for providing the agent's sense of situatedness in the environment, and handles translation and communication between the KL and the SAL. The motivation for the development of GLAIR has been "Computational Philosophy", the computational understanding and implementation of human-level intelligent behavior without necessarily being bound by the actual implementation of the human mind. Nevertheless, the approach has been inspired by human psychology and biology.

1. Introduction

GLAIR (Grounded Layered Architecture with Integrated Reasoning) is a multi-layered cognitive architecture for embodied agents operating in real, virtual, or simulated environments containing other agents (Hexmoor, Lammens, and Shapiro 1993; Lammens, Hexmoor, and Shapiro 1995; Shapiro and Ismail 2003). It was an outgrowth of the SNePS Actor (Kumar and Shapiro 1991). Our motivating goal has been what is called "Computational Philosophy" in (Shapiro 1992), that is, the computational understanding and implementation of human-level intelligent behavior without necessarily being bound by the actual implementation of the human mind. Nevertheless, our approach has been inspired by human psychology and biology.

Although GLAIR is a cognitive architecture appropriate for implementing various cognitive agents, we tend to name all our cognitive agents "Cassie." So whenever in this paper we refer to Cassie, we mean one or another of our implemented GLAIR agents.

Copyright © 2009, Stuart C. Shapiro and Jonathan P. Bona. All rights reserved.

2. GLAIR as a Layered Architecture

2.1 The Layers

The highest layer of the GLAIR Architecture, the Knowledge Layer (KL), contains the beliefs of the agent, and is the layer in which conscious reasoning, planning, and act selection is performed.

The lowest layer of the GLAIR Architecture, the Sensori-Actuator Layer (SAL), contains the controllers of the sensors and effectors of the hardware or software robot.

Between the KL and the SAL is the Perceptuo-Motor Layer (PML), which, itself is divided into three sublayers. The highest, the PMLa, grounds the KL symbols in perceptual structures and subconscious actions, and contains various registers for providing the agent's sense of situatedness in the environment. The lowest of these, the PMLc, directly abstracts the sensors and effectors into the basic behavioral repertoire of the robot body. The middle PML layer, the PMLb, handles translation and communication between the PMLa and the PMLc.

2.2 Mind-Body Modularity

The KL constitutes the mind of the agent; the PML and SAL, its body. However, the KL and PMLa layers are independent of the implementation of the agent's body, and can be connected, without modification, to a hardware robot or to a variety of software-simulated robots or avatars. Frequently, the KL, PMLa, and PMLb have run on one computer; the PMLc and SAL on another. The PMLb and PMLc handle communication over I/P sockets.[1]

3. The KL: Memory and Reasoning

The KL contains the beliefs of the agent, including: short-term and long-term memory; semantic and episodic memory; quantified and conditional beliefs used for reasoning; plans for carrying out complex acts and for achieving goals; beliefs about the preconditions and effects of acts; policies about when, and under what circumstances, acts should be performed; self-knowledge; and metaknowledge.

The KL is the layer in which conscious reasoning, planning, and act selection is performed. The KL is implemented

[1] Other interprocess communication methods might be used in the future.

in SNePS (Shapiro and Rapaport 1992; Shapiro 2000b; Shapiro and The SNePS Implementation Group 2008), which is simultaneously a logic-based, frame-based, and network-based knowledge representation and reasoning system, that employs various styles of inference as well as belief revision.

As a logic-based KR system, SNePS implements a predicate logic with variables, quantifiers, and function symbols. Although equivalent to First-Order Logic, its most unusual feature is that every well-formed expression is a term, even those that denote propositons (Shapiro 1993). This allows for metapropositions, propositions about propositions, without restriction and without the need for an explicit `Holds` predicate (Morgado and Shapiro 1985; Shapiro et al. 2007). For example the asserted term, `Believe(B8,Rich(B8))` in the context of the asserted term, `Propername(B8,Oscar)`, denotes the proposition that Oscar believes himself to be rich (Rapaport, Shapiro, and Wiebe 1997). SNePS supports forward- backward- and bidirectional-reasoning (Shapiro 1987; Shapiro, Martins, and McKay 1982) using a natural-deduction proof theory, and belief revision (Martins and Shapiro 1988).

Every functional term in SNePS is represented as an assertional frame in which the argument positions are slots and the arguments are fillers. This allows for sets of arguments to be used to represent combinatorially many assertions. For example, `instanceOf({Fido, Lassie, Rover}, {dog, pet})` might be used to represent the assertion that Fido, Lassie, and Rover are dogs and pets. It also allows sets to be used for symmetric relationships, for example `adjacent({US, Canada})` can represent the assertion that the US and Canada are adjacent to each other (Shapiro 1986). The frame view of SNePS supports "slot-based inference", whereby an asserted frame logically implies one with a subset or superset of fillers in given slots (Shapiro 2000a).

By treating the terms as nodes and the slots as labeled directed arcs, SNePS can be used as a propositional network (Shapiro and Rapaport 1987). This supports a style of inference driven by following paths in the network (Shapiro 1978; 1991).

3.1 The Active Connection Graph

Reasoning is performed by an active connection graph (ACS) (McKay and Shapiro 1980; 1981). Viewing the SNePS knowledge base as a propositional graph, every proposition-denoting term can be considered to be a node with arcs pointing to its arguments. This includes non-atomic propositions such as implications, each of which has one set of arcs pointing to its antecedents and another pointing to its consequents. Each proposition has a process charged with collecting and transmitting inferred instances of its propositions along the arcs to interested other processes in a multiprocessing, producer-consumer, message-passing system (Shubin 1981). This allows recursive rules to be used without getting into an infinite loop, and prevents the same inference from being worked on multiple times even if it is a subgoal in multiple ways (McKay and Shapiro 1981), and has not yet been satisfied.

The ACS is key to SNePS' bidirectional inference (Shapiro, Martins, and McKay 1982; Shapiro 1987). Inference processes are created both by backward inference and by forward inference. If such a process is needed and already exists, a forward-chaining process (producer) adds its results to the process's collection, and a backward-chaining process (consumer) is added to the producer-process's consumers to be notified. If a query is asked that can't be answered, the processes established for it remain, and can be found be subsequent forward inferences. When new beliefs are added to the KL with forward inference, and existing consumer-processes are found for them, new consumer-processes are not established. The result of this is that after a query, additional new information is considered in light of this concern. In other words, a GLAIR agent working on a problem considers relevant new data only as it relates to that problem, focussing its attention on it.

The ACS can be deleted. It is then reestablished the next time a forward- or backward- inference begins. In this way the GLAIR agent changes its attention from one problem to another. When this change of attention happens is, however, currently rather *ad hoc*. A better theory of when it should happen is a subject of future research.

3.2 Contexts

Propositions may be asserted in the KL because they entered from the environment. Either they were told to the agent by some other agent, possibly a human, or they are the result of some perception. Alternatively, a proposition might be asserted in the KL because it was derived by reasoning from some other asserted propositions. We call the former *hypotheses* and the latter *derived propositions.* When a proposition is derived, an *origin set,* consisting of the set of hypotheses used to derive it is stored with it (Martins and Shapiro 1988) à la an ATMS (de Kleer 1986). At each moment, some particular *context,* consisting of a set of hypotheses, is current. The asserted propositions, the propositions the GLAIR agent believes, are the hypotheses of the current context and those derived propositions whose origin sets are subsets of that set of hypotheses. If some hypothesis is removed from the current context (*i.e.,* is disbelieved), the derived propositions that depended on it remain in the KL, but are no longer believed. If all the hypotheses in the origin set of a derived proposition return to the current context, the derived proposition is automatically believed again, without having to be rederived (Martins and Shapiro 1983; Shapiro 2000b).

4. The PMLa

The PMLa, contains: the subconscious implementation of the cognitively primitive actions of the KL; the structures used for the perception of objects and properties in the environment; various registers for providing the agent's sense of situatedness in the environment, such as its sense of "I", "You", "Now", and the actions it is currently engaged in; and procedures for natural language comprehension and generation. Further discussion of the PMLa, and its connections to the KL may be found in §9. and (Shapiro and Ismail 2003).

5. The Behavior Cycle

Several cognitive architectures, such as ACT-R (Anderson and Lebiere 1998), Soar (Laird, Newell, and Rosenbloom 1987), Icarus (Langley, Cummings, and Shapiro 2004), and PRODIGY (Carbonell, Knoblock, and Minton 1990) are based on problem-solving or goal-achievement as their basic driver. GLAIR, on the contrary, is based on reasoning: either thinking about some percept (often linguistic input), or answering some question. The acting component is a more recent addition, allowing an GLAIR agent also to obey a command, either to perform an act or to achieve a goal. However, the focus of the design remains on reasoning. Problem solving *vs.* reasoning, however, are not incompatible tasks, but alternative approaches to the ultimate goal of achieving an AI-complete (Shapiro 1992) system.

GLAIR agents execute a sense-reason-act cycle, but not necessarily in a strict cyclical order. GLAIR was developed around implementations of SNePS as an interactive natural language comprehension, knowledge representation, and reasoning system. The basic behavior cycle is:

1. input a natural language utterance.
2. analyze the utterance in the context of the current beliefs
   - the analysis may require and trigger reasoning
   - the analysis may cause new beliefs to be added to the KL
3. if the utterance is a statement
   (a) add the main proposition of the statement as a belief
   (b) that proposition will be output

   if the utterance is a question
   (a) perform backward reasoning to find the answer to the question
   (b) the answer will be output

   if the utterance is a command
   (a) perform the indicated act
   (b) the proposition that the agent performed the act will be output
4. generate a natural language utterance expressing the output proposition
   - reasoning may be performed to formulate the utterance

The categorization of input into either statement (informative), question (interrogative), or command (imperative) assumes that there are no indirect speech acts (Searle 1975) or that the real speech act has already been uncovered. An alternative would be to represent each input as *"X said S,"* and reason about what the agent should do about it. Natural language analysis and generation is an optional part of the GLAIR architecture. If it is omitted, the utterance is expressed in a formal language, such as SNePSLOG (Shapiro and The SNePS Implementation Group 2008) (the formal language used in this paper) and only step (3) is performed.

If this input-reason-output behavior cycle seems too restricted for a cognitive agent, note that the input might be "Perform a", where a is an act, or "Achieve g", where g is a goal, and that might start an arbitrarily long sequence of behaviors. In fact, any of the reasoning episodes might trigger afferent or efferent acts, and any act might trigger reasoning (Kumar 1993; Kumar and Shapiro 1994).

There can be both passive and active sensing. Passive sensing, such as seeing the environment as the agent navigates through it, may result in percepts that, in a data-driven fashion, motivate the agent to perform some act. Active sensing, such as attending to some specific aspect of the environment, may be used in a goal-directed fashion to gain particular information that can be used to decide among alternative acts. For example, we have implemented a GLAIR delivery agent that navigates the hallways of one floor of a simulated building, and may be told to get a package from one room, and deliver it to another. A primitive act of this agent is `goForward()`: " move one unit in the direction it is facing." As a result of such a move, and without another act on its part, it believes either that it is facing a room, a blank wall, or more corridor. Adding the appropriate belief to the KL is built into the PMLa implementation of `goForward()`, and is an example of passive sensing. On the other hand, if the agent needs to know where it is, and it is facing a room, it can deliberately read the room number by performing the primitive act, `readRoomNumber()`. This is an example of active sensing.

6. The Acting Model[2]

GLAIR's acting model consists of: actions and acts; propositions about acts; and policies.

6.1 Policies

Policies specify circumstances under which reasoning leads to action. An example of a policy is, "when the walk light comes on, cross the street." Policies are neither acts nor propositions. We say that an agent *performs* an act, *believes* a proposition, and `adopts` a policy. To see that policies are not acts, note that one cannot perform "when the walk light comes on, cross the street." A good test for an expression ϕ being a proposition is its ability to be put in the frame, "I believe that it is not the case that ϕ." It does not make sense to say, "I believe that it is not the case that when the walk light comes on, cross the street." Note that this is different than saying, "I believe that it is not the case that when the walk light comes on, I *should* cross the street." An agent might explicitly believe "I *should* cross the street" without actually doing it. However, if an GLAIR agent has adopted the policy, "when the walk light comes on, cross the street," and it comes to believe that the walk light is on, it *will* cross the street (or at least try to).

Policies are represented as functional terms in the KL, along with other conscious memory structures. Three policy-forming function symbols are built into GLAIR, each of which take as arguments a proposition ϕ and an act α:

- `ifdo`(ϕ, α) is the policy, "to decide whether or not ϕ, perform α";
- `whendo`(ϕ, α) is the policy, "when ϕ holds, perform α";

[2] Parts of this section were taken from (Shapiro et al. 2007).

- `wheneverdo(`ϕ, α`)` is the policy, "whenever ϕ holds, perform α".

A blocks-world example of `ifdo` is "*To decide whether block A is red, look at it*": `ifdo(ColorOf(A,red), lookAt(A))` (Kumar and Shapiro 1994).[3]

The policies `whendo` and `wheneverdo` are similar to the production rules of production systems in that they are condition-action rules triggered when forward-chaining matches the condition. In the case of both `whendo` and `wheneverdo`, if the policy has been adopted, the agent performs α when forward inference causes ϕ to be believed. Also, α is performed if ϕ is already believed when the policy is adopted. The difference is that a `whendo` policy is unadopted after firing once, but a `wheneverdo` remains adopted until explicitly unadopted.

6.2 Categories of Acts

An act may be performed by an agent, and is composed of an *action* and zero or more arguments. For example, for the Fevahr[4] version of Cassie (Shapiro 1998) (henceforth Cassie$_F$), the term `find(Bill)` denotes the act of finding Bill (by looking around in a room for him), composed of the action `find` and the object `Bill`.[5]

Acts may be categorized on two independent dimensions: an act may be either an external, a mental, or a control act; and an act may be either a primitive, a defined, or a composite act.

External, Mental, and Control Acts Actions and, by extension, acts, may be subclassified as either external, mental, or control. External acts either sense or affect the real, virtual, or simulated outside world. An example mentioned above from the Fevahr version of Cassie is `find(Bill)`. No external acts are predefined in the architecture; they must be supplied by each agent designer.

Mental acts affect the agent's beliefs and policies. There are four:

1. `believe(`ϕ`)` is the act of asserting the proposition ϕ and doing forward inference on it;
2. `disbelieve(`ϕ`)` is the act of unasserting the proposition ϕ, so that it is not believed, but its negation is not necessarily believed;
3. `adopt(`π`)` is the act of adopting the policy π;
4. `unadopt(`π`)` is the act of unadopting the policy π.

Before `believe` changes the belief status of a proposition ϕ, it performs a limited form of prioritized belief revision (Alchourrón, Gärdenfors, and Makinson 1985). If `andor(0,0)`$\{\ldots, \phi, \ldots\}$ is believed,[6] it is `disbelieve`d. If `andor(`i`,1)`$\{\phi_1, \phi_2, \ldots\}$ is believed, for $i = 0$ or $i = 1$, and ϕ_2 is believed, ϕ_2 is `disbelieve`d.

Control acts are the control structures of the GLAIR acting system. The predefined control actions are:

- `achieve(`ϕ`)`: If the proposition ϕ is not already believed, infer plans for bringing it about, and then perform `do-one` on them.
- `snsequence(`α_1, α_2`)`: Perform the act α_1, and then the act α_2.
- `prdo-one({pract(`x_1, α_1`), ..., pract(`x_n, α_n`)})`: Perform one of the acts α_j, with probability $x_j / \sum_i x_i$.
- `do-one(`$\{\alpha_1, \ldots, \alpha_n\}$`)`: Nondeterministically choose one of the acts $\alpha_1, \ldots, \alpha_n$, and perform it.
- `do-all(`$\{\alpha_1, \ldots, \alpha_n\}$`)`: Perform all the acts $\alpha_1, \ldots, \alpha_n$ in a nondeterministic order.
- `snif({if(`ϕ_1, α_1`), ..., if(`ϕ_n, α_n`), [else(`δ`)]})`: Use backward inference to determine which of the propositions ϕ_i hold, and, if any do, nondeterministically choose one of them, say ϕ_j, and perform the act α_j. If none of the ϕ_i can be inferred, and if `else(`δ`)` is included, perform δ. Otherwise, do nothing.
- `sniterate({if(`ϕ_1, α_1`), ..., if(`ϕ_n, α_n`), [else(`δ`)]})`: Use backward inference to determine which of the propositions ϕ_i hold, and, if any do, nondeterministically choose one of them, say ϕ_j, and perform the act `snsequence(`α_j`, sniterate({if(`ϕ_1, α_1`), ..., if(`ϕ_n, α_n`), [else(`δ`)]}))`. If none of the ϕ_i can be inferred, and if `else(`δ`)` is included, perform δ. Otherwise, do nothing.
- `withsome(`$x, \phi(x), \alpha(x), [\delta]$`)`: Perform backward inference to find entities e such that $\phi(e)$ is believed, and, if such entities are found, choose one of them nondeterministically, and perform the act α on it. If no such e is found, and the optional act δ is present, perform δ.
- `withall(`$x, \phi(x), \alpha(x), [\delta]$`)`: Perform backward inference to find entities e such that $\phi(e)$ is believed, and, if such entities are found, perform the act α on them all in a nondeterministic order. If no such e is found, and the optional act δ is present, perform δ.

The acts `snif`, `sniterate`, `withsome`, and `withall` all trigger reasoning. The default implementation of `do-one` uses a pseudorandom number generator to choose the act to perform, and the default implementation of `do-all` uses a pseudorandom number generator to choose the order of the acts. However, an agent implementer may replace either pseudorandom number generator with reasoning rules to make the choice, in which case these acts will also trigger reasoning.

[3] `ifdo` was called `DoWhen` in (Kumar and Shapiro 1994).

[4] "Fevahr" is an acronym standing for "Foveal Extra-Vehicular Activity Helper-Retriever".

[5] Actually, since the Fevahr Cassie uses a natural language interface, the act of finding Bill is represented by the term `act(lex(find),b6)`, where: `find` is a term aligned with the English verb *find*; `lex(find)` is the action expressed in English as *"find"*; and `b6` is a term denoting Bill. However, we will ignore these complications in this paper.

[6] `andor` (Shapiro 1979) is a parameterized connective that takes a set of argument-propositions, and generalizes *and*, *inclusive or*, *exclusive or*, *nand*, *nor*, and *exactly n of*. A formula of the form `andor(`i, j`)`$\{\phi_1, \ldots, \phi_n\}$ denotes the proposition that at least i and at most j of the ϕ_k's are true.

Primitive, Defined, and Composite Acts GLAIR actions and acts may also be classified as either primitive, defined, or composite. Primitive acts constitute the basic repertoire of an GLAIR agent. They are either provided by the architecture itself, or are implemented at the PMLa. An example predefined action is `believe`; an example primitive action defined at the PMLa is the Fevahr Cassie's `find` (Shapiro 1998). Because primitive actions are implemented below the KL, GLAIR agents have no cognitive insight into how they perform them.

A composite act is one structured by one of the control acts. For example, the Wumpus-World Cassie (Shapiro and Kandefer 2005), whose only primitive turning acts are `go(right)` and `go(left)`, can turn around by performing the composite act, `snsequence(go(right), go(right))`.

A defined act is one that, unlike composite acts, is syntactically atomic, and unlike primitive acts, is not implemented at the PML. If a GLAIR agent is to perform a defined act α, it deduces plans p for which it believes the proposition `ActPlan`(α, p), and performs a `do-one` of them. Such a plan is an act which, itself, can be either primitive, composite, or defined. For example, the Wumpus-World Cassie has a defined act `turn(around)`, which is defined by `ActPlan(turn(around), snsequence(go(right), go(right)))`.

6.3 Propositions About Acts

Four propositions about acts are predefined parts of the GLAIR architecture:

1. `Precondition`(α, ϕ): In order for the agent to perform the act α, the proposition ϕ must hold.
2. `Effect`(α, ϕ) An effect of an agent's performing the act α is that the proposition ϕ will hold. The proposition ϕ could be a negation, to express the effect that some proposition no longer holds, such as `Effect(putOn(`x, y`), clear(`y`))`.
3. `ActPlan`(α, p): One way to perform the act α is to perform the plan p.
4. `GoalPlan`(ϕ, p): One way to achieve the goal that the proposition ϕ holds is to perform the plan p.

The only difference between a "plan" and an "act" is that a plan is an act that appears in the second argument position of an `ActPlan` or a `GoalPlan` proposition. However, in a proposition of the form `ActPlan`(α, p), it is assumed that p is "closer" to primitive acts than α is.

6.4 Conditional Plans

Consider a defined act for which there are different plans depending on circumstances. For example, to get the mail, if I'm at home, I go to the mailbox, but if I'm in the office, I go to the mailroom. Such conditional plans may be represented by implications:

```
at(home) => ActPlan(get(mail),
                    go(mailbox))
at(office) => ActPlan(get(mail),
                      go(mailroom))
```

```
perform(act):
  pre := {p | Precondition(act,p)};
  notyet := pre - {p | p ∈ pre & ⊢ p};
  if notyet ≠ nil
    then
       perform(
         snsequence(
           do-all({a | p ∈ notyet
                        & a = achieve(p)}),
           act))
    else
      {effects := {p | Effect(act,p)};
       if act is primitive
         then apply(
                primitive-function(act),
                objects(act));
         else perform(
                do-one({p
                        | ActPlan(act,p)}))
       believe(effects)}
```

Figure 1: The acting executive

In a context in which `at(home)` is derivable, the plan for getting the mail will be `go(mailbox)`. When the context changes so that `at(home)` is no longer derivable, `ActPlan(get(mail), go(mailbox))` will no longer be asserted, nor derivable. However, when the context is reentered, `ActPlan(get(mail), go(mailbox))` will again be asserted without the need to rederive it.

7. The Acting Executive

The procedure for performing an act is shown in Fig. 1. Notice that:

- Backward inference is triggered to find:
  - the preconditions of `act`;
  - whether each precondition currently holds;
  - the effects of `act`;
  - plans that can be used to perform `act`, if `act` is not primitive.
- After the attempt is made to achieve the preconditions of `act`, `perform(act)` is called again, which will again check the preconditions, in case achieving some of them undid the achievement of others.
- Effects of `act` are derived before `act` is performed in case the effects depend on the current state of the world.
- If `act` is a defined act, only one way of performing it is tried, and that is assumed to be successful. This will be changed in future versions of GLAIR.
- After `act` is performed, all its effects are believed to hold. This is naive, and will be changed in future versions of GLAIR. We have already implemented GLAIR agents that only believe the effects of their acts that they sense holding in the world, but this has been done by giving them no `Effect` assertions.

8. Modalities

Especially on hardware robots, the sensors and effectors can operate simultaneously. To take advantage of this, GLAIR supports a set of modalities. A modality represents a limited resource—a PMLc-level behavior that is limited in what it can do at once (for example, a robot cannot go forward and backward at the same time), but is independent of the behaviors of other modalities (a robot can navigate and speak at the same time). Each modality runs in a separate thread, and uses its own communication channel between the PMLb and PMLc layers. Each KL primitive action is assigned, at the PMLa layer, to one or more modalities. Modalities that have been implemented in various GLAIR agents include speach, hearing, navigation, and vision. We intend to make the organization into modalities a more thoroughgoing and pervasive principle of the architecture. That version of the architecture will be called MGLAIR.

9. Symbol Anchoring[7]

9.1 Alignment

There are KL terms for every mental entity Cassie has conceived of, including individual entities, categories of entities, colors, shapes, and other properties of entities. There are PML structures (at the PMLb and PMLc sub-levels) for features of the perceivable world that Cassie's perceptual apparatus can detect and distinguish. Each particular perceived object is represented at this level by an n-tuple of such structures, $\langle v_1, \ldots, v_n \rangle$, where each component, v_i, is a possible value of some perceptual feature domain, D_i. What domains are used and what values exist in each domain depend on the perceptual apparatus of the robot. We call the n-tuples of feature values "PML-descriptions".

Each KL term for a perceivable entity, category, or property is grounded by aligning it with a PML-description, possibly with unfilled (null) components. For example, Cassie$_F$ used two-component PML-descriptions in which the domains were color and shape. The KL term denoting Cassie$_F$'s idea of blue was aligned with a PML-description whose color component was the PML structure the vision system used when it detected blue in the visual field, but whose shape component was null. The KL term denoting people was aligned with a PML-description whose shape component was the PML structure the vision system used when it detected a people in the visual field, but whose color component was null.

Call a PML-description with some null components an "incomplete PML-description", and one with no null components a "complete PML-description". KL terms denoting perceivable properties and KL terms denoting recognizable categories of entities are aligned with incomplete PML-descriptions. Examples include the terms for blue and for people mentioned above, and may also include terms for the properties tall, fat, and bearded, and the categories man and woman. The words for these terms may be combined into verbal descriptions, such as "a tall, fat, bearded man," whose incomplete PML-descriptions may be used to perceptually recognize the object corresponding to the entity so described.

A complete PML-description may be assembled for an entity by unifying the incomplete PML-descriptions of its known (conceived-of) properties and categories. Once a PML-description is assembled for an entity, it is cached by aligning the term denoting the entity directly with it. Afterwards, Cassie can recognize the entity without thinking about its description. On the other hand, Cassie may have a complete PML-description for some object without knowing any perceivable properties for it. In that case, Cassie would be able to recognize the object, even though she could not describe it verbally.

If Cassie is looking at some object, she can recognize it if its PML-description is the PML-description of some entity she has already conceived of. If there is no such entity, Cassie can create a new KL term to denote this new entity, align it with the PML-description, and believe of it that it has those properties and is a member of those categories whose incomplete PML-descriptions unify with the PML-description of the new entity. If there are multiple entities whose PML-descriptions match the object's PML-description, disambiguation is needed, or Cassie might simply not know which one of the entities she is looking at.

9.2 Deictic Registers

An important aspect of being embodied is being situated in the world and having direct access to components of that situatedness. This is modeled in GLAIR via a set of PML registers (variables), each of which can hold one or more KL terms or PML structures. Some of these registers derive from the theory of the Deictic Center (Duchan, Bruder, and Hewitt 1995), and include: `I`, the register that holds the KL term denoting the agent itself; `YOU`, the register that holds the KL term denoting the individual the agent is talking with; and `NOW`, the register that holds the KL term denoting the current time.

9.3 Modality Registers

GLAIR agents know what they are doing via direct access to a set of PML registers termed "modality registers". For example, if one of Cassie's modalities were speech, and she were currently talking to Stu, her `SPEECH` register would contain the KL term denoting the state of Cassie's talking to Stu (and the term denoting Stu would be in the `YOU` register). In many cases, a single modality of an agent can be occupied by only one activity at a time. In that case the register for that modality would be constrained to contain only one term at a time.

One of the modality registers we have used is for keeping track of what Cassie is looking at. When she recognizes an object in her visual field, the KL term denoting the state of looking at the recognized entity is placed in the register, and is removed when the object is no longer in the visual field. If one assumed that Cassie could be looking at several objects at once, this register would be allowed to contain several terms. If asked to look at or find something that is already

[7]This section is taken from (Shapiro and Ismail 2003).

in her visual field, Cassie recognizes that fact, and doesn't need to do anything.

9.4 Actions

Each KL action term that denotes a primitive action is aligned with a procedure in the PMLa. The procedure takes as arguments the KL terms for the arguments of the act to be performed. For example, when Cassie is asked to perform the act of going to Bill, the PMLa going-procedure is called on the KL Bill-term. It then finds the PML-description of Bill, and (via the SAL) causes the robot hardware to go to an object in the world that satisfies that description (or causes the robot simulation to simulate that behavior). The PMLa going-procedure also inserts the KL term denoting the state of Cassie's going to Bill into the relevant modality register(s), which, when `NOW` moves , causes an appropriate proposition to be inserted into Cassie's KL.

9.5 Time

As mentioned above, the `NOW` register always contains the KL term denoting the current time (Shapiro 1998; Ismail 2001; Ismail and Shapiro 2000; 2001). Actually, since "now" is vague (it could mean this minute, this day, this year, this century, etc.), `NOW` is considered to include the entire semi-lattice of times that include the smallest current now-interval Cassie has conceived of, as well as all other times containing that interval.

`NOW` moves whenever Cassie becomes aware of a new state. Some of the circumstances that cause her to become aware of a new state are: she acts, she observes a state holding, she is informed of a state that holds. `NOW` moves by Cassie's conceiving of a new smallest current now-interval (a new KL term is introduced with that denotation), and `NOW` is changed to contain that time. The other times in the old `NOW` are defeasibly extended into the new one by adding propositions asserting that the new `NOW` is a subinterval of them.

Whenever Cassie acts, the modality registers change (see above), and `NOW` moves. The times of the state(s) newly added to the modality registers are included in the new `NOW` semi-lattice, and the times of the state(s) deleted from the modality registers are placed into the past by adding propositions that assert that they precede the new `NOW`.

To give GLAIR agents a "feel" for the amount of time that has passed, the PML has a `COUNT` register acting as an internal pacemaker (Ismail 2001; Ismail and Shapiro 2001). The value of `COUNT` is a non-negative integer, incremented at regular intervals. Whenever `NOW` moves, the following happens:

1. the value of `COUNT` is quantized into a value δ which is the nearest half-order of magnitude (Hobbs 2000) to `COUNT`, providing an equivalence class of PML-measures that are not noticeably different;
2. a KL term d, aligned with δ, is found or created, providing a mental entity denoting each class of durations;
3. a belief is introduced into the KL that the duration of t_1, the current falue of `NOW,` is d, so that the agent can have beliefs that two different states occurred for about the same length of time;
4. a new KL term, t_2 is created and a belief is introduced into the KL that t_1 is before t_2;
5. `NOW` is reset to t_2;
6. `COUNT` is reset to 0, to prepare for measuring the new now-interval.

9.6 Language

Cassie interacts with humans in a fragment of English. Although it is possible to represent the linguistic knowledge of GLAIR agents in the KL, use reasoning to analyze input utterances (Neal and Shapiro 1985; 1987b; 1987a; Shapiro and Neal 1982), and use the acting system to generate utterances (Haller 1996; 1999), we do not currently do this. Instead, the parsing and generation grammars, as well as the lexicon, are at the PML. (See, e.g. (Rapaport, Shapiro, and Wiebe 1997; Shapiro 1982; Shapiro and Rapaport 1995).) There are KL terms for lexemes, and these are aligned with lexemes in the PML lexicon. We most frequently use a KL unary functional term to denote the concept expressed by a given lexeme, but this does not allow for polysemy, so we have occasionally used binary propositions that assert that some concept may be expressed by some lexeme. There may also be KL terms for inflected words, strings of words, and sentences. This allows one to discuss sentences and other language constructs with GLAIR agents.

10. Bodily Feedback

The control acts `snsequence`, `do-all`, `sniterate`, and `withall` each cause a sequence of acts to be performed before it is completed. In a normal, single-processor, procedural/functional architecture this would not cause a problem as each act in the sequence would be performed only after the previous one returns control to the control act. However, in GLAIR, primitive acts are performed in modalities operating concurrently with reasoning, so it is important for the control act to get feedback from the body that an act has completed before it proceeds to the next act in the sequence. Think of the problem deaf people have speaking at a "normal" rate without being able to hear themselves. In previous agents (Shapiro et al. 2005b; 2005c), bodily feedback for the speech modality was provided for via the hearing modality, but this was included explicitly at the KL and using a special `pacedSequence` act. We intend to build bodily feedback directly into the GLAIR architecture in the future.

11. Properties of cognitive architectures

In this section, we discuss GLAIR using propertes listed in (Langley, Laird, and Rogers 2009).

11.1 Representation of knowledge

Knowledge (more properly, beliefs) is represented in the GLAIR Knowledge Layer in SNePS, which is simultaneously a logic-based, assertional frame-based, and

graph(network)-based knowledge representation and reasoning system. Noteworthy features of the SNePS representation are: every well-formed expression is a term, even those denoting propositions; all beliefs and conceived-of entities are represented in the same formalism, including reasoning rules (such as conditionals) and acting plans. SNePS is more fully discussed above and in the cited papers.

Single notation vs. Mixture of formalisms Although all knowledge is represented in a single formalism, namely SNePS, SNePS, itself, is simultaneously three different formalisms: logic-based, which supports a natural-deduction-style inference mechanism; assertional frame-based, which supports inference from one frame to another with a subset or superset of fillers in some of the slots; and graph/network-based, which supports inference of labeled arcs from the presence of paths of labeled arcs.

Support for metaknowledge Since every SNePS expression is a term, including those that denote propositions, propositions about propositions may be represented without restriction and without the need for an explicit `Holds` predicate. The default acts included as options in `snif`, `sniterate`, `withsome`, and `withall` provide for lack-of-knowledge acting. The use of conditional plans, as discussed in § 6.4 , has allowed a GLAIR agent to use contextual information to select among alternative mathematical procedures to perform (Shapiro et al. 2007).

By including in the Knowledge Layer a term that refers to the agent itself, GLAIR agents are able to represent and reason about themselves. As mentioned in § 9.2, a deictic register in the PML is a pointer to the self-concept. PMLa implementations of primitive acts can insert beliefs into the KL about what the agent is currently doing, and the movement of time, as discussed in § 9.5, gives the agent an episodic memory.

Giving GLAIR agents knowledge of the actions they are currently performing above the level of primitive actions is a subject of further work.

Declarative vs. Procedural representations The Knowledge Layer contains declarative representations of knowledge, even of procedures for carrying out defined acts (*see* § 6.2). The PMLa contains implementations of primitive acts in a way that is not cognitively penetrable. We have not yet experimented with GLAIR agents that learn such procedural representations of primitive acts.

Semantic memory vs. Episodic memory The Knowledge Layer is the locus of both semantic and episodic memory. Most of the beliefs of GLAIR agents we have developed so far are parts of semantic memory. As mentioned above, PMLa implementations of primitive acts can insert beliefs into the KL about what the agent is currently doing, and the movement of time, as discussed in § 9.5, gives the agent an episodic memory.

11.2 Organization of knowledge

Flat vs. Structured/Hierarchical organization of knowledge SNePS uses an inherently structured organization of knowledge. Its term-based predicate logic representation allows for nested functional terms, including proposition-valued terms, the act-valued terms that constitute composite acts, and reasonng rules. SNePS has often been used to represent hierarchical information, including subsumption hierarchies, parthood and other mereological relations, and similar information used in ontological reasoning.

Short-term vs. long-term memory GLAIR currently has no short-term memory from which some memories migrate into long-term memory. The closest thing to a short-term or working memory is the active connection graph (*see* § refsec:acg), which contains the demons currently working on one problem, which are discarded when the agent changes to another problem.

12. Evaluation criteria for cognitive architectures

In this section, we evaluate GLAIR according to criteria listed in (Langley, Laird, and Rogers 2009).

12.1 Generality, versatility, and taskability

Generality The KL and PMLa layers are independent of the implementation of the lower body and the environment as long as there is some way for the primitive sensory and effector acts at the PMLa layer to be implemented in the SAL layer. The agent designer designs the PMLb and PMLc layers to effect the connection. GLAIR agents have been active in: a real world laboratory setting (Shapiro 1998); a virtual reality world (Shapiro et al. 2005c); a world simulated by ASCII input/output (Kandefer and Shapiro 2007); and graphically simulated worlds (Shapiro and Kandefer 2005; Anstey et al. in press).

Versatility The GLAIR architecture lends itself to modular design for new environments and tasks. If the designers have a specific agent body and environment in mind, they must identify the afferent and efferent behavior repertoire of the agent. They can then specify the actions to be implemented at the PMLa layer. These become the primitive actions at the KL layer, and high-level actions can be programmed using the acting model described in § 6.
Since the control actions, which include `snsequence`, `snif`, and `sniterate`, form a Turing-complete set, a GLAIR agent can perform any task that can be composed computationally from its primitive acts (Böhm and Jacopini 1966).

Once the KL primitive actions have been designed, it is common to test and further develop the agent in a simulated environment before moving it to a hardware robot in the real world, or to a more detailed simulation in a graphical or virtual world.

Taskability One benefit of representing acts in the same formalism as other declarative knowledge is that agents that

communicate with a GLAIR agent can command it to perform tasks using the same communication language. The formal language commonly used is SNePSLOG, the language in which the acting model was explained in § 6., but GLAIR agents have been built that use fragments of English (Shapiro 1989; Shapiro, Ismail, and Santore 2000; Shapiro and Ismail 2003; Kandefer and Shapiro 2007). The meaning of verb phrases are represented in the act structures of the acting model. If English versions of contol acts are included in the fragment of English, GLAIR agents may be given composite acts to perform. For example, Cassie$_F$ (Shapiro 1998) can be told to *"Go to the green robot and then come here and help me."* With appropriate grammatical support, natural language may be used to teach GLAIR agents new tasks. For example, an early GLAIR agent was told, *"IF a block is on a support then a plan to achieve that the support is clear is to pick up the block and then put the block on the table"* (Shapiro 1989; Shapiro, Ismail, and Santore 2000).

12.2 Rationality and optimality

Rationality When attempting to achieve a goal ϕ, a GLAIR agent chooses an act α to perform based on its belief that the act will achieve the goal, as expressed by `GoalPlan(`ϕ, α`)`. However, we have not yet experimented with GLAIR agents that formulate such beliefs by reasoning about the effects of various acts. That is, we have not yet developed GLAIR agents that do traditional AI planning. Nor have we experimented with GLAIR agents that formulate `GoalPlan` beliefs after acting and noting the effects of its acts. So a GLAIR agent is rational in the sense that it selects an act that it *believes* will lead to the goal. However, it doesn't *know* that the act will lead to the goal.

Optimality The GLAIR architecture allows for, but does not explicitly implement as part of the architecture, agents that choose optimal actions based on preferences they form. This has been used to implement agents that can prefer certain shortcut ways of performing arithmetical operations (Goldfain 2008), and by metacognitive agents such as those described in (Shapiro et al. 2007), which are able to observe themselves at work and prefer efficient (requiring fewer steps than the alternatives) ways of accomplishing a goal.

12.3 Efficiency and scalability

SNePS does not place any formal restriction on the number of terms that can be represented and stored, nor on the number of relations between them. There is a naturally-occurring limit that depends on the computational resources available to the system and will vary from one machine to the next. The upper limit for any instance of SNePS depends on the heap size of the Lisp image in which it is running. We have not evaluated SNePS in terms of formal computational complexity, A recent technical report on SNePS' efficiency (Seyed, Kandefer, and Shapiro 2008) shows that the system can reason over knowledge bases that include tens of thousands of terms/propositions, though some reasoning tasks take many seconds to complete in this situation. The same report outlines steps to increase the number of supported memory elements and the speed with which they are processed by the system. Some of these planned modifications have already been implemented in the latest releases. Other proposed changes include introducing a sophisticated scheme for moving to long-term memory information in the KB that is not being used in the service of reasoning at that time and is not likely to be used soon. SNePS3 (currently under development) will introduce even more efficiency gains.

12.4 Reactivity and persistence

GLAIR's use of separate buffers for separate perceptual modalities facilitates reactivity by ensuring that sensory data from one modality does not block and demand all of an agent's "attention." In some of our work with GLAIR-based agent-actors for virtual drama(Shapiro et al. 2005a), agents interact with human audience participants in a 3D virtual world that allows the human a great deal of freedom to move around in, and effect changes within, the world. In one case, the agent's task is to guide the participant on a quest and complete a series of activities. The participant's actions cannot be fully anticipated, and may include verbally addressing the agent, losing interest and wandering off, making unrestricted movements unrelated to the task, etc. The agent's task then consists of following and keeping up with the participant and reacting as appropriately as is possible to her actions while simultaneously trying to convince her to participate in the assigned task. This requires an implementation of persistence in which the agent keeps track of goals for the current task (and the greater quest), while simultaneously dealing with unpredictable changes in the environment due to the participant's activities.

12.5 Improvability

Improvability GLAIR includes several forms of learning:

Learning by being told: Propositions and policies added to the KL, whether from a human, another agent, or via perception are immediately available for use. For example, a GLAIR agent is instructable. If it is unable, due to lack of knowledge, to perform some act, a human may instruct it so that the agent will be able to perform that act in the future.

Contextual learning: As discussed in §3.2, when a proposition ϕ is derived in a context $\mathcal{C}$, its origin set o, a set of hypotheses, is stored with it. As the agent performs, the context will probably change and some of the hypotheses in o be removed from the context. When a new context arises that again contains all the hypotheses in o, ϕ will again be asserted without having to be rederived. Consider a conditional plan such as ϕ `=> GoalPlan(`α, p`)`. The first time the plan is considered in an appropriate context, ϕ and then `GoalPlan(`α, p`)` will have to be derived. If another situation arises in which the hypotheses in o are asserted, `GoalPlan(`α, p`)` will be asserted without the need for rederivation.

Experience-Based Deductive Learning Consider the general definition of transitivity, expressible in SNePSLOG

as

```
all(r)(Transitive(r)
       => all(x,y,z)({r(x,y), r(y,z)}
                      &=> r(x,z)))
```

along with the belief that Transitive(ancestor). The first time an ancestor question is posed to Cassie, she will use the transitivity definition to derive

```
all(x,y,z)({ancestor(x,y),
            ancestor(y,z)}
           &=> ancestor(x,z)))
```

and then answer the question. The specific ancestor rule will be stored in the Knowledge Layer. The next time an ancestor question is posed, Cassie will use the specific ancestor rule, but not the general transitivity definition. Even though the knowledge base is now larger (two rules are stored instead of one), the second question will be answered more quickly than if the first question hadn't been asked. Cassie has developed a kind of expertise in ancestor-reasoning (Choi and Shapiro 1991; Choi 1993).

Several other forms of improvability have not yet been added to GLAIR. For example, we have not yet experimented with agents that use the observed effects of its acts to modify or extend its plans. Nor have we yet experimented with agents that "compile" defined acts into primitive acts.

12.6 Autonomy and extended operation

The GLAIR acting system allows for agents that act autonomously for long periods of time, though we have not made any formal measure of agents' degrees of autonomy. Many of our agents can act indefinitely independent of any explicit instructions from, or interactions with, an operator by pursuing goals, following plans, and responding to changes in the environment as they are perceived.

13. Future Concerns

Several issues that are certainly important for cognitive architectures have not yet been addressed in the development of GLAIR. These include uncertainty and considerations of real-time operation to limit the amount of reasoning.

14. Current Status

SNePS has been under development, with numerous uses, modifications, additions, and reimplementations since before 1979 (Shapiro 1979). Likewise, GLAIR has been under development since before 1993 (Hexmoor, Lammens, and Shapiro 1993), and has been used for a variety of agents, for some examples see (Shapiro and Kandefer 2005; Kandefer and Shapiro 2007; Anstey et al. in press). MGLAIR is still being defined, although prototype versions have been used to build a variety of agents (Shapiro et al. 2005a).

References

Alchourrón, C. E.; Gärdenfors, P.; and Makinson, D. 1985. On the logic of theory change: Partial meet contraction and revision functions. *The Journal of Symbolic Logic* 50(2):510–530.

Anderson, J. R., and Lebiere, C. 1998. *The Atomic Components of Thought*. Mahwah, NJ: Lawrence Erlbaum.

Anstey, J.; Seyed, A. P.; Bay-Cheng, S.; Bona, J.; Hibit, S.; Pape, D.; Shapiro, S. C.; and Sena, V. in press. The agent takes the stage. *International Journal of Arts and Technology,*.

Böhm, C., and Jacopini, G. 1966. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM* 9(5):366–371.

Carbonell, J. G.; Knoblock, C. A.; and Minton, S. 1990. PRODIGY: An integrated architecture for planning and learning. In VanLehn, K., ed., *Architectures for Intelligence*. Hillsdale, NJ: Lawrence Erlbaum. 241–278.

Choi, J., and Shapiro, S. C. 1991. Experience-based deductive learning. In *Third International Conference on Tools for Artificial Intelligence TAI '91*. Los Alamitos, CA: IEEE Computer Society Press. 502–503.

Choi, J. 1993. *Experience-Based Learning in Deductive Reasoning Systems*. PhD dissertation, Technical Report 93-20, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY.

de Kleer, J. 1986. An assumption-based truth maintenance system. *Artificial Intelligence* 28(2):127–162.

Duchan, J. F.; Bruder, G. A.; and Hewitt, L. E., eds. 1995. *Deixis in Narrative: A Cognitive Science Perspective*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Goldfain, A. 2008. *A Computational Theory of Early Mathematical Cognition*. Ph.D. Dissertation, State University of New York at Buffalo, Buffalo, NY.

Haller, S. 1996. Planning text about plans interactively. *International Journal of Expert Systems* 9(1):85–112.

Haller, S. 1999. An introduction to interactive discourse processing from the perspective of plan recognition and text planning. *Artificial Intelligence Review* 13(4):259–333.

Hexmoor, H.; Lammens, J.; and Shapiro, S. C. 1993. Embodiment in GLAIR: a grounded layered architecture with integrated reasoning for autonomous agents. In Dankel II, D. D., and Stewman, J., eds., *Proceedings of The Sixth Florida AI Research Symposium (FLAIRS 93)*. The Florida AI Research Society. 325–329.

Hobbs, J. R. 2000. Half orders of magnitude. In Obrst, L., and Mani, I., eds., *Papers from the Workshop on Semantic Approximation, Granularity, and Vagueness*, 28–38. A Workshop of the Seventh International Conference on Principles of Knowldege Representation and Reasoning, Breckenridge, CO.

Ismail, H. O., and Shapiro, S. C. 2000. Two problems with reasoning and acting in time. In Cohn, A. G.; Giunchiglia, F.; and Selman, B., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR 2000)*, 355–365. San Francisco: Morgan Kaufmann.

Ismail, H. O., and Shapiro, S. C. 2001. The cognitive clock: A formal investigation of the epistemology of time. Technical Report 2001-08, Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY.

Ismail, H. O. 2001. *Reasoning and Acting in Time*. PhD dissertation, Technical Report 2001-11, University at Buffalo, The State University of New York, Buffalo, NY.

Kandefer, M., and Shapiro, S. C. 2007. Knowledge acquisition by an intelligent acting agent. In Amir, E.; Lifschitz, V.; and Miller, R., eds., *Logical Formalizations of Commonsense Reasoning, Papers from the AAAI Spring Symposium, Technical Report SS-07-05*, 77–82. Menlo Park, CA: AAAI Press.

Kumar, D., and Shapiro, S. C. 1991. Architecture of an intelligent agent in SNePS. *SIGART Bulletin* 2(4):89–92.

Kumar, D., and Shapiro, S. C. 1994. Acting in service of inference (and *vice versa*). In Dankel II, D. D., ed., *Proceedings of The Seventh Florida AI Research Symposium (FLAIRS 94)*. The Florida AI Research Society. 207–211.

Kumar, D. 1993. A unified model of acting and inference. In *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*. Los Alamitos, CA: IEEE Computer Society Press.

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

Lammens, J. M.; Hexmoor, H. H.; and Shapiro, S. C. 1995. Of elephants and men. In Steels, L., ed., *The Biology and Technology of Intelligent Autonomous Agents*. Berlin: Springer-Verlag, Berlin. 312–344.

Langley, P.; Cummings, K.; and Shapiro, D. 2004. Hierarchical skills and cognitive architectures. In *Proceedings of the Twenty-Sixth Annual Conference of the Cognitive Science Society*, 779–784.

Langley, P.; Laird, J. E.; and Rogers, S. 2009. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research* 10(2):141–160.

Lehmann, F., ed. 1992. *Semantic Networks in Artificial Intelligence*. Oxford: Pergamon Press.

Martins, J. P., and Shapiro, S. C. 1983. Reasoning in multiple belief spaces. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann. 370–373.

Martins, J. P., and Shapiro, S. C. 1988. A model for belief revision. *Artificial Intelligence* 35:25–79.

McKay, D. P., and Shapiro, S. C. 1980. MULTI — a LISP based multiprocessing system. In *Proceedings of the 1980 LISP Conference*, 29–37.

McKay, D. P., and Shapiro, S. C. 1981. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann. 368–374.

Morgado, E. J. M., and Shapiro, S. C. 1985. Believing and acting: A study of meta-knowledge and meta-reasoning. In *Proceedings of EPIA-85 "Encontro Portugues de Inteligencia Artificial"*, 138–154.

Neal, J. G., and Shapiro, S. C. 1985. Parsing as a form of inference in a multiprocessing environment. In *Proceedings of the Conference on Intelligent Systems and Machines*, 19–24. Rochester, Michigan: Oakland University.

Neal, J. G., and Shapiro, S. C. 1987a. Knowledge-based parsing. In Bolc, L., ed., *Natural Language Parsing Systems*. Berlin: Springer-Verlag. 49–92.

Neal, J. G., and Shapiro, S. C. 1987b. Knowledge representation for reasoning about language. In Boudreaux, J. C.; Hamill, B. W.; and Jernigan, R., eds., *The Role of Language in Problem Solving 2*. Elsevier Science Publishers. 27–46.

Orilia, F., and Rapaport, W. J., eds. 1998. *Thought, Language, and Ontology: Essays in Memory of Hector-Neri Castañeda*. Dordrecht: Kluwer Academic Publishers.

Rapaport, W. J.; Shapiro, S. C.; and Wiebe, J. M. 1997. Quasi-indexicals and knowledge reports. *Cognitive Science* 21(1):63–107. Reprinted in (Orilia and Rapaport 1998, pp. 235–294).

Searle, J. R. 1975. Indirect speech acts. In Cole, P., and Morgan, J. L., eds., *Speech Acts: Syntax and Semantics*, volume 3. Academic Press. 59–82.

Seyed, A. P.; Kandefer, M.; and Shapiro, S. C. 2008. Sneps efficiency report. SNeRG Technical Note 43, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY.

Shapiro, S. C., and Ismail, H. O. 2003. Anchoring in a grounded layered architecture with integrated reasoning. *Robotics and Autonomous Systems* 43(2–3):97–108.

Shapiro, S. C., and Kandefer, M. 2005. A SNePS approach to the wumpus world agent or Cassie meets the wumpus. In Morgenstern, L., and Pagnucco, M., eds., *IJCAI-05 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'05): Working Notes*. Edinburgh, Scotland: IJCAII. 96–103.

Shapiro, S. C., and Neal, J. G. 1982. A knowledge engineering approach to natural language understanding. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*. Menlo Park, CA: ACL. 136–144.

Shapiro, S. C., and Rapaport, W. J. 1987. SNePS considered as a fully intensional propositional semantic network. In Cercone, N., and McCalla, G., eds., *The Knowledge Frontier*. New York: Springer-Verlag. 263–315.

Shapiro, S. C., and Rapaport, W. J. 1992. The SNePS family. *Computers & Mathematics with Applications* 23(2–5):243–275. Reprinted in (Lehmann 1992, pp. 243–275).

Shapiro, S. C., and Rapaport, W. J. 1995. An introduction to a computational reader of narratives. In Duchan, J. F.; Bruder, G. A.; and Hewitt, L. E., eds., *Deixis in Narrative: A Cognitive Science Perspective*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc. 79–105.

Shapiro, S. C., and The SNePS Implementation Group. 2008. *SNePS 2.7 User's Manual*. Department of Com-

puter Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY.

Shapiro, S. C.; Anstey, J.; Pape, D. E.; Nayak, T. D.; Kandefer, M.; and Telhan, O. 2005a. MGLAIR agents in a virtual reality drama. Technical Report 2005-08, Department of Computer Science & Engineering, University at Buffalo, Buffalo, NY.

Shapiro, S. C.; Anstey, J.; Pape, D. E.; Nayak, T. D.; Kandefer, M.; and Telhan, O. 2005b. MGLAIR agents in virtual and other graphical environments. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*. Menlo Park, CA: AAAI Press. 1704–1705.

Shapiro, S. C.; Anstey, J.; Pape, D. E.; Nayak, T. D.; Kandefer, M.; and Telhan, O. 2005c. The Trial The Trail, Act 3: a virtual reality drama using intelligent agents. In Young, R. M., and Laird, J., eds., *Proceedings of the First Annual Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, 157–158. Menlo Park, CA: AAAI Press.

Shapiro, S. C.; Rapaport, W. J.; Kandefer, M.; Johnson, F. L.; and Goldfain, A. 2007. Metacognition in SNePS. *AI Magazine* 28:17–31.

Shapiro, S. C.; Ismail, H. O.; and Santore, J. F. 2000. Our dinner with Cassie. In *Working Notes for the AAAI 2000 Spring Symposium on Natural Dialogues with Practical Robotic Devices*, 57–61. Menlo Park, CA: AAAI.

Shapiro, S. C.; Martins, J. P.; and McKay, D. P. 1982. Bidirectional inference. In *Proceedings of the Fourth Annual Meeting of the Cognitive Science Society*, 90–93.

Shapiro, S. C. 1978. Path-based and node-based inference in semantic networks. In Waltz, D. L., ed., *Tinlap-2: Theoretical Issues in Natural Languages Processing*. New York: ACM. 219–225.

Shapiro, S. C. 1979. The SNePS semantic network processing system. In Findler, N. V., ed., *Associative Networks: The Representation and Use of Knowledge by Computers*. New York: Academic Press. 179–203.

Shapiro, S. C. 1982. Generalized augmented transition network grammars for generation from semantic networks. *The American Journal of Computational Linguistics* 8(1):12–25.

Shapiro, S. C. 1986. Symmetric relations, intensional individuals, and variable binding. *Proceedings of the IEEE* 74(10):1354–1363.

Shapiro, S. C. 1987. Processing, bottom-up and top-down. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*. New York: John Wiley & Sons. 779–785. Reprinted in Second Edition, 1992, pages 1229–1234.

Shapiro, S. C. 1989. The CASSIE projects: An approach to natural language competence. In Martins, J. P., and Morgado, E. M., eds., *EPIA 89: 4th Portugese Conference on Artificial Intelligence Proceedings, Lecture Notes in Artificial Intelligence 390*. Berlin: Springer-Verlag. 362–380.

Shapiro, S. C. 1991. Cables, paths and "subconscious" reasoning in propositional semantic networks. In Sowa, J., ed., *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Los Altos, CA: Morgan Kaufmann. 137–156.

Shapiro, S. C. 1992. Artificial intelligence. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*. New York: John Wiley & Sons, second edition. 54–57.

Shapiro, S. C. 1993. Belief spaces as sets of propositions. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)* 5(2&3):225–235.

Shapiro, S. C. 1998. Embodied Cassie. In *Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium, Technical Report FS-98-02*. Menlo Park, California: AAAI Press. 136–143.

Shapiro, S. C. 2000a. An introduction to SNePS 3. In Ganter, B., and Mineau, G. W., eds., *Conceptual Structures: Logical, Linguistic, and Computational Issues*, volume 1867 of *Lecture Notes in Artificial Intelligence*. Berlin: Springer-Verlag. 510–524.

Shapiro, S. C. 2000b. SNePS: A logic for natural language understanding and commonsense reasoning. In *Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language*. Menlo Park, CA: AAAI Press/The MIT Press. 175–195.

Shubin, H. 1981. Inference and control in multiprocessing environments. Technical Report 186, Department of Computer Science, SUNY at Buffalo.

Research with Collaborative Unmanned Aircraft Systems

P. Doherty J. Kvarnström F. Heintz D. Landen
P-M. Olsson
Department of Computer and Information Science
Linköping University, SE-58183 Linköping, Sweden
{patdo,jonkv,frehe,davla,perol}@ida.liu.se

Abstract

We provide an overview of ongoing research which targets development of a principled framework for mixed-initiative interaction with unmanned aircraft systems (UAS). UASs are now becoming technologically mature enough to be integrated into civil society. Principled interaction between UASs and human resources is an essential component in their future uses in complex emergency services or bluelight scenarios. In our current research, we have targeted a triad of fundamental, interdependent conceptual issues: delegation, mixed-initiative interaction and adjustable autonomy, that is being used as a basis for developing a principled and well-defined framework for interaction. This can be used to clarify, validate and verify different types of interaction between human operators and UAS systems both theoretically and practically in UAS experimentation with our deployed platforms.

1 Introduction

In the past decade, the Unmanned Aircraft Systems Technologies Lab[1] at the Department of Computer and Information Science, Linköping University, has been involved in the development of autonomous unmanned aerial vehicles and associated hardware and software technologies [13, 11, 12]. The size of our research platforms range from the RMAX helicopter system [14, 39, 36, 33, 7] (Figure 1) developed by Yamaha Motor Company, to smaller micro-size rotor based systems such as the LinkQuad (Figure 2)[2] and LinkMAV [23, 34] (Figure 1), in addition to a fixed wing platform, the PingWing [8] (Figure 1). The latter three have been designed and developed by the Unmanned Aircraft Systems Technologies Lab. Previous work has focused on the development of robust autonomous systems for UAV's which seamlessly integrate control, reactive and deliberative capabilities that meet the requirements of hard and soft realtime constraints [14, 32]. Additionally, we have focused on the development and integration of many high-level autonomous capabilities studied in the area of cognitive robotics such as task planners [15, 16], motion planners [37, 36, 38], execution monitors [18], and reasoning systems [19, 17, 31], in addition to novel middleware frameworks which support such integration [27, 29, 30]. Although research with individual high-level cognitive functionalities is quite advanced, robust integration of such capabilities in robotic systems which meet real-world constraints is less developed but essential to introduction of robotic systems into society in the future.

[1] www.ida.liu.se/~patdo/aiicssite1/
[2] www.uastech.com

Consequently, our research has focused, not only on such high-level cognitive functionalities, but also on integrative issues.

Figure 1: The UASTech RMAX (left), LinkMAV (center) and the PingWing (right)

More recently, our research efforts have begun to focus on applications where heterogeneous UASs are required to collaborate not only with each other but also with diverse human resources [20, 21, 28]. UASs are now becoming technologically mature enough to be integrated into civil society. Principled interaction between UASs and human resources is an essential component in the future uses of UASs in complex emergency services or bluelight scenarios. Some specific target UAS scenario examples are search and rescue missions for inhabitants lost in wilderness regions and assistance in guiding them to a safe destination; assistance in search at sea scenarios; assistance in more devastating scenarios such as earthquakes, flooding or forest fires; and environmental monitoring.

As UASs become more autonomous, mixed-initiative interaction between human operators and such systems will be central in mission planning and tasking. By mixed-initiative, we mean that interaction and negotiation between a UAS and a human will take advantage of each of their skills, capacities and knowledge in developing a mission plan, executing the plan and adapting to contingencies during the execution of the plan. In the near future, the practical use and acceptance of UASs will have to be based on a verifiable, principled and well-defined interaction foundation between one or more human operators and one or more autonomous systems. In developing a principled framework for such complex interaction between UASs and humans in complex scenarios, a great many interdependent conceptual and pragmatic issues arise and need clarification both theoretically, but also pragmatically in the form of demonstrators. Additionally, an iterative research methodology is essential which combines foundational theory, systems building and empirical testing in real-world applications from the start.

Figure 2: The UASTech LinkQuad Quadrotor Helicopter

2 A Conceptual Triad

In our current research, we have targeted a triad of fundamental, interdependent conceptual issues: delegation, mixed-initiative interaction and adjustable autonomy. The triad of concepts is being used as a basis for developing a principled and well-defined framework for interaction that can be used to clarify, validate and verify different types of interaction between human operators and UAS systems both theoretically and practically in UAS experimentation with our deployed platforms. The concept of delegation is particularly important and in some sense provides a bridge between mixed-initiative interaction and adjustable autonomy.

Delegation – In any mixed initiative interaction, humans request help from robotic systems and robotic systems may request help from humans. One can abstract and concisely model such requests as a form of delegation, $Delegate(A, B, task, constraints)$, where A is the delegating agent, B is the potential contractor, *task* is the task being delegated and consists of a goal and possibly a plan to achieve the goal, and *constraints* represents a context in which the request is made and the task should be carried out.

Adjustable Autonomy – In solving tasks in a mixed-initiative setting, the robotic system involved will have a potentially wide spectrum of autonomy, yet should only use as much autonomy as is required for a task and should not violate the degree of autonomy mandated by a human operator unless agreement is made. One can begin to develop a principled means of adjusting autonomy through the use of the *task* and *constraint* parameters in the $Delegate(A, B, task, constraints)$ function. A task delegated with only a goal and no plan, with few constraints, allows the robot to use much of its autonomy in solving the task, whereas a task specified as a sequence of actions and many constraints allows only limited autonomy.

Mixed-Initiative Interaction – Mixed-initiative interaction involves a very broad set of issues,

both theoretical and pragmatic. One central part of such interaction is the ability of a a ground operator (GOP) to be able to delegate tasks to a UAS, $Delegate(GOP, UAS, task, constraints)$ and in a symmetric manner, the ability of a UAS to be able to delegate tasks to a GOP, $Delegate(UAS, GOP, task, constraints)$. Issues pertaining to safety, security, trust, etc., have to be dealt with in the interaction process and can be formalized as particular types of constraints associated with a delegated task. Additionally, the task representation must be highly flexible, distributed and dynamic. Tasks need to be delegated at varying levels of abstraction and also expanded and modified as parts of tasks are recursively delegated to different UAS agents. Consequently, the structure must also be distributable.

3 A First Iteration

3.1 The Architecture

Our RMAX helicopters use a CORBA-based distributed architecture [14]. For our experimentation with collaborative UASs, we view this as a legacy system and extend it with what is conceptually an additional outer layer in order to leverage the functionality of JADE [24]. "JADE (Java Agent Development Framework) is a software environment to build agent systems for the management of networked information resources in compliance with the FIPA specifications for interoperable multi-agent systems." [25]. The reason for this is pragmatic. Our formal characterization of the *Delegate()* operator is as a speech act. We also use speech acts as an agent communication language and JADE provides a straightforward means for integrating the FIPA ACL language which supports speech acts with our existing systems. The outer layer may be viewed as a collection of JADE agents that interface to the legacy system. We are currently using four agents in the outer layer:

1. **Interface agent** - This agent is the clearinghouse for communication. All requests for delegation and other types of communication pass through this agent. Externally, it provides the interface to a specific robotic system.
2. **Delegation agent**- The delegation agent coordinates delegation requests to and from other UAS systems, with the executor, Resource and Interface agents. It does this essentially by verifying that the pre-conditions to a $Delegate()$ request are satisfied.
3. **Executor agent** - After a task is contracted to a particular UAS, it must eventually execute that task relative to the constraints associated with it. The Executor agent coordinates this execution process.
4. **Resource agent** - The Resource agent determines whether the UAS of which it is part has the resources and ability to actually do a task as a potential contractor. Such a determination may include the invocation of schedulers, planners and constraint solvers in order to determine this.

3.2 Semantic Perspective: Delegation as a Speech Act

In [4, 26], Falcone & Castelfranchi provide an illuminating, but informal discussion about delegation as a concept from a social perspective. Their approach to delegation builds on a BDI model of agents, that is, agents having beliefs, goals, intentions, and plans [6], but the specification lacks

a formal semantics for the operators used. Based on intuitions from their work, we provided a formal characterization of their concept of strong delegation using a communicative speech act with pre- and post-conditions which update the belief states associated with the delegator and contractor, respectively [21]. In order to formally characterize the operators used in the definition of the speech act, we used KARO [35] to provide a formal semantics. The KARO formalism is an amalgam of dynamic logic and epistemic / doxastic logic, augmented with several additional (modal) operators in order to deal with the motivational aspects of agents.

First, we define the notion of a task as a pair consisting of a goal and a plan for that goal, or rather, a plan and the goal associated with that plan. Paraphrasing Falcone & Castelfranchi into KARO terms, we consider a notion of strong/strict delegation represented by a speech act S-Delegate(A, B, τ) of A delegating a task $\tau = (\alpha, \phi)$ to B, where α is a possible plan and ϕ is a goal. It is specified as follows:

S-Delegate(A, B, τ), where $\tau = (\alpha, \phi)$

Preconditions:

(1) $Goal_A(\phi)$

(2) $Bel_A Can_B(\tau)$ (Note that this implies $Bel_A Bel_B(Can_B(\tau))$)

(3) $Bel_A(Dependent(A, B, \alpha))$

Postconditions:

(1) $Goal_B(\phi)$ and $Bel_B Goal_B(\phi)$

(2) $Committed_B(\alpha)$.

(3) $Bel_B Goal_A(\phi)$

(4) $Can_B(\tau)$ (and hence $Bel_B Can_B(\tau)$, and by (1) also $Intend_B(\tau)$)

(5) $MutualBel_{AB}$(“the statements above” $\wedge$ $SociallyCommitted(B, A, \tau)$)

Informally speaking this expresses the following: the preconditions of the S-delegation act of A delegating task τ to B are that (1) ϕ is a goal of delegator A (2) A believes that B can (is able to) perform the task τ (which implies that A believes that B himself believes that he can do the task!) (3) A believes that with respect to the task τ he is dependent on B.

The postconditions of the delegation act mean: (1) B has ϕ as his goal and is aware of this (2) he is committed to the task (3) B believes that A has the goal ϕ (4) B can do the task τ (and hence believes it can do it, and furthermore it holds that B intends to do the task, which was a separate condition in F&C’s set-up), and (5) there is a mutual belief between A and B that all preconditions and other postconditions mentioned hold, as well as that there is a contract between A and B, i.e. B is socially committed to A to achieve τ for A. In this situation we will call agent A the delegator and B the contractor.

Typically a social commitment (contract) between two agents induces obligations to the partners involved, depending on how the task is specified in the delegation action. This dimension has to be added in order to consider how the contract affects the autonomy of the agents, in particular the contractor’s autonomy. We consider a few relevant forms of delegation specification below.

3.2.1 Closed vs Open delegation

Falcone & Castelfranchi furthermore discuss the following variants of task specification:

- closed delegation: the task is completely specified: both goal and plan should be adhered to.
- open delegation: the task is not completely specified: either only the goal has to be adhered to while the plan may be chosen by the contractor, or the specified plan contains 'abstract' actions that need further elaboration (a 'sub-plan') to be dealt with by the contractor.

So in open delegation the contractor may have some freedom to perform the delegated task, and thus it provides a large degree of flexibility in multi-agent planning, and allows for truly distributed planning.

The specification of the delegation act in the previous subsection was in fact based on closed delegation. In case of open delegation α in the postconditions can be replaced by an α', and τ by $\tau' = (\alpha', \phi)$. Note that the fourth clause, viz. $Can_B(\tau')$, now implies that α' is indeed believed to be an alternative for achieving ϕ, since it implies that $Bel_B[\alpha']\phi$ (B believes that ϕ is true after α' is executed). Of course, in the delegation process, A must agree that α' is indeed viable. This would depend on what degree of autonomy is allowed.

This particular specification of delegation follows Falcone & Castelfranchi closely. One can easily foresee other constraints one might add or relax in respect to the basic specification resulting in other variants of delegation [5, 10].

3.2.2 Strong Delegation in Agent Programming

When devising a system like the one we have in mind for our scenario, we need programming concepts that support delegation and in particular the open variant of delegation. In the setting of an agent programming language such as 2APL [9], we may use plan generation rules to establish a contract between two agents. Very briefly, a 2APL agent has a belief base, a goal base, a plan base, a set of capabilities (basic actions it can perform), and sets of rules to change its bases: PG rules, PR rules and PC rules. PG-rules have the form $\gamma \leftarrow \beta \mid \pi$, meaning that if the agent has goal γ and belief β then it may generate plan π and put it in its plan base. PR rules can be used to repair plans if execution of the plan fails: they are of the form $\pi \leftarrow \beta \mid \pi'$, meaning that if π is the current plan (which is failing), and the agent believes β then it may revise π into π'. PC-rules are rules for defining macros and recursive computations. (We will not specify them here.)

The act of strong delegation can now be programmed in 2APL by providing the delegator with a rule

$$\phi \leftarrow Can_B(\tau) \wedge Dependent(A, B, \tau) \mid SDelegate(A, B, \tau)$$

(where $\tau = (\alpha, \phi)$), which means that the delegation act may be generated by delegator A exactly when the preconditions that we described earlier are met. The action $SDelegate(A, B, \tau)$ is a communication action requesting to adapt the goal and belief bases of B according to the KARO specification given earlier, and should thus, when successful (depending upon additional assumptions such as that there is some authority or trust relation between A and B), result in a state where contractor B has ϕ in its goal base, $Goal_A(\phi)$, $Can_B(\tau)$ and $MB(\text{‘}contract')$ in its belief base, and plan α in its plan base. That is to say, in the case of a closed delegation specification. If the specification is an open delegation, it instead will have an alternative plan

α' in its plan base and a belief $Can_B(\alpha', \phi)$ in its belief base. It is very important to note that in the case of such a concrete setting of an agent programmed in a language such as 2APL, we may provide the Can-predicate with a more concrete interpretation: $Can_B(\alpha, \phi)$ is true if (either ϕ is in its goal base and α is in its plan base already, or) B has a PG-rule of the form $\phi \leftarrow \beta \mid \alpha'$ for some β that follows from B's belief base, and the agent has the resources available for executing plan α. This would be a concrete interpretation of the condition that the agent has the ability as well as the opportunity to execute the plan!

3.3 Pragmatic Perspective: Delegation as Contract Networks and Constraints

From a semantic perspective, delegation as a speech act provides us with insight and an abstract specification which can be used as a basis for a more pragmatic implementation on actual UAS platforms. There is a large gap between these perspectives though. We have chosen to also work from a bottom-up perspective and have developed a prototype software system that implements a delegation framework using the JADE architecture specified above. The system has been tested using a number of complex collaborative scenarios described later in the paper.

In the software architecture, we have focused on a number of issues central to making such a system work in practice:

- **Task Specification** – A specification for a task representation which we call *task specification trees*. This representation has to be implicitly sharable, dynamically extendable, and distributed in nature. Such a task structure is passed from one agent to another and possibly extended in more detail as delegation process is invoked recursively among agents and humans. If the delegation process is successful, the resulting shared structure is in fact executable in a distributed manner. The delegation agents associated with specific UASs are responsible for passing and extending such structures in order to meet the requirements of goal specifications or instantiations of abstract task specifications. The Executor agents associated with specific UASs have the capacity to execute specific nodes in a shared task specification tree that have been delegated to them.
- **Ability and Resource Allocation** – A central precondition to the Delegate speech act specified previously is whether a potential contracting agent *can* do a task. This involves a UAS determining whether it has the proper resources both statically (sensors) and dynamically (use of sensors, power, fuel), and whether it can schedule execution of the processes required to achieve a task at a particular time. In essence, a pragmatic grounding of the $Can()$ predicate in the architecture is required. We are approaching this problem through the use of a combination of distributed constraint solving and loosely coupled distributed task planning. The resource agents associated with specific UASs are responsible for reasoning about resources and solving constraint problems when queried by the associated UAS's delegation agent.
- **Collaborative Planning** – UASs which have accepted a delegated task are responsible for insuring that they can put together a team of UASs which can consistently contribute to the solution of the task. This involves recursive calls to the delegation process, local generation of sub-plans which achieve specific aspects of a task and the integration of these sub-plans into a coherent global plan which can be executed consistently and in a distributed manner to achieve the task requirements. For this we have been developing extensions to TALplanner which combine forward chaining with partial-order planning.

- **Delegation process** - The delegation process itself involves the use of speech acts and contract networks in combination. The process of delegation is quite complex in that a task specification tree has to be constructed dynamically in time and then executed in a distributed manner while meeting all the constraints specified in recursive delegation calls. This also has to be done in a tractable manner in order to ensure that temporal and spatial constraints are met.

These topics are work in progress and will be presented in more detail in future work. A prototype implementation does exist and is being used for experimentation with a number of complex collaborative UAS scenarios briefly described in the next section.

4 Collaborative Scenarios

We have chosen two relatively complex collaborative UAS scenarios in which to develop our mixed-initiative framework.

4.1 An Emergency Services Scenario with Logistics

On December 26, 2004, a devastating earthquake of high magnitude occurred off the west coast off Sumatra. This resulted in a tsunami which hit the coasts of India, Sri Lanka, Thailand, Indonesia and many other islands. Both the earthquake and the tsunami caused great devastation. During the initial stages of the catastrophe, there was a great deal of confusion and chaos in setting into motion rescue operations in such wide geographic areas. The problem was exacerbated by shortage of manpower, supplies and machinery. The highest priorities in the initial stages of the disaster were searching for survivors in many isolated areas where road systems had become inaccessible and providing relief in the form of delivery of food, water and medical supplies. Similar real-life scenarios have occurred more recently in China and Haiti where devastating earthquakes have caused tremendous material and human damage.

Let's assume for a particular geographic area, one had a shortage of trained helicopter and fixed-wing pilots and/or a shortage of helicopters and other aircraft. Let's also assume that one did have access to a fleet of autonomous unmanned helicopter systems with ground operation facilities. How could such a resource be used in the real-life scenario described?

Leg I In the first part of the scenario, it is essential that for specific geographic areas, the UAS platforms should cooperatively scan large regions in an attempt to identify injured persons. The result of such a cooperative scan would be a saliency map pinpointing potential victims, their geographical coordinates and sensory output such as high resolution photos and thermal images of potential victims. The resulting saliency map would be generated as the output of such a cooperative UAS mission and could be used directly by emergency services or passed on to other UASs as a basis for additional tasks.

Leg II In the second part of the scenario, the saliency map generated in Leg I would be used as a basis for generating a logistics plan for several of the UASs with the appropriate capabilities to deliver food, water and medical supplies to the injured identified in Leg I. This of course would also be done in a cooperative manner among the platforms.

Leg I of this mission has been flown using two RMAX helicopters flying autonomously and using the prototype software system described above. The output of the mission is a saliency map

Figure 3: Identified bodies from Leg I of the emergency services scenario

Figure 4: Emergency Supply Delivery

with geo-located injured humans and infrared and digital photos of the injured (Figure 3). Leg II of this mission has been tested in hardware in the loop simulation (Figure 4). Initial work with this scenario is reported in [21, 22, 33].

4.2 A UAS Communication Relay Scenario

A wide variety of applications of UASs include the need for surveillance of distant targets, including search and rescue operations, traffic surveillance and forest fire monitoring as well as law enforcement and military applications. In many cases, the information gathered by a surveillance UAS must be transmitted in real time to a base station where the current operation is being coordinated. This information often includes live video feeds, where it is essential to achieve uninterrupted communication with high bandwidth. UAS applications may therefore require line-of-sight communications to minimize quality degradation, which is problematic in urban or mountainous areas. Even given free line of sight, bandwidth requirements will also place strict restrictions on the maximum achievable communication range. These limitations are particularly important when smaller UASs are used, such as the 500-gram battery-powered LinkMAV (Figure 1), or the LinkQuad (Figure 2). In these situations, transmitting information directly to the base station can be difficult or impossible.

Both intervening obstacles and limited range can be handled using a chain of intermediate *relay UASs* passing on information from the surveillance UAV to the base station (Figure 5). A surveillance UAV can then be placed freely in a location that yields information of high quality. We are therefore interested in positioning relay UASs to maximize the quality of the resulting chain, given a known target position. However, we are also interested in minimizing resource

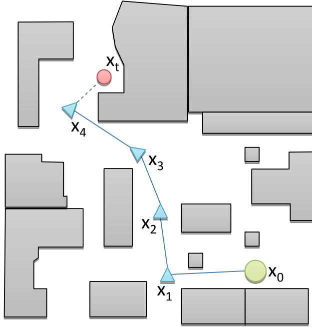


Figure 5: UAVs at x_1, x_2 and x_3 are acting as relays in a city, connecting the base station at x_0 with the surveillance UAV at x_4, surveilling the target at x_t.

usage in terms of the number of relay UAVs required. Given these two objectives, a variety of trade-offs are possible. For example, decreasing the distance between adjacent relay UASs may improve transmission quality but instead requires additional relays.

In [1, 3, 2], we have developed a number of graph search algorithms which are scalable and efficient approximations to continuous bi-objective optimization problems and applicable to relay positioning in discrete space. Using these algorithms as a basis, we are experimenting in simulation with multiple platforms which combine the use of such algorithms with our collaborative framework. A human operator or a contracted UAS will set up such a relay and delegate subparts of the relay mission to other UASs in the area. Currently, we are experimenting with hardware-in-the-loop simulations. An example of one the environments used in testing is an urban environment with semi-random placement of 100 tall buildings, as shown in Figure 6. To reduce clutter, the figure is based on a sparse discretization and shows only the "lowest" level of grid cells. Figure 7 shows a ground operation interface used to generate UAS communication relays.

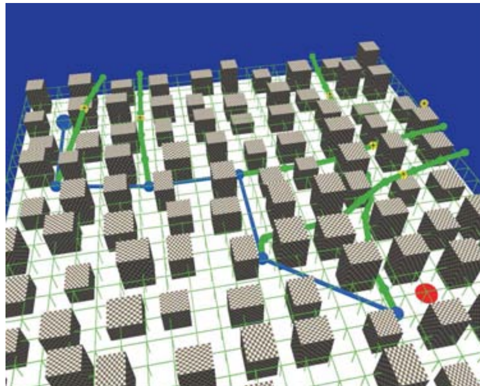


Figure 6: Randomized urban environment.

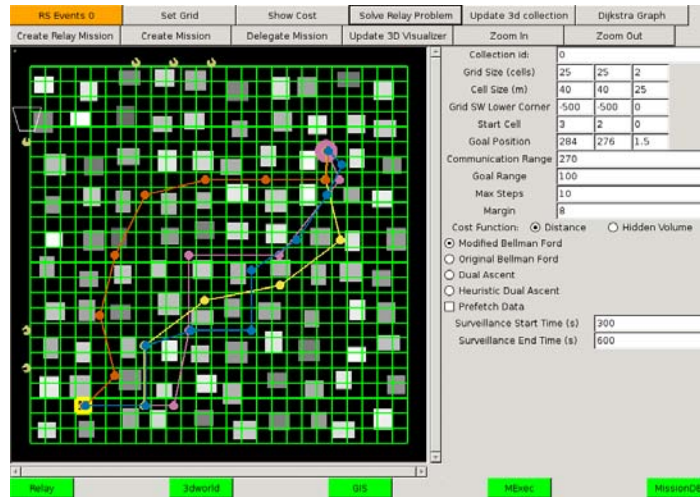


Figure 7: Ground operator interface for generating UAS communication relays

5 Conclusions

The gap between research done in cognitive robotics and pragmatic use of such results in real-life deployed systems embedded in highly dynamic outdoor environments is currently quite large. The research pursued and described in this paper is intended to take steps toward closing this gap by developing theories in a more traditional manner from the top-down as exemplified by the formal characterization of delegation as a speech act, but also by building demonstrators and prototype software systems which deal with all the complexities associated with systems and architectures constrained to operate in dynamic outdoor environments. This type of research demands an iterative and open-ended approach to the problem by combining theory, engineering and application in suitable doses, and continually trying to close the loop at early stages in the research. We will continue to pursue this approach and hope to report additional details in the near future.

Acknowledgements

This overview of research was written specifically for the 2010 Dagstuhl Workshop on Cognitive Robotics, Feb 22-26, 2010. It is intended to summarize work in progress which is jointly supported by grants from the Swedish Research Council (VR), the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, the Swedish Research Council (VR) Linnaeus Center CADICS, ELLIIT Excellence Center at Linköping-Lund for Information Technology and the Center for Industrial Information Technology (CENIIT). The work described in Section 3.2 was done in cooperation with Professor J-J. Ch. Meyer, Utrecht University.

References

[1] O. Burdakov, P. Doherty, K. Holmberg, J. Kvarnström, and P-M. Olsson. Positioning unmanned aerial vehicles as communication relays for surveillance tasks. In *Proceedings of the 5th Robotics: Science and Systems Conference (RSS)*, 2009.

[2] O. Burdakov, P. Doherty, K. Holmberg, J. Kvarnström, and P-M. Olsson. Relay positioning for unmanned aerial vehicle surveillance. *International Journal of Robotics Research*, 2010.

[3] O. Burdakov, P. Doherty, K. Holmberg, and P-M. Olsson. Optimal placement of UV-based communications relay nodes. *Journal of Global Optimization*, 2010.

[4] C. Castelfranchi and R. Falcone. Toward a theory of delegation for agent-based systems. In *Robotics and Autonomous Systems*, volume 24, pages 141–157, 1998.

[5] P. Cohen and H. Levesque. Teamwork. *Nous, Special Issue on Cognitive Science and AI*, 25(4):487–512, 1991.

[6] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[7] G. Conte and P. Doherty. Vision-based unmanned aerial vehicle navigation using geo-referenced information. *EURASIP Journal of Advances in Signal Processing*, 2009.

[8] G. Conte, M. Hempel, P. Rudol, D. Lundström, S. Duranti, M. Wzorek, and P. Doherty. High accuracy ground target geo-location using autonomous micro aerial vehicle platforms. In *Proceedings of the AIAA-08 Guidance,Navigation, and Control Conference*, 2008.

[9] M. Dastani and J.-J. Ch. Meyer. A practical agent programming language. In *Proceedings of AAMAS07 Workshop on Programming Multi-Agent Systems (ProMAS2007)*, pages 72–87, 2007.

[10] E. Davis and L. Morgenstern. A first-order theory of communication and multi-agent plans. *Journal Logic and Computation*, 15(5):701–749, 2005.

[11] P. Doherty. Advanced research with autonomous unmanned aerial vehicles. In *Proceedings on the 9th International Conference on Principles of Knowledge Representation and Reasoning*, 2004. Extended abstract for plenary talk.

[12] P. Doherty. Knowledge representation and unmanned aerial vehicles. In *Proceedings of the IEEE Conference on Intelligent Agent Technolology (IAT 2005)*, 2005.

[13] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund. The WITAS unmanned aerial vehicle project. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 747–755, 2000.

[14] P. Doherty, P. Haslum, F. Heintz, T. Merz, T. Persson, and B. Wingman. A distributed architecture for intelligent unmanned aerial vehicle experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, 2004.

[15] P. Doherty and J. Kvarnström. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

[16] P. Doherty and J. Kvarnström. TALplanner: A temporal logic based planner. *Artificial Intelligence Magazine*, Fall Issue 2001.

[17] P. Doherty and J. Kvarnström. Temporal action logics. In V. Lifschitz, F. van Harmelen, and F. Porter, editors, *The Handbook of Knowledge Representation*, chapter 18, pages 709–757. Elsevier, 2008.

[18] P. Doherty, J. Kvarnström, and F. Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Journal of Automated Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[19] P. Doherty, W. Łukaszewicz, and A. Szałas. Approximative query techniques for agents with heterogenous ontologies and perceptual capabilities. In *Proceedings on the 7th International Conference on Information Fusion*, 2004.

[20] P. Doherty, W. Łukaszewicz, and A. Szałas. Communication between agents with heterogeneous perceptual capabilities. *Journal of Information Fusion*, 8(1):56–69, January 2007.

[21] P. Doherty and J-J. Ch. Meyer. Towards a delegation framework for aerial robotic mission scenarios. In *Proceedings of the 11th International Workshop on Cooperative Information Agents*, 2007.

[22] P. Doherty and P. Rudol. A UAV search and rescue scenario with human body detection and geolocalization. In *20th Australian Joint Conference on Artificial Intelligence (AI07)*, 2007.

[23] S. Duranti, G. Conte, D. Lundström, P. Rudol, M. Wzorek, and P. Doherty. LinkMAV, a prototype rotary wing micro aerial vehicle. In *Proceedings of the 17th IFAC Symposium on Automatic Control in Aerospace*, 2007.

[24] G. Caire F. Bellifemine and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley and Sons, Ltd, 2007.

[25] G. Caire F. Bellifemine, F. Bergenti and A. Poggi. JADE – a Java agent development framework. In J. Dix R. H. Bordini, M. Dastani and A. Seghrouchni, editors, *Multi-Agent Programming - Languages, Platforms and Applications*. Springer, 2005.

[26] R. Falcone and C. Castelfranchi. The human in the loop of a delegated agent: The theory of adjustable social autonomy. *IEEE Transactions on Systems, Man and Cybernetics–Part A: Systems and Humans*, 31(5):406–418, 2001.

[27] F. Heintz and P. Doherty. DyKnow: A knowledge processing middleware framework and its relation to the JDL fusion model. *Journal of Intelligent and Fuzzy Systems*, 17(4), 2006.

[28] F. Heintz and P. Doherty. DyKnow federations: Distributing and merging information among UAVs. In *Eleventh International Conference on Information Fusion (FUSION-08)*, 2008.

[29] F. Heintz, J. Kvarnström, and P. Doherty. A stream-based hierarchical anchoring framework. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[30] F. Heintz, J. Kvarnström, and P. Doherty. Bridging the sense-reasoning gap: DyKnow - stream-based middleware for knowledge processing. *Journal of Advanced Engineering Informatics*, 24(1):14–25, 2010.

[31] M. Magnusson, D. Landen, and P. Doherty. Planning, executing, and monitoring communication in a logic-based multi-agent system. In *18th European Conference on Artificial Intelligence (ECAI 2008)*, 2008.

[32] T. Merz, P. Rudol, and M. Wzorek. Control System Framework for Autonomous Robots Based on Extended State Machines. In *Proceedings of the International Conference on Autonomic and Autonomous Systems*, 2006.

[33] P. Rudol and P. Doherty. Human body detection and geolocalization for uav rescue missions using color and thermal imagery. In *IEEE Aerospace Conference*, 2008.

[34] P. Rudol, M. Wzorek, G. Conte, and P. Doherty. Micro unmanned aerial vehicle visual servoing for cooperative indoor exploration. In *Proceedings of the IEEE Aerospace Conference*, 2008.

[35] B. van Linder W. van der Hoek and J.-J. Ch. Meyer. An integrated modal approach to rational agents. In M. Wooldridge and A. Rao, editors, *Foundations of Foundations of Rational Agency*, volume 14 of *Applied Logic Series*. An Integrated Modal Approach to Rational Agents, 1998.

[36] M. Wzorek, G. Conte, P. Rudol, T. Merz, S. Duranti, and P. Doherty. From motion planning to control – a navigation framework for an unmanned aerial vehicle. In *Proceedings of the 21st Bristol International Conference on UAV Systems*, 2006.

[37] M. Wzorek and P. Doherty. Reconfigurable path planning for an autonomous unmanned aerial vehicle. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*, pages 438–441, 2006.

[38] M. Wzorek, J. Kvarnström, and P. Doherty. Choosing path replanning strategies for unmanned aircraft systems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2010.

[39] M. Wzorek, D. Landen, and P. Doherty. GSM technology as a communication media for an autonomous unmanned aerial vehicle. In *Proceedings of the 21st Bristol International Conference on UAV Systems*, 2006.

Exploiting Spatial and Temporal Flexibility for Plan Execution of Hybrid, Under-actuated Systems

Andreas G. Hofmann and Brian C. Williams

Computer Science and Artificial Intelligence Lab, MIT
32 Vassar St. rm. 32-275
Cambridge, MA 02139
hofma@csail.mit.edu, williams@mit.edu

Abstract

Robotic devices, such as rovers and autonomous spacecraft, have been successfully controlled by plan execution systems that use plans with temporal flexibility to dynamically adapt to temporal disturbances. To date these execution systems apply to discrete systems that abstract away the detailed dynamic constraints of the controlled device. To control dynamic, under-actuated devices, such as agile bipedal walking machines, we extend this execution paradigm to incorporate detailed dynamic constraints.

Building upon prior work on dispatchable plan execution, we introduce a novel approach to flexible plan execution of hybrid under-actuated systems that achieves robustness by exploiting spatial as well as temporal plan flexibility. To accomplish this, we first transform the high-dimensional system into a set of low dimensional, weakly coupled systems. Second, to coordinate these systems such that they achieve the plan in real-time, we compile a plan into a *concurrent timed flow tube description.* This description represents all feasible control trajectories and their temporal coordination constraints, such that each trajectory satisfies all plan and dynamic constraints. Finally, the problem of runtime plan dispatching is reduced to maintaining state trajectories in their associated flow tubes, while satisfying the coordination constraints. This is accomplished through an efficient local search algorithm that adjusts a small number of control parameters in real-time. The first step has been published previously; this paper focuses on the last two steps. The approach is validated on the execution of a set of bipedal walking plans, using a high fidelity simulation of a biped.

Introduction

Past work in qualitative reasoning has produced methods for controlling dynamic systems that are distinguished in their use of qualitative descriptions of dynamics to control robustly over large regions of state space [Kuipers and Ramamoorthy, 2001; Hofbaur, 1999]. In particular, the qualitative concept of flow tubes [Bradley and Zhao, 1993; Frazzoli, 2001] explicitly defines the control regime that is handled robustly. With this approach, compile time methods identify bundles of trajectories, called *flow tubes*, that navigate the system to a desired equilibrium point.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

One limitation of the approach is the exhaustive state space exploration performed by these methods. Consequently, they have only been applied to relatively low-dimensional systems, and have not scaled well to high-dimensional systems at the level of complexity, for example, of walking robots (Fig. 1).

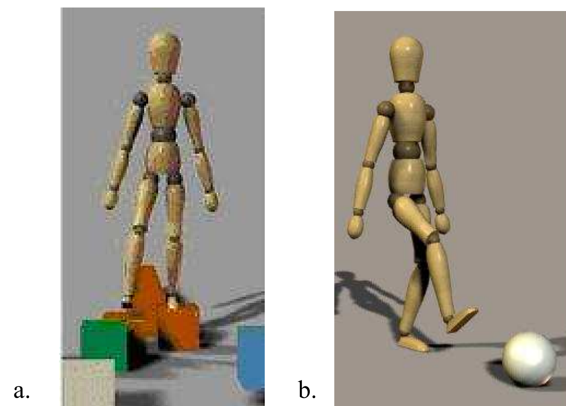


Fig. 1 – a. Walking on difficult terrain; b. kicking a ball

A second limitation is that the goals achieved by these qualitative controllers are restricted to simple set point regions; this type of description is not adequate for specifying complex tasks. Such complex tasks are distinguished by the execution of sequences of concurrent activities that are coordinated through timing constraints. Furthermore, while flow tubes provide state-space trajectory flexibility, they do not provide a representation of temporal flexibility.

Conversely, work on temporally flexible plan execution deals with the execution of complex tasks with many of these attributes [Tsamardinos et al., 1998; Muscettola et al., 1998]. This approach achieves robustness to disturbances through compile-time methods that make explicit the family of activity execution schedules that satisfy a plan's temporal constraints, and

then dynamically updating this family of schedules in response to disturbances.

However, this approach ignores the continuous dynamics of the underlying plant, and assumes that plan activities can be started and finished at arbitrary times, as long as these times are within the bounds specified by the temporal constraints of the plan. This is not valid for under-actuated dynamic systems, like walking bipeds, because the state of these systems is modified continuously through their second derivative (e.g., acceleration) resulting in inertia that can delay the achievement of a desired goal region.

Furthermore, both flow tube based control systems, and temporally flexible plan execution systems, as well as STRIPS planners in general, rely heavily on the notion of equilibrium points. Equilibrium points are points in state space where the system is at rest, and remains at rest if there is no disturbance or control action. This emphasis on equilibrium points makes these methods unsuitable for an important class of problems in which high performance is desired from an under-actuated plant.

Agile bipedal walking on difficult terrain, shown in Fig. 1a, is an important example of this class of problem. Such walking is achieved through appropriate velocity control, with the emphasis on limit cycle stability, rather than on achieving an equilibrium point. In fact, agile walking, and similar high performance tasks, is characterized by a lack of equilibrium points; successful execution is defined by achieving a sequence of goal regions that do not contain any equilibrium points. A related example is kicking a soccer ball, as shown in Fig. 1b. Stepping movement must be synchronized with ball movement so that the kick happens when the ball is close enough.

For such systems, traditional concepts of stability are meaningless. Therefore, flow tube systems that explore state space in search of equilibrium points are not useful for this type of application. Similarly, the ability of a temporally flexible plan execution system to set activity execution times relies on the ability of the system to idle in an equilibrium point for an arbitrary period of time. This is not guaranteed in an under-actuated system. Thus, neither the flow tube systems, nor the activity execution systems, deal adequately with applications where there are few or no equilibrium points.

Key Innovations of Approach

In this paper we present *Chekhov*, a system for robust, task-level control of high-performance, under-actuated systems with continuous dynamics. Chekhov achieves responsiveness and robustness through a compiled representation, called a *qualitative control plan* (QCP). This type of plan unifies the seemingly disparate representations used in flow tube control and activity execution systems, resulting in a system that combines the strengths of each.

Similar to flow tubes, a QCP maintains a representation of bundles of feasible trajectories. However, unlike flow tubes, a QCP addresses the problem of a high dimensional state space by factoring the state space into a set of concurrently operating single-input, single-output 2nd-order systems, and by representing feasible trajectories with a set of concurrent flow tubes related through timing constraints. Similar to activity execution systems, the flow tubes in a QCP are associated with sequences of concurrent activities, rather than with a single setpoint; successful execution of the activities corresponds to successful execution of the plan. Also similar to activity execution systems, the temporal constraints of a QCP are compiled into sets of feasible schedules, and are updated dynamically using a simple dispatching algorithm. However, the approach is distinguished in that the QCP incorporates the temporal constraints imposed by the plant dynamics.

Similar to both flow tube control and activity execution systems, a QCP is executed robustly using a combination of off-line compilation methods, and efficient on-line dispatching that adapts dynamically to disturbances. As with flow tube systems, the dispatcher selects control values that maintain the system along a feasible trajectory within a flow tube. As with activity execution systems, the QCP dispatcher schedules the execution times of activities according to the compiled temporal constraints.

However, the approach is unique in that it also avoids the reliance on equilibrium points of the other two methods by supporting sequences of goal regions without such points. Because the temporal constraints imposed by the plant dynamics are included, the system knows how long it is able to reside in a goal region that does not contain an equilibrium point.

Compiled Model-based Executive for Under-actuated Systems

We use a *model-based executive* [Williams and Nayak, 1997; Leaute and Williams, 2005] to interpret plan goals, monitor plant state, and compute control actions, as shown in Fig. 2.

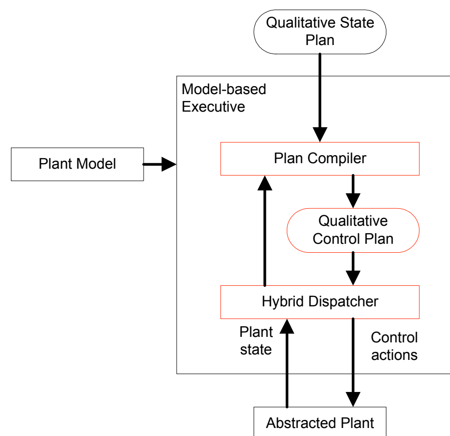


Fig. 2 – Model-based executive

The inputs to the model-based executive include a *Qualitative State Plan* (QSP), which specifies state-space and temporal requirements, the current plant state, and a plant model. The outputs are control actions for the *abstracted plant*. As shown in Fig. 2, the executive consists of a plan compiler, which is the off-line component, and a dispatcher, which is the on-line component. The plan compiler generates a QCP corresponding to the QSP. The QCP contains the flow tubes and the dispatchable graph. It is executed by the dispatcher, which schedules activity start times using a method similar to that used in the plan execution systems, and executes activities by keeping trajectories in their flow tubes.

Abstracted Plant

The abstracted plant is a set of linear, 2nd-order single-input single-output (SISO) systems, as shown in Fig. 3. Each SISO system is controlled by a *proportional-differential* (PD) control law [Ogata, 1982], which is of the form

$$\ddot{y} = k_p(y_{set} - y) + k_d(\dot{y}_{set} - \dot{y}) \qquad (1)$$

where y_{set} and $\dot{y}_{set}$ are position and velocity setpoints, and k_p and k_d are proportional and differential gains.

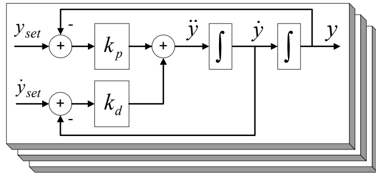


Fig. 3 – Abstracted plant is a set of SISO systems

Definition 1 (SISO System): A single-input single-output (SISO) system is a tuple, $\langle y_{set}, \dot{y}_{set}, k_p, k_d \rangle$. Given an SISO system, S, a plant trajectory of S, $traj(S)$, is a function of time, $y(t)$, that satisfies Eq. (1), where the control parameters in Eq. (1) are given by S. The SISO system may, optionally, have a set of constant constraints on the control parameters in S. These constraints are of the form

$$y_{set_\min} \le y_{set} \le y_{set_\max}$$
$$\dot{y}_{set_\min} \le \dot{y}_{set} \le \dot{y}_{set_\max}$$
$$k_{p_\min} \le k_p \le k_{p_\max}$$
$$k_{d_\min} \le k_d \le k_{d_\max}$$

These constraints may vary in time, according to discrete mode of the plant.

The modes of the plant, and associated operation constraints, are specified by activities in the QSP, as discussed in the next section. For a walking biped, discrete modes are defined by the *base of support*, the convex hull of points in contact with the ground. The base of support changes discontinuously with each step. The horizontal balance force that can be applied to the biped's *center of mass* (CM) is limited by the base of support, as shown in Fig. 4. The *ground reaction force vector* represents the overall force exerted on the CM. This vector emanates from a point on the ground called the *zero moment point* (ZMP) [Vukobratovic and Juricic, 1969], and points directly at the CM if the moment about the CM is zero [Popovic, et al., 2004]. Thus, the horizontal force on the CM can be adjusted by moving the ZMP. However, this horizontal force is limited because the ZMP is required to be within the base of support.

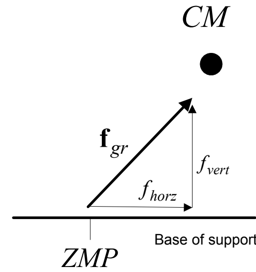


Fig. 4 – The ground reaction force vector, f_{gr}, emanates from the ZMP and exerts a horizontal force on the CM

For normal walking, the horizontal force on the CM is well approximated by

$$f_{horz} = k_s \, ZMP$$

, where k_s is an empirically determined spring constant [Popovic, et al., 2004]. In terms of Def. 1, y_{set} is the ZMP position, and k_p is k_s scaled by the total mass of the biped. Thus, a limit on ZMP position represents a limit on y_{set} of the form given in Def. 1.

In some applications, the actual plant is of the form shown in Fig. 3 and Def. 1. More typically, as is the case with a walking robot, the actual plant is nonlinear and tightly coupled. In this case, we linearize and decouple the actual plant using an enhanced feedback linearizing controller [Hofmann, et. al., 2004; Slotine and Li, 1991], which transforms the actual plant into the required abstracted form.

Qualitative State Plan

A QSP specifies the desired behavior of the plant in terms of allowed state trajectories. We use a *qualitative state* to specify desired state-space goal regions, and temporal constraints to specify time ranges by which the state space goals must be achieved. For a walking biped, a qualitative state indicates which feet are on the ground, and includes constraints on foot position. It may also include constraints on the biped's center of mass position and velocity. A sequence of qualitative states represents intermediate goals that lead to the final overall plan goal, as shown in Fig. 5.

A qualitative state plan has a set of *activities* representing constraints on desired state evolution. In Fig. 5, the activity *left foot ground 1* is for the left foot, *right foot ground 1*, is for the right foot, and CM1 – 4 are for the center of mass. Every activity starts and ends with an *event*, represented by a circle in Fig. 4. Events in this plan relate to behavior of the stepping foot. Thus, a *toe-off* event represents the stepping foot lifting off the ground, and a *heel-strike* event represents the stepping foot landing on the ground. Events define the boundaries of qualitative states. For example, the right toe-off event defines the end of the first qualitative state (double support), and the beginning of the second qualitative state (left single support).

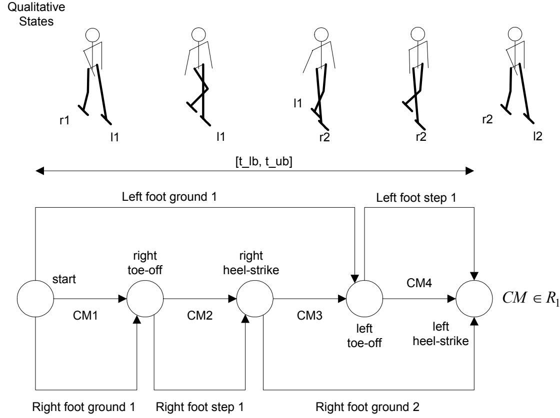


Fig. 5 - Example qualitative state plan for walking gait cycle. Circles represent events, and horizontal arrows represent activities.

The qualitative state plan in Fig. 5 has a temporal constraint between the start and finish events. In addition to temporal constraints, qualitative state plans can include required initial and goal regions for activities. In Fig. 5, the goal region constraint $CM \in R_1$ represents the requirement that the CM trajectory must be in region R_1 in order for the activity to finish successfully.

Each activity has an associated SISO system in the abstracted plant. An activity may specify constraints on the control parameters in the SISO system, corresponding to actuation limits. We assume that these constraints are constant limits; more general types of constraints are possible, but are beyond the scope of this discussion. In Fig. 5, the activities CM1 – CM4, representing CM movement, have different actuation limits. This is due to the discontinuous changes in the base of support resulting from the foot contact events.

Definition 2 (QSP): A qualitative state plan (QSP) is a tuple $\langle E, A, TC \rangle$, where E is a set of events, A is a set of activities, (Def. 3), and TC is a set of externally imposed temporal constraints on the events (Def. 4). An event, ev, represents a point in time.

Definition 3 (Activity): An activity is a tuple $\langle ev_s, ev_f, R_{op}, R_{init}, R_{goal}, S, A_{next} \rangle$, where ev_s and ev_f are activity start and finish events, R_{op} is a set of actuation constraints, R_{init} and R_{goal} are required state-space regions for start and finish of the activity, S is the SISO system (Def. 1), associated with the activity, and A_{next} is an optional successor activity. R_{op} specifies constant limits on the SISO control parameters.

Definition 4 (Temporal Constraint): A temporal constraint is a tuple $\langle ev_1, ev_2, l, u \rangle$, where ev_1 and ev_2 are events (Def. 4.6), and l and u represent lower and upper bounds on the time between these events, where $l \in \Re \cup \{-\infty\}$, $u \in \Re \cup \{\infty\}$ such that $l \leq t(ev_2) - t(ev_1) \leq u$.

Qualitative Control Plan

The interaction of constraints explicitly specified in the QSP, with constraints due to plant dynamics, makes determination of feasible trajectories computationally intensive. Because the system must run in real time, we seek to minimize the dispatcher's runtime computation by pre-computing sets of feasible trajectories that satisfy both types of constraints. The plan compiler performs this off-line computation, as shown in Fig. 2, outputting a QCP that contains the feasible trajectories.

A QCP augments the input QSP by adding flow tubes and a dispatchable graph [Muscettola, 1998]. The flow tubes represent feasible trajectory sets. The dispatchable graph represents temporal constraints in a form easily interpreted by the dispatcher. In this case, the temporal constraints represented in the QCP include both the temporal constraints explicitly specified in the QSP, and ones due to plant dynamics.

Definition 5 (QCP): A qualitative control plan (QCP) is a tuple $\langle q, F, g \rangle$, where q is the associated QSP, F is a set of flow tubes (Def. 6), and g is a dispatchable graph.

A flow tube is associated with a QSP activity. It represents feasible trajectories that result in successful execution of the activity. Thus, it is a function of the activity's constraints, and the dynamics of the activity's SISO system.

Definition 6 (Flow Tube): A flow tube is a set of trajectories, $Y = TUBE(a)$, defined over a time interval $\lfloor t_0, t_g \rfloor$, where a is a QSP activity, such that the goal and operating constraints of a are satisfied. Thus, a trajectory, $y(t) \in Y$ iff $\langle y(t_g), \dot{y}(t_g) \rangle \in R_{goal}(a)$ and all constraints in $R_{op}(a)$ are satisfied over the interval $\lfloor t_0, t_g \rfloor$, while obeying the dynamics of $S(a)$, as specified by Eq. (1).

An example flow tube is shown in Fig. 6a. A flow tube can be characterized as a set of cross-sectional regions in position-velocity phase space, one for each time in the interval $\lfloor t_0, t_g \rfloor$. Thus, such a cross section, r_{cs}, is a

function of the flow tube, and of time: $r_{cs} = CS(Y,t): t_0 \le t \le t_g$. Fig. 6b depicts cross sections for times t_0, t_1, and t_g.

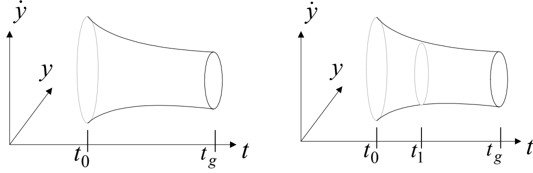


Fig. 6 – a. Example flow tube over interval $[t_0, t_g]$; b. cross sections at t_0, t_1, and t_g.

Next, consider the set, R_{cs}, of all cross sections in an interval $[t_0, t_1]$, where $t_0 \le t_1 \le t_g$. We use this set to investigate conditions under which the associated activity can be executed successfully for any start time in the interval $[t_0, t_1]$.

Theorem 1 (Temporal controllability of an activity): Let R_{cs} be a set of cross sections of a flow tube, Y, for activity a, where

$$R_{cs} = \bigcup_{t_0 \le t \le t_1} CS(Y,t)$$

If an allowed initial region, r_1, is a subset of every cross section in R_{cs} ($r_1 \subseteq r_{cs} \forall r_{cs} \in R_{cs}$), then the duration of a is controllable over the interval $[l, u]$, where $l = t_g - t_1$, and $u = t_g - t_0$. Conversely, each cross section of Y of which r_1 is a subset corresponds to a controllable duration of a. Furthermore, if the position trajectories in Y change monotonically, then the set of controllable durations specified in this way by r_1 will form a contiguous interval.

The monotonicity assumption is crucial in that it avoids disjunctions in the temporal constraints.

We now extend temporal controllability concepts to sequences of activities. Recall that activity sequences, such as the CM1-CM2 sequence in Fig. 5, can be used to represent discontinuous changes in operating constraints. The transition from CM1 to CM2 represents a transition between qualitative states; from double to single support.

In order to ensure that a flow tube for CM1 does not contain "dead end" trajectories, we require that all trajectories in such a flow tube have a feasible continuation in a flow tube for CM2. Hence, we require that the goal cross section of a flow tube, Y_1, for CM1 be a subset of a cross section of a flow tube, Y_2, for CM2, as shown in Fig. 6a. Thus, if Y_1 is defined over the interval $[t_0(Y_1), t_g(Y_1)]$, and Y_2 is defined over the interval $[t_0(Y_2), t_g(Y_2)]$, then the following must hold: $CS(Y_1, t_g(Y_1)) \subseteq CS(Y_2, t_2)$, where $t_0(Y_2) \le t_2 \le t_g(Y_2)$. For example, t_2 may be $t_0(Y_2)$, corresponding to the initial cross section of Y_2, as shown in Fig. 7a.

Suppose there exists a region, r_1, which, along with Y_1, determines a controllable interval $[l_1, u_1]$ for CM1, as discussed in Theorem 1. Suppose, also, that the goal region of Y_1 is a subset of the initial cross section of Y_2, as shown in Fig. 6a. Then, CM1 is temporally controllable over the range $[l_1, u_1]$, but CM2 must have a duration exactly equal to $t_g(Y_2) - t_0(Y_2)$. We now investigate ways to extend the controllable duration of CM2.

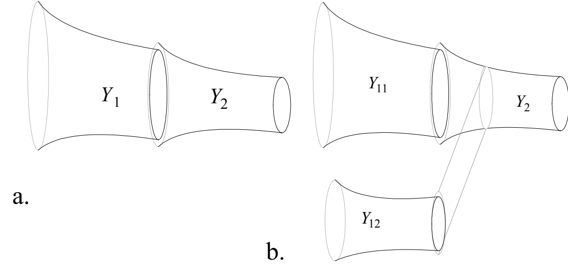


Fig. 7. – a. The goal region of flow tube Y_1 is a subset of a cross section of flow tube Y_2. b. Two flow tubes that intersect with Y_2.

Suppose there exists a set of flow tubes, Y_{1j}, for CM1 that all have the property that their goal regions are subsets of a cross section of Y_2. Two such flow tubes are depicted in Fig. 7b. Suppose, further, that these cross sections of Y_2 are contiguous, and correspond to an interval $[l_2, u_2]$, where $u_2 = t_g(Y_2) - t_0(Y_2)$, $l_2 = t_g(Y_2) - t_2$, and $t_0(Y_2) \le t_2 \le t_g(Y_2)$.

If a region, r_1, determines a controllable temporal range, $[l_{1j}, u_{1j}]$ for each Y_{1j}, and if the intersection of these temporal ranges is $[l_1, u_1]$, then CM1 is temporally controllable in the range $[l_1, u_1]$, and CM2 is temporally controllable in the range $[l_2, u_2]$. This concept can be applied recursively to successive activities in a sequence. In this way, if the initial state of the system is in r_1, the controllable duration of all activities in the sequence are known. These controllable durations are the temporal constraints due to the plant dynamics. They are added to the temporal constraints specified explicitly in the QSP.

Plan Compiler

Plan compilation is accomplished in two steps. First, the dispatchable graph is computed based on the temporal constraints in the QSP. This graph represents the tightest temporal constraints on all activities. Second, flow tubes are computed for each activity, based on the temporal constraints for the activity specified in the dispatchable graph. The computation of the dispatchable graph is based on the Floyd-Warshall all pairs shortest path algorithm. This computation has been described previously [Muscettola, 1998], hence we focus our discussion on computation of the flow tubes.

Flow tubes have a complex geometry. Therefore, any tractable flow tube representation will be an approximation of the feasible set. In order to ensure that any trajectory chosen by the dispatcher leads to plan execution success, we require our flow tube representation to include only feasible trajectories; the representation may include a

subset of all feasible trajectories, but not a superset [Kurzhanski and Varaiya, 1999].

Our flow tube approximation uses polyhedral cross sections at discrete time intervals [Vestal, 2001]. The time interval chosen matches the control increment of the dispatcher. Therefore, the dispatcher will always be able to access flow tube cross sections for exactly the correct time. Fig. 8 shows a flow tube cross-sectional region in position-velocity phase space, and its polyhedral approximation. Note that the approximation is a subset of the true region; the approximation does not include points in state-space that do not belong to feasible trajectories.

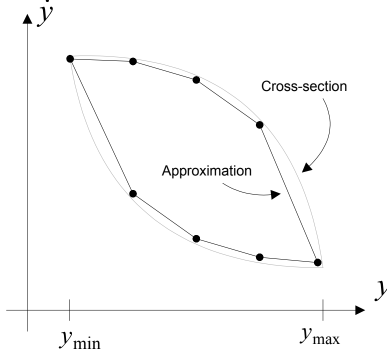


Fig. 8 – Flow tube cross section and approximation

In order to generate the polyhedral cross-sections, the plan compiler performs a *reachability analysis* that, for every vertex position, computes extreme corresponding velocities such that the resulting polygon contains only feasible trajectory points for the time associated with the cross section. We accomplish this reachability analysis by formulating constraints on cross section vertices as a linear program (LP).

The LP formulation is based on the analytical solution of Eq. (1). Eq. (1) is a 2nd-order linear differential equation, so its analytic solution is

$$\begin{aligned} y &= e^{\alpha t}\left(K_1 \cos \beta t + iK_2 \sin \beta t\right) + u/c \\ \dot{y} &= e^{\alpha t}\left(\beta\left(-K_1 \sin \beta t + iK_2 \cos \beta t\right) + \alpha\left(K_1 \cos \beta t + iK_2 \sin \beta t\right)\right) \end{aligned} \quad (2)$$

where

$$K_1 = y(0) - u/c,\ K_2 = (i/\beta)(\alpha K_1 - \dot{y}(0))$$

$$\alpha = -k_d/2, \beta = \left(-i\sqrt{k_d^{\,2} - 4k_p}\right)/2, u = k_p y_{set} + k_d \dot{y}_{set}$$

If we set the time, t, in Eq. (2) to a particular duration, d_i, corresponding to a particular cross section of interest, and if we fix gains k_p and k_d, then Eq. (2) can be expressed as

$$\begin{aligned} y &= f_1\left(y(0), \dot{y}(0), y_{set}, \dot{y}_{set}\right) \\ \dot{y} &= f_2\left(y(0), \dot{y}(0), y_{set}, \dot{y}_{set}\right) \end{aligned} \quad (3)$$

where f_1 and f_2 are linear for a particular setting of t, k_p, and k_d. Eq. (3) forms a set of equality constraints in the LP formulation. We also include a set of inequality constraints of the form

$$\begin{aligned} y_{set_\min} \le y_{set} \le y_{set_\max} \\ \dot{y}_{set_\min} \le \dot{y}_{set} \le \dot{y}_{set_\max} \end{aligned} \quad (4)$$

to represent the actuation limits, specified for the activity in the QSP. Further, we use a set of equality constraints to express

$$\langle y, \dot{y} \rangle \in R_{goal} \quad (5)$$

to ensure that state at the end of duration d_i is in the goal region, specified for the activity in the QSP. The formulation of (5) as a set of linear inequalities is straightforward because R_{goal} is required to be convex.

To compute a cross section for a particular R_{goal} and d_i, the plan compiler uses the formulation described by Eqs. (3 – 5), and sets the LP cost function to minimize $y(0)$. Solving this formulation yields the minimum initial position, $y_{\min}$, shown in Fig. 8. Repeating this process with the cost function set to maximize $y(0)$ yields the maximum initial position, $y_{\max}$. The compiler then establishes vertex positions at regular increments between $y_{\min}$ and $y_{\max}$. For each such vertex position, the compiler solves the LP formulation with the cost function set to first minimize, and then maximize, $\dot{y}(0)$, in order to find the minimum and maximum velocities for that position. This results in a set of vertices in position-velocity state space, which form the polyhedral approximation, as shown in Fig. 8.

The compiler computes cross section approximations for every d_i in the temporal range $[l, u]$, where this range is given for each activity by the minimum dispatchable graph. This set of cross sections approximates a flow tube, such as the one shown in Fig. 6.

Consider, next, the problem of computing flow tube approximations for a sequence of activities, as in Fig. 7. As stated previously, the goal region for flow tube Y_1 if Fig. 7a must be a subset of a cross section of Y_2. For the sake of completeness, we compute a separate set of cross sections for Y_1 for each cross section of Y_2 serving as a goal. This results in a set of flow tube approximations for Y_1, as shown in Fig. 7b. Each flow tube approximation in this set represents valid trajectories that satisfy the plan goals.

Dispatcher

In order to execute a QCP, the dispatcher must successfully execute each activity in the QCP. The dispatcher accomplishes this by scheduling start and finish events, using the QCP's dispatchable graph, and by setting

control parameters for each activity such that the associated trajectory reaches the activity's goal region at an acceptable time.

In order to execute an activity, the dispatcher performs three key functions: initialization, monitoring, and transition. Initialization is performed at the start of an activity's execution, monitoring is performed continuously during the activity's execution, and transition is performed at the finish of the activity's execution.

For initialization, assuming that all trajectories begin in the flow tube of their activity, the dispatcher chooses a goal duration for the control activity that is consistent with its execution window [Muscettola, 1998], and sets control parameters such that the state trajectory is predicted to be in the activity's goal region after the goal duration. The initialization function formulates a small *quadratic program* (QP) and solves it in order to determine these control parameters. This formulation is given in Fig. 9. Key to this formulation's simplicity is the fact that the analytic solution of Eq. 3 (functions f_1 and f_2) is used to predict the future state of the SISO system associated with the activity, and the fact that the formulation is guaranteed to produce a feasible solution, because the trajectory is within its flow tube. Further, presence of the trajectory in the flow tube guarantees that there exists a set of feasible control settings for all remaining activities in the plan, if there are no further trajectory disturbances.

FormulateControlQP($R_{goal}, y_{curr}, \dot{y}_{curr}, t_s, t_f$)

Parameters to optimize: y_{pred} , $\dot{y}_{pred}$, $y_{set}, \dot{y}_{set}$
Equality constraints: Eq. (3)
Inequality constraints: Eqs. (4, 5)
(Eq. (5) requires that trajectory prediction be within goal region)
Cost function

$$y_{goal} = \left(y_{\min}\left(R_{goal}\right) + y_{\max}\left(R_{goal}\right)\right)/2$$
$$\dot{y}_{goal} = \left(\dot{y}_{\min}\left(R_{goal}\right) + \dot{y}_{\max}\left(R_{goal}\right)\right)/2$$
$$cost = \left(y_{goal} - y_{pred}\right)^2 + \left(\dot{y}_{goal} - \dot{y}_{pred}\right)^2$$

Fig. 9 - Dispatcher QP formulation.

After initializing an activity, the dispatcher begins monitoring execution of that activity. To monitor execution, the dispatcher continually checks whether the state trajectory remains in its flow tube. If this is not the case, then plan execution has failed, and the dispatcher aborts to a higher-level control authority. Such a control authority might issue a new plan in response to such an abort. For example, the biped trying to kick the soccer ball may give up on this goal if it is no longer feasible.

If the state trajectory is in its flow tube, the dispatcher checks whether it is on track to be in the goal region at the end of the goal duration. This check is accomplished by evaluating Eq. (3) for the current state, and checking whether the predicted state is within the goal region. If this is not the case, the dispatcher corrects this situation by adjusting control parameters using the QP formulation of Fig. 9. Note that because the state trajectory is in its flow tube at this point, such a correction will always be possible.

As part of the monitoring function, the dispatcher also continually checks whether an activity's completion conditions are satisfied. Thus, it checks whether the state trajectory is in the activity's goal region, and whether the state trajectories of other activities whose completion must be synchronized are in their activity's goal regions. If all completion conditions for a control activity are satisfied, the dispatcher switches to the transition function.

If the activity being executed has a successor, the transition function invokes the initialization function for this successor. As part of this transition, the dispatcher notes the time of the transition event and propagates this through the temporal constraints using a local constraint propagation algorithm [Muscettola, 1998]. This propagation tightens execution windows of future events. When all activities in the QCP have been executed successfully, execution terminates.

Results and Discussion

Fig. 10 shows an example sequence of flow tubes corresponding to lateral CM movement for activities CM1 – CM4 in Fig. 5. This sequence represents two steps, which takes approximately 1.4 seconds.

Fig. 11 shows initial cross sections for the flow tube set for activity CM1 such that the goal region for CM2 is achieved. The goal region is specified explicitly in the QSP. If the initial state of the biped is in the controllable initial region, then the goal region can be achieved after any duration in the range [0.7, 1.0]. Since CM1 and CM2 represent a single step (single support followed by double support), the temporal controllability for two steps is [1.4, 2.0]. Such controllability can be used to synchronize biped movement with that of a moving soccer ball in order to kick it, as shown in Fig. 12.

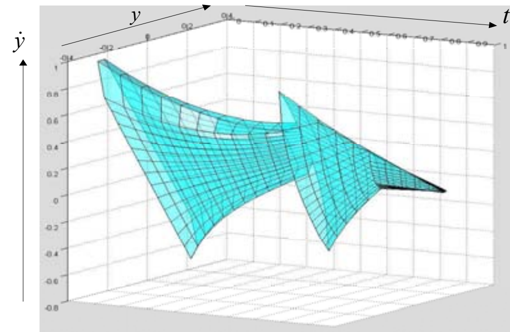


Fig. 10 – Example flow tube sequence.

If the initial state of the biped is outside the controllable initial region shown in Fig. 11, then temporal controllability is reduced. This expansion of the initial state region can be thought of as an enlargement of r_1 in Theorem 1, which results in r_1 being a subset of fewer cross sections. Thus, the requirement $r_1 \subseteq r_{cs} \forall r_{cs} \in R_{cs}$,

from Theorem 1 implies a smaller set R_{cs}, and a correspondingly smaller temporally controllable range. This trade-off is a direct result of the actuation limits and plant dynamics. Our plan compiler supports user control over this tradeoff through adjustment of required initial regions in the QSP.

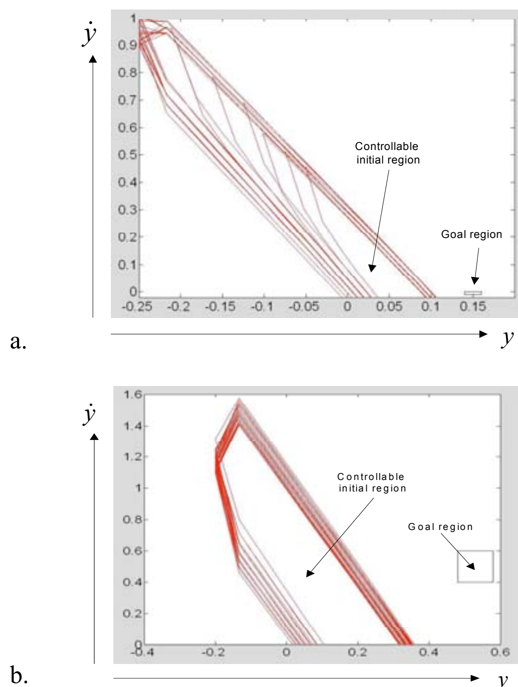


Fig. 11 – Initial cross sections for CM1 that achieve goal region in CM2; a. lateral, b. forward.

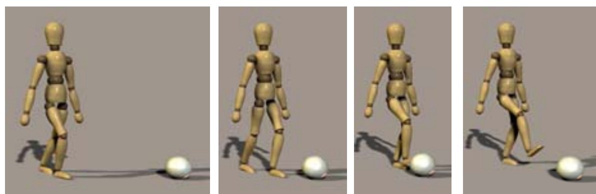


Fig. 12 – Walking to a moving soccer ball and kicking it.

An important challenge with this approach is the potentially large number of flow tubes in the set Y_{1j} that may have to be computed, especially when the method is applied recursively to longer activity sequences. This results in a fan-out for each predecessor activity. This problem can be mitigated by explicitly specifying tight temporal bounds in the QSP, and by explicitly specifying goal regions. An explicitly specified goal region serves as a root that terminates fan-out. An area of current research is incremental shifting of flow tubes. With such a capability, the full number of flow tubes in the set Y_{1j} would not have to be computed. Rather, a sparse subset would be computed, the elements of which would be shifted as needed.

References

Bradley, E. and Zhao, F. 1993. *Phase-space control system design*. *Control Systems*, 13(2),39-46

Frazzoli, E. 2001. *Robust Hybrid Control for Autonomous Vehicle Motion Planning*. Ph.D. Thesis, MIT

Hofbaur, M. 1999. *Lyapunov Methods for Semiquantitative Simulation*. Ph.D. Thesis, TU Graz

Hofmann, A., Massaquoi, S., Popovic, M., and Herr, H., 2004. *A sliding controller for bipedal balancing using integrated movement of contact and non-contact limbs. Proc. International Conference on Intelligent Robots and Systems* (IROS). Sendai, Japan

Kuipers, B., and Ramamoorthy, S. 2001. *Qualitative Modeling and Heterogeneous Control of Global System Behavior*. Hybrid Systems Control Conference.

Kurzhanski, A., and Varaiya, P. 1999. *Ellipsoidal Techniques for Reachability Analysis: Internal Approximation*

Leaute, T., Williams, A. 2005. *Coordinating Agile Systems Through the Model-based Execution of Temporal Plans*. ICAPS, 2005

Muscettola, N., Morris, P., and Tsamardinos, L. 1998 *Reformulating temporal plans for efficient execution. Proc. Of Sixth Int. Conf. On Principles of Knowledge Representation and Reasoning*

Popovic, M., Hofmann, A., Herr, H. 2004. *Angular momentum regulation during human walking: biomechanics and control. Proceedings of the International Conference on Robotics and Automation (ICRA). New Orleans (LA, USA).*

Tsamardinos, I., Muscettola, N., Morris, P. 1998. *Fast Transformation of Temporal Plans for Efficient Execution.* AAAI

Vestal, S. 2001. *A New Linear Hybrid Automata Reachability Procedure.* HSCC

Vukobratovic, M. and Juricic, D. 1969. *Contribution to the Synthesis of biped Gait. IEEE Transactions on Bio-Medical Engineering, Vol. BME-16, No. 1, 1969, pp. 1-6*

Williams, B. and Nayak, P. 1997. *A Reactive Planner for a Model-based Executive. Proceedings of the International Joint Conference on Artificial Intelligence*

Stream-Based Reasoning in DyKnow⋆

Fredrik Heintz, Jonas Kvarnström, and Patrick Doherty
{frehe, jonkv, patdo}@ida.liu.se

Dept. of Computer and Information Science
Linköping University, 581 83 Linköping, Sweden

Abstract. The information available to modern autonomous systems is often in the form of streams. As the number of sensors and other stream sources increases there is a growing need for incremental reasoning about the incomplete content of sets of streams in order to draw relevant conclusions and react to new situations as quickly as possible. To act rationally, autonomous agents often depend on high level reasoning components that require crisp, symbolic knowledge about the environment. Extensive processing at many levels of abstraction is required to generate such knowledge from noisy, incomplete and quantitative sensor data. We define *knowledge processing middleware* as a systematic approach to integrating and organizing such processing, and argue that connecting processing components with *streams* provides essential support for steady and timely flows of information. DyKnow is a concrete and implemented instantiation of such middleware, providing support for stream reasoning at several levels. First, the formal KPL language allows the specification of streams connecting knowledge processes and the required properties of such streams. Second, chronicle recognition incrementally detects complex events from streams of more primitive events. Third, complex metric temporal formulas can be incrementally evaluated over streams of states. DyKnow and the stream reasoning techniques are described and motivated in the context of a UAV traffic monitoring application.

1 Introduction

Modern autonomous systems usually have many sensors producing continuous streams of data. As the systems become more advanced the number of sensors grow, as exemplified by the humanoid robot CB^2 which has 2 cameras, 2 microphones, and 197 tactile sensors [1]. Further communication streams are produced when such systems interact. Some systems may also be connected to the Internet and have the opportunity to access streams of online information such as weather reports, news about the area, and so on. The fact that much of this information is available in the form of streams highlights the growing need for advanced stream processing capabilities in autonomous systems, where one can incrementally reason about the incomplete content of a set of streams in order to draw new conclusions as quickly as possible. This is in contrast to many of the current techniques used in formal knowledge representation and reasoning, which

⋆ This work is partially supported by grants from the Swedish Foundation for Strategic Research (SSF) Strategic Research Center MOVIII, the Swedish Research Council (VR) Linnaeus Center CADICS, and the Center for Industrial Information Technology CENIIT (10.04).

assume a more or less static knowledge base of facts to be reasoned about.

Additionally, much of the required knowledge must ultimately originate in physical sensors, but whereas deliberative functionalities tend to assume symbolic and crisp knowledge about the current state of the world, the information extracted from sensors often consists of noisy and incomplete quantitative data on a much lower level of abstraction. Thus, there is a wide gap between the information about the world normally acquired through sensing and the information that deliberative functionalities assume to be available for reasoning.

Bridging this gap is a challenging problem. It requires constructing suitable representations of the information that can be extracted from the environment using sensors and other available sources, processing the information to generate information at higher levels of abstraction, and continuously maintaining a correlation between generated representations and the environment itself. We use the term *knowledge processing middleware* for a principled and systematic software framework for bridging the gap between sensing and reasoning in a physical agent.

We believe that a stream-based approach to knowledge processing middleware is appropriate. To demonstrate the feasibility we have developed DyKnow, a fully implemented stream-based framework providing both conceptual and practical support for structuring a knowledge processing system as a set of streams and computations on streams [2, 3]. The properties of each stream is specified by a declarative *policy*. Streams represent aspects of the past, current, and future state of a system and its environment. Input can be provided by a wide range of distributed information sources on many levels of abstraction, while output consists of streams representing objects, attributes, relations, and events. DyKnow also explicitly supports two techniques for incremental reasoning with streams: Chronicle recognition for detecting complex events and progression of metric temporal logic to incrementally evaluate temporal logical formulas.

DyKnow and the stream reasoning techniques are described and motivated in the context of a UAV traffic monitoring application.

2 A Traffic Monitoring Scenario

Traffic monitoring is an important application domain for autonomous unmanned aerial vehicles (UAVs), providing a plethora of cases demonstrating the need for stream reasoning and knowledge processing middleware. It includes surveillance tasks such as detecting accidents and traffic violations, finding accessible routes for emergency vehicles, and collecting traffic pattern statistics.

Suppose a human operator is trying to maintain situational awareness about traffic in an area using static and mobile sensors such as surveillance cameras together with an unmanned helicopter. Reducing the amount of information sent to the operator also reduces her cognitive load, helping her to focus her attention on salient events. Therefore, each sensor platform should monitor traffic situations and only report back relevant high-level events, such as reckless overtakes and probable drunk driving.

Traffic violations, or other events to be detected, should be represented formally and declaratively. This can be done using *chronicle recognition* [4], where each chronicle defines a parameterized class of complex events as a simple temporal network [5] whose nodes correspond to occurrences of high-level qualitative events and edges correspond

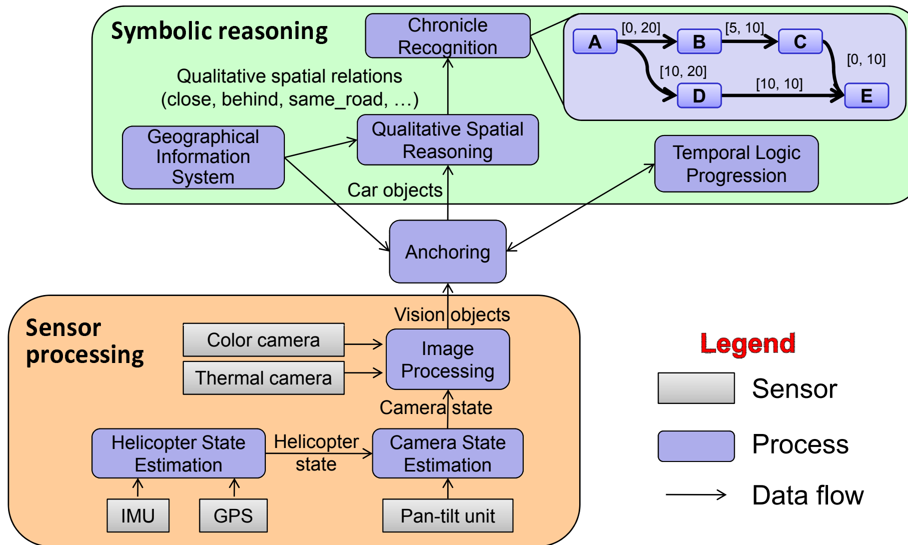


Fig. 1. An overview of how the processing required for traffic surveillance could be organized.

to metric temporal constraints. For example, events representing changes in qualitative spatial relations such as $\mathsf{beside}(car_1, car_2)$, $\mathsf{close}(car_1, car_2)$, and $\mathsf{on}(car_1, road_7)$ might be used to detect a reckless overtake. Creating these high-level representations from low-level sensor data, such as video streams from color and thermal cameras, involves extensive information and knowledge processing within each sensor platform.

Fig. 1 provides an overview of how part of the incremental processing required for the traffic surveillance task could be organized as a set of distinct DyKnow knowledge processes. At the lowest level, a *helicopter state estimation component* uses data from an *inertial measurement unit* (IMU) and a *global positioning system* (GPS) to generate a stream of position and attitude estimates. A *camera state estimation component* uses this information, together with a stream of states from the *pan-tilt unit* on which the cameras are mounted, to generate a stream of current camera states. The *image processing component* uses the camera state stream to determine where the camera is currently pointing. Video streams from the *color* and *thermal cameras* can then be analyzed in order to generate a stream of *vision percepts* representing hypotheses about moving and stationary physical entities, including their approximate positions and velocities.

Symbolic formalisms such as chronicle recognition require a consistent assignment of symbols, or identities, to the physical objects being reasoned about and the sensor data received about those objects. This is a process known as *anchoring* [6]. Image analysis may provide a partial solution, with vision percepts having symbolic identities that persist over short intervals of time. However, changing visual conditions or objects temporarily being out of view lead to problems that image analysis cannot (and should not) handle. This is the task of the anchoring component, which uses *progression* over a stream of states to evaluate potential hypotheses expressed as formulas in a metric temporal logic. The anchoring system also assists in object classification and in the ex-

traction of higher level attributes of an object. For example, a *geographic information system* can be used to determine whether an object is currently on a road or in a crossing. Such attributes can in turn be used to derive streams of relations *between* objects, including *qualitative spatial relations* such as $\mathsf{beside}(car_1, car_2)$ and $\mathsf{close}(car_1, car_2)$. Streams of concrete events corresponding to changes in these predicates and attributes finally provide sufficient information for the *chronicle recognition system* to determine when higher-level events such as reckless overtakes occur.

3 Stream-Based Knowledge Processing Middleware

Knowledge processing for a physical agent is fundamentally incremental in nature. Each part and functionality in the system, from sensing up to deliberation, needs to receive relevant information about the environment with minimal delay and send processed information to interested parties as quickly as possible. Rather than using polling, explicit requests, or similar techniques, we have therefore chosen to model and implement the required flow of data, information, and knowledge in terms of *streams*, while computations are modeled as active and sustained *knowledge processes* ranging in complexity from simple adaptation of raw sensor data to complex reactive and deliberative processes. This forms the basis for *stream-based knowledge processing middleware*, which we believe will be useful in a broad range of applications. A concrete implemented instantiation, DyKnow, will be discussed later.

Streams lend themselves easily to a *publish/subscribe* architecture. Information generated by a knowledge process is published using one or more *stream generators*, each of which has a (possibly structured) *label* serving as a global identifier within a knowledge processing application. Knowledge processes interested in a particular stream of information can subscribe using the label of the associated stream generator, which creates a new stream without the need for explicit knowledge of which process hosts the generator. Information produced by a process is immediately provided to the stream generator, which asynchronously delivers it to all subscribers, leaving the knowledge process free to continue its work.

In general, streams tend to be asynchronous in nature. This can often be the case even when information is sampled and sent at regular intervals, due to irregular and unpredictable transmission delays in a distributed system. In order to minimize delays and avoid the need for frequent polling, stream implementations should be push-based and notify receiving processes as soon as new information arrives.

Using an asynchronous publish / subscribe pattern of communication decouples knowledge processes in time, space, and synchronization [7], providing a solid foundation for distributed knowledge processing applications.

For processes that do not require constant updates, such as an automated task planner that needs an initial state snapshot, stream generators also provide a query interface to retrieve current and historic information generated by a process. Integrating such queries into the same framework allows them to benefit from decoupling and asynchronicity and permits lower level processing to build on a continuous stream of input before a snapshot is generated.

3.1 Streams

Intuitively, a stream serves as a communication channel between two knowledge processes, where elements are incrementally added by a source process and eventually arrive at a destination process. Verifying whether the contents such a stream satisfies a specific policy requires a formal model. For simplicity, we define a stream as a snapshot containing its own history up to a certain point in time, allowing us to determine exactly which elements had arrive at any preceding time. This is essential for the ability to validate an execution trace relative to a formal system description.

Definition 1 (Stream). *A stream is a set of stream elements, where each stream element is a tuple $\langle t_a, \ldots \rangle$ whose first value, t_a, is a time-point representing the time when the element is available in the stream. This time-point is called the available time of a stream element and has to be unique within a stream.*

Given a stream structure, the information that has arrived at its receiving process at a particular time-point t consists of those elements having an available time $t_a \leq t$.

3.2 Policies

Each stream is associated with a *policy* specifying a set of requirements on its contents. Such requirements may include the fact that each value must constitute a significant change relative to the previous value, that updates should be sent with a specific sample frequency, or that there is a maximum permitted delay. A policy can also give advice on how to ensure that these requirements are satisfied, for example by indicating how to handle missing or excessively delayed values. For introspection purposes, policies should be declaratively specified. Concrete examples are given in Section 4.

Each subscription to a stream generator includes a specific policy to be used for the generated stream. The stream generator can use this policy to filter the output of a knowledge process or forward it to the process itself to control its internal setup. Those parts of the policy that are affected by transmission through a distributed system, such as constraints on delays, can also be used by a *stream proxy* at the receiving process. This separates the generation of stream content from its adaptation.

Definition 2 (Policy). *A policy is a declarative specification of the desired properties of a stream, which may include advice on how to generate the stream.*

3.3 Knowledge Processes

A knowledge process operates on streams. Some processes take streams as input, some produce streams as output, and some do both. A process that generates stream output does so through one or more stream generators to which an arbitrary number of processes may subscribe using different policies. An abstract view of a knowledge process is shown in Fig. 2.

Definition 3 (Knowledge process). *A knowledge process is an active and sustained process whose inputs and outputs are in the form of streams.*

Four distinct process types are identified for the purpose of modeling: Primitive processes, refinement processes, configuration processes, and mediation processes.

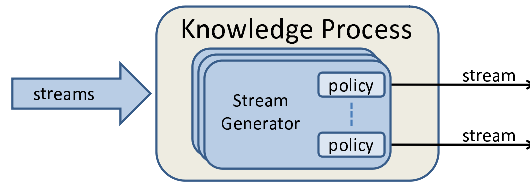


Fig. 2. A prototypical knowledge process

Primitive Processes *Primitive processes* serve as interfaces to the outside world, connecting to sensors, databases, or other information sources and providing their output in the form of streams. Such processes have no stream inputs but provide a non-empty set of stream generators. Their complexity may range from simple adaptation of external data into a stream-based framework to more complex tasks such as image processing.

Definition 4 (Primitive process). *A primitive process is a knowledge process without input streams that provides output through one or more stream generators.*

Refinement Processes The main functionality of stream-based knowledge processing middleware is to process streams to create more refined data, information, and knowledge. This type of processing is done by a *refinement process* which takes a set of streams as input and provides one or more stream generators providing stream outputs. For example, a refinement process could do image processing, fuse sensor data using Kalman filters estimating positions from GPS and IMU data, or reason about qualitative spatial relations between objects.

Definition 5 (Refinement process). *A refinement process is a knowledge process that takes one or more streams as input and provides output through one or more stream generators.*

When a refinement process is created it subscribes to its input streams. For example, a position estimation process computing the position of a robot at 10 Hz could either subscribe to its inputs with the same frequency or use a higher frequency in order to filter out noise. If a middleware implementation allows a process to change the policies of its inputs during run-time, the process can dynamically tailor its subscriptions depending on the streams it is supposed to create.

In certain cases, a process must first collect information over time before it is able to compute an output. For example, a filter might require a number of measurements before it is properly initialized. This introduces a processing delay that can be remedied if the process is able to request 30 seconds of historic data, which is supported by the DyKnow implementation.

Configuration Processes Traffic monitoring requires position and velocity estimates for all currently monitored cars, a set that changes dynamically over time as new cars enter an area and as cars that have not been observed for some time are discarded.

This is an instance of a recurring pattern where the same type of information must be produced for a dynamically changing set of objects.

This could be achieved with a static process network, where a single refinement process estimates positions for all currently visible cars. However, processes and stream policies would have to be quite complex to support more frequent updates for a specific car which is the current focus of attention.

As an alternative, one can use a dynamic network of processes, where each refinement process estimates positions for a single car. A *configuration process* provides a fine-grained form of dynamic reconfiguration by instantiating and removing knowledge processes and streams as indicated by its input.

Definition 6 (Configuration process). *A configuration process is a knowledge process that takes streams as inputs, has no stream generators, and creates and removes knowledge processes and streams.*

For traffic monitoring, the input to the configuration process would be a single stream where each element contains the set of currently monitored cars. Whenever a new car is detected, the new (complete) set of cars is sent to the configuration process, which may create new processes. Similarly, when a car is removed, associated knowledge processes may be removed.

Mediation Processes Finally, a *mediation process* allows a different type of dynamic reconfiguration by aggregating or selecting information from a static or dynamic set of existing streams.

Aggregation is particularly useful in the fine-grained processing networks described above: If there is one position estimation process for each car, a mediation process can aggregate the outputs of these processes into a single stream to be used by those processes that do want information about all cars at once. In contrast to refinement processes, a mediation process can change its inputs over time to track the currently monitored set of cars as indicated by a stream of labels or label sets.

Selection forwards information from a particular stream in a set of potential input streams. For example, a mediation process can provide position information about the car that is the current focus of attention, automatically switching between position input streams as the focus of attention changes. Other processes interested in the current focus can then subscribe to a single semantically meaningful stream.

Definition 7 (Mediation process). *A mediation process is a knowledge process that changes its input streams dynamically and mediates the content on the varying input streams to a fixed number of stream generators.*

Stream Generators A knowledge process can have multiple outputs. For example, a single process may generate separate position and velocity estimates for a particular car. Each raw output is sent to a single *stream generator*, which can create an arbitrary number of output streams adapted to specific policies. For example, one process may wish to receive position estimates every 100 ms, while another may require updates only when the estimate has changed by at least 10 meters.

Definition 8 (Stream generator). *A stream generator is a part of a knowledge process that generates streams according to policies from output generated by the knowledge process.*

Using stream generators separates the generic task of adapting streams to policies from the specific tasks performed by each knowledge process. Should the generic policies supported by a particular middleware implementation be insufficient, a refinement process can still subscribe to the unmodified output of a process and provide arbitrarily complex processing of this stream.

Note that a stream generator is not necessarily a passive filter. For example, the generator may provide information about its current output policies to the knowledge process, allowing the process to reconfigure itself depending on parameters such as the current sample rates for all output streams.

4 DyKnow

DyKnow is a concrete instantiation of the generic stream-based middleware framework defined in the previous section. DyKnow provides both a conceptual framework for modeling knowledge processing and an implementation infrastructure for knowledge processing applications. The formal framework can be seen as a specification of what is expected of the implementation infrastructure. It can also be used by an agent to reason about its own processing. A detailed formal description of DyKnow is available in [2, 3].

DyKnow views the world as consisting of *objects* and *features*, where features may for example represent attributes of objects. The general stream concept is specialized to define *fluent streams* representing an approximation of the value of a feature over time. Two concrete classes of knowledge processes are introduced: *Sources*, corresponding to primitive processes, and *computational units*, corresponding to refinement processes. A computational unit is parameterized with one or more fluent streams. Each source and computational unit provides a *fluent stream generator* creating fluent streams from the output of the corresponding knowledge process according to *fluent stream policies*. The declarative language KPL is used for specifying knowledge processing applications.

DyKnow is implemented as a CORBA middleware service. It uses the CORBA event notification service [8] to implement streams and to decouple knowledge processes from clients subscribing to their output. See [3] for the details.

A *knowledge processing domain* defines the objects, values, and time-points used in a knowledge processing application. From them the possible fluent streams, sources, and computational units are defined. The semantics of a knowledge processing specification is defined on an interpretation of its symbols to a knowledge processing domain.

Definition 9 (Knowledge processing domain). *A knowledge processing domain is a tuple $\langle O, T, V \rangle$, where O is a set of objects, T is a set of time-points, and V is a set of values.*

4.1 Fluent Streams

Due to inherent limitations in sensing and processing, an agent cannot always expect access to the actual value of a feature over time but will have to use approximations. Such approximations are represented as *fluent streams*, a specialization of the previously introduced stream structure where elements are *samples*. Each sample represents an observation or estimation of the value of a feature at a specific point in time called the *valid time*. Like any stream element, a sample is also tagged with its *available time*, the time when it is ready to be processed by the receiving process after having been transmitted through a potentially distributed system.

The available time is essential when determining whether a system behaves according to specification, which depends on the information actually available as opposed to information that may have been generated but has not yet arrived. Having a specific representation of the available time also allows a process to send multiple estimates for a single valid time, for example by quickly providing a rough estimate and then running a more time-consuming algorithm to provide a higher quality estimate. Finally, it allows us to formally model delays in the availability of a value and permits an application to use this information introspectively to determine whether to reconfigure the current processing network to achieve better performance.

Definition 10 (Sample). *A sample in a domain $D = \langle O, T, V \rangle$ is either the constant* `no_sample` *or a stream element $\langle t_a, t_v, v \rangle$, where $t_a \in T$ is its available time, $t_v \in T$ is its valid time, and $v \in V$ is its value. The set of all possible samples in a domain D is denoted by S_D.*

Example 1. Assume a picture p is taken by a camera source at time-point 471, and that the picture is sent through a fluent stream to an image processing process where it is received at time 474. This is represented as the sample $\langle 474, 471, p \rangle$.

Assume image processing extracts a set b of blobs that may correspond to vehicles. Processing finishes at time 479 and the set of blobs is sent to two distinct recipients, one receiving it at time 482 and one receiving it at time 499. This information still pertains to the state of the environment at time 471, and therefore the valid time remains the same. This is represented as the two samples $\langle 482, 471, b \rangle$ and $\langle 499, 471, b \rangle$ belonging to distinct fluent streams.

The constant `no_sample` will be used to indicate that a fluent stream contains no information at a particular point in time, and can never be part of a fluent stream.

Definition 11 (Fluent stream). *A fluent stream in a domain D is a stream where each stream element is a sample from $S_D \setminus \{$*`no_sample`*$\}$.*

4.2 Sources

Primitive processes can be used to provide interfaces to external data producers or sensors, such as the GPS, IMU, and cameras on a UAV. A primitive process is formally modeled as a *source*, a function from time-points to samples representing the output of the primitive process at any point in time. If the function returns `no_sample`, the primitive process does not produce a sample at the given time.

Definition 12 (Source). *Let* $D = \langle O, T, V \rangle$ *be a domain. A source is a function* $T \mapsto S_D$ *mapping time-points to samples.*

4.3 Computational Units

Refinement processes are used to perform computations on streams, ranging from simple addition of integer values to Kalman filters, image processing systems, and even more complex functions. In a wide variety of cases, only a single output stream is required (though this stream may consist of complex values). It is also usually sufficient to have access to the current internal state of the process together with the most recent sample of each input stream to generate a new output sample. A process of this type can be modeled as a *computational unit*.

Definition 13 (Computational unit). *Let* $D = \langle O, T, V \rangle$ *be a domain. A computational unit with arity* $n > 0$*, taking n inputs, is associated with a partial function* $T \times S_D^n \times V \mapsto S_D \times V$ *of arity* $n + 2$ *mapping a time-point, n input samples, and a value representing the previous internal state to an output sample and a new internal state.*

The input streams to a computational unit do not necessarily contain values with synchronized valid times or available times. For example, two streams could be sampled with periods of 100 ms and 60 ms while a third could send samples asynchronously. In order to give the computational unit the maximum amount of information, we choose to apply its associated function whenever a new sample becomes available in *any* of its input streams, and to use the most recent sample in each stream. Should the unit prefer to wait for additional information, it can store samples in its internal state and return `no_sample` to indicate that no new output sample should be produced at this stage.

4.4 Fluent Stream Policies

A policy specifies the desired properties of a fluent stream and is defined as a set of constraints on the fluent stream. There are five types of constraints: Approximation, change, delay, duration, and order constraints.

A *change constraint* specifies what must change between two consecutive samples. Given two consecutive samples, **any update** indicates that some part of the new sample must be different, while **any change** indicates that the value or valid time must be different, and **sample every** *t* indicates that the difference in valid time must equal the sample period *t*.

A *delay constraint* specifies a maximum acceptable delay, defined as the difference between the valid time and the available time of a sample. Note that delays may be intentionally introduced in order to satisfy other constraints such as ordering constraints.

A *duration constraint* restricts the allowed valid times of samples in a fluent stream.

An *order constraint* restricts the relation between the valid times of two consecutive samples. The constraint **any order** does not constrain valid times, while **monotone order** ensures valid times are non-decreasing and **strict order** ensures valid times are strictly increasing. A sample change constraint implies a strict order constraint.

An *approximation constraint* restricts how a fluent stream may be extended with

new samples in order to satisfy its policy. If the output of a knowledge process does not contain the appropriate samples to satisfy a policy, a fluent stream generator could approximate missing samples based on available samples. The constraint **no approximation** permits no approximated samples to be added, while **use most recent** permits the addition of samples having the most recently available value.

For the stream generator to be able to determine at what valid time a sample must be produced, the **use most recent** constraint can only be used in conjunction with a complete duration constraint **from** t_f **to** t_t and a change constraint **sample every** t_s. For the stream generator to determine at what available time it should stop waiting for a sample and produce an approximation, this constraint must be used in conjunction with a delay constraint **max delay** t_d.

4.5 KPL

DyKnow uses the *knowledge processing language* KPL to declaratively specify *knowledge processing applications*, static networks of primitive processes (sources) and refinement processes (computational units) connected by streams. Mediation and configuration processes modify the setup of a knowledge processing application over time and are left for future work. For details of KPL including the formal semantics see [2, 3].

Definition 14 (KPL Grammar).

```
       KPL_SPEC ::= (  SOURCE_DECL | COMP_UNIT_DECL
                    |  FSTREAM_GEN_DECL | FSTREAM_DECL )+
    SOURCE_DECL ::= source SORT_SYM SOURCE_SYM
 COMP_UNIT_DECL ::= compunit SORT_SYM
                    COMP_UNIT_SYM '(' SORT_SYM ( ',' SORT_SYM )* ')'
FSTREAM_GEN_DECL ::= strmgen LABEL '='
                   ( SOURCE_SYM
                   | COMP_UNIT_SYM '(' FSTREAM_TERM ( ',' FSTREAM_TERM )* ')' )
   FSTREAM_DECL ::= stream STREAM_SYM '=' FSTREAM_TERM
          LABEL ::= FEATURE_SYM ( '[' OBJECT_SYM ( ',' OBJECT_SYM )* ']' )?
   FSTREAM_TERM ::= LABEL ( with FSTREAM_POLICY )?
 FSTREAM_POLICY ::= STREAM_CONSTR ( ',' STREAM_CONSTR )*
  STREAM_CONSTR ::= APPRX_CONSTR | CHANGE_CONSTR | DELAY_CONSTR
                  | DURATION_CONSTR | ORDER_CONSTR
   APPRX_CONSTR ::= no approximation | use most recent
  CHANGE_CONSTR ::= any update | any change | sample every TIME_SYM
   DELAY_CONSTR ::= max delay ( TIME_SYM | oo )
DURATION_CONSTR ::= from TIME_SYM | ( from TIME_SYM )? to ( TIME_SYM | oo )
   ORDER_CONSTR ::= any order | monotone order | strict order
```

5 Chronicle Recognition

Many applications of autonomous vehicles involve surveillance and monitoring where it is crucial to recognize *events* related to objects in the environment. For example, a UAV monitoring traffic must be able to recognize events such as a car overtaking another, a

car stopping at an intersection, and a car parking next to a certain building.

We can classify events as being either *primitive* or *complex*. A primitive event is either directly observed or grounded in changes in feature values, while a complex event is defined as a spatio-temporal pattern of other events. The purpose of an event recognition system is to detect complex events from a set of observed or previously detected events. In the traffic monitoring domain, for example, the complex event of car A *overtaking* car B can be defined in terms of a chain of events where a car A is first behind, then left of, and finally in front of car B together with temporal constraints on the events such as the total overtake should take less than 120 seconds.

One formalism for expressing complex events is the chronicle formalism which represents and detects complex events described in terms of temporally constrained events [4]. The chronicle recognition algorithm takes a stream of time-stamped event occurrences and finds all matching chronicle instances as soon as possible. This makes it a good example of a stream reasoning technique. To do this, the algorithm keeps track of all possible developments in an efficient manner by compiling chronicles into simple temporal constraint networks [5]. To detect chronicle instances, the algorithm keeps track of all partially instantiated chronicle models. To begin with each chronicle model is associated with a completely uninstantiated instance. Each time a new event is received it is checked against all the partial instances to see if it matches any previously unmatched event. If that is the case, then a copy of the instance is created and the new event is integrated into the temporal constraint network by instantiating the appropriate variables and propagating all constraints [4]. This propagation can be done in polynomial time since the temporal constraint network is simple. It is necessary to keep the original partial chronicle instance to match a chronicle model against all subsets of event occurrences. If all the events have been matched then a complete instance has been found. Recognized instances of a chronicle can be used as events in another chronicle. The chronicle recognition algorithm is complete as long as the observed event stream is complete, i.e. any change of a value of an attribute is captured by an event.

Time is considered a linearly ordered discrete set of instants, whose resolution is sufficient to represent the changes in the environment. Time is represented by time-points and all the interval constraints permitted by the restricted interval algebra [9] are allowed. This means that it is possible to represent relations such as before, after, equal, and metric distances between time-points but not their disjunctions.

We have used chronicle recognition in the traffic monitoring application to detect traffic patterns [10].

6 Progression of Metric Temporal Logic

First order logic is a powerful technique for expressing complex relationships between objects. Metric temporal logics extends first order logics with temporal operators that allows metric temporal relationships to be expressed. For example, our temporal logic, which is a fragment of the Temporal Action Logic (TAL) [11], supports expressions which state that a formula F should hold within 30 seconds and that a formula F' should hold in every state between 10 and 20 seconds from now. This fragment is similar to the well known Metric Temporal Logic [12]. Informally, $\Diamond_{[\tau_1,\tau_2]} \phi$ ("eventually") holds at τ iff ϕ holds at some $\tau' \in [\tau+\tau_1, \tau+\tau_2]$, while $\Box_{[\tau_1,\tau_2]} \phi$ ("always") holds at τ iff ϕ

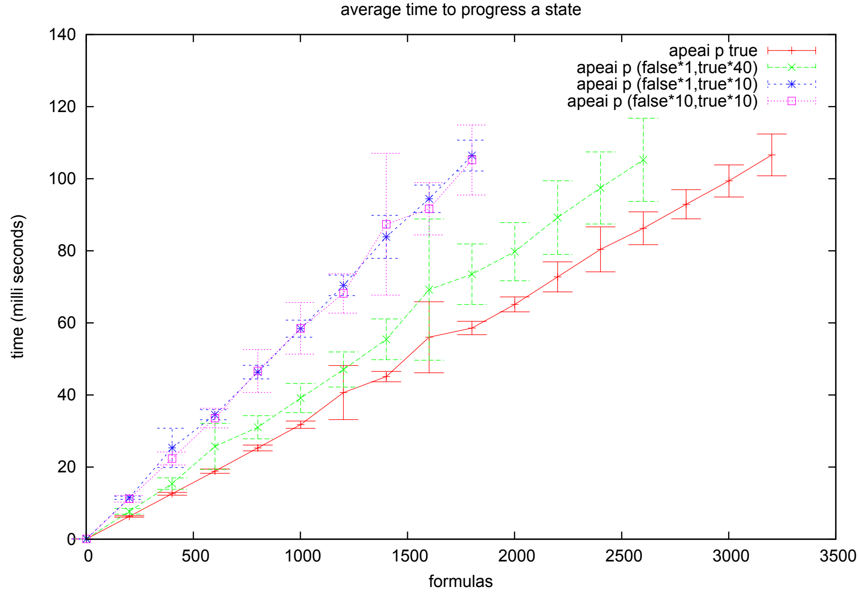


Fig. 3. Testing: Always Not p Implies Eventually Always p (average progression time).

holds at all $\tau' \in [\tau + \tau_1, \tau + \tau_2]$. Finally, $\phi \, \mathsf{U}_{[\tau_1,\tau_2]} \, \psi$ ("until") holds at τ iff ψ holds at some $\tau' \in [\tau + \tau_1, \tau + \tau_2]$ such that ϕ holds in all states in (τ, τ').

The semantics of these formulas are defined over infinite state sequences. To make metric temporal logic suitable for stream reasoning, the formulas are incrementally evaluated by DyKnow using progression over a timed state stream. The result of progressing a formula through the first state in a stream is a new formula that holds in the remainder of the state stream iff the original formula holds in the complete state stream. If progression returns true (false), the entire formula must be true (false), regardless of future states. See Heintz [3, 13] for formal details.

DyKnow also provides support for generating streams of states synchronizing distributed streams. Using their associated policies it is possible to determine when the best possible state at each time-point can be extracted.

Even though the size of a progressed formula may grow exponentially in the worst case, many common formulas do not. One example is the formula $\Box \neg p \rightarrow \Diamond_{[0,1000]} \Box_{[0,999]} \, p$, corresponding to the fact that if p is false, then within 1000 ms, there must begin a period lasting at least 1000 ms where p is true. To estimate the cost of evaluating this formula, it was progressed through several different state streams corresponding to the best case, the worst case, and two intermediate cases. A new state in the stream was generated every 100 ms, which means that all formulas must be progressed within this time limit or the progression will fall behind. The results in Fig. 3 shows that 100 ms is sufficient for the progression of between 1500 and 3000 formulas of this form on the computer on-board our UAV, depending on the state stream.

We have used this expressive metric temporal logic to monitor the execution of

complex plans in a logistics domain [13] and to express conditions for when to hypothesize the existence and classification of observed objects in an anchoring module [3, 14]. For example in execution monitoring, suppose that a UAV supports a maximum continuous power usage of M, but can exceed this by a factor of f for up to τ units of time, if this is followed by normal power usage for a period of length at least τ'. The following formula can be used to detect violations of this specification: $\Box\, \forall uav.(\mathsf{power}(uav) > M \rightarrow \mathsf{power} < f \cdot M \,\mathsf{U}_{[0,\tau]}\, \Box_{[0,\tau']}\, \mathsf{power}(uav) \leq M)$

7 Related Work

The conceptual stream reasoning architecture proposed by Della Valle et al [15] consists of four stages: Select, Abstract, Reason, and Decide. The Select component uses filtering and sampling to select a subset of the available streams. These streams are then processed by the Abstract component to turn data into richer information by converting the content to RDF streams. The Reason component takes these RDF streams and reasons about their content. Finally, the Decide component evaluates the output and determines if the result is good enough or if some of the previous stages have to be adapted and further data processed.

Compared to this framework, DyKnow provides support for all four stages to a varying degree without restricting itself to serial processing of the four steps. The policies and the computational units provide tools for selection and abstraction, with particular support from the anchoring module to associated symbols to sensor data. The chronicle recognition component and the formula progression engine are two particular stream reasoning techniques that can be applied to streams.

In general, stream reasoning is related to many areas, since the use of streams is common and have many different uses. Some of the most closely related areas are data stream management systems [16–18], publish/subscribe middleware [7, 19], event-based systems [20–23], complex event processing [24, 25], and event stream processing [26]. Even though most of these systems provide some contributions to stream reasoning few of them provide explicit support for lifting the abstraction level and doing general reasoning on the streams. The approaches that come the closest are complex event processing, but they are limited to events and do not reason about objects or situations.

8 Conclusions

We have presented DyKnow, a stream-based knowledge processing middleware framework, and shown how it can be used for stream reasoning. Knowledge processing middleware is a principled and systematic software framework for bridging the gap between sensing and reasoning in a physical agent. Since knowledge processing is fundamentally incremental in nature it is modeled as a set of active and sustained knowledge processes connected by streams where each stream is specified by a declarative policy.

DyKnow is a concrete and implemented instantiation of such middleware, providing support for stream reasoning at several levels. First, the formal KPL language allows the specification of streams connecting knowledge processes and the required properties of such streams. Second, chronicle recognition incrementally detects complex events

from streams of more primitive events. Third, complex metric temporal formulas can be incrementally evaluated over streams of states using progression.

Since DyKnow is a general framework providing both conceptual and implementation support for stream processing it is easy to add new functionality and further improve its already extensive support for stream reasoning.

References

1. Minato, T., Yoshikawa, Y., Noda, T., Ikemoto, S., Ishiguro, H., Asada, M.: Cb2: A child robot with biomimetic body for cognitive developmental robotics. In: Proceedings of IEEE/RAS International Conference on Humanoid Robotics. (2007)
2. Heintz, F., Kvarnström, J., Doherty, P.: Bridging the sense-reasoning gap: DyKnow – stream-based middleware for knowledge processing. Journal of Advanced Engineering Informatics **24**(1) (2010) 14–26
3. Heintz, F.: DyKnow: A Stream-Based Knowledge Processing Middleware Framework. PhD thesis, Linköpings universitet (2009)
4. Ghallab, M.: On chronicles: Representation, on-line recognition and learning. In: Proceedings of KR. (1996) 597–607
5. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. AIJ **49** (1991)
6. Coradeschi, S., Saffiotti, A.: An introduction to the anchoring problem. Robotics and Autonomous Systems **43**(2-3) (2003) 85–96
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2) (2003) 114–131
8. Gore, P., Schmidt, D.C., Gill, C., Pyarali, I.: The design and performance of a real-time notification service. In: Proc. of Real-time Technology and Application Symposium. (2004)
9. Nebel, B., Burckert, H.J.: Reasoning about temporal relations: A maximumal tractable subclass of allen's interval algebra. Journal of ACM **42**(1) (1995) 43–66
10. Heintz, F., Rudol, P., Doherty, P.: From images to traffic behavior – a UAV tracking and monitoring application. In: Proceedings of Fusion'07. (2007)
11. Doherty, P., Kvarnström, J.: Temporal action logics. In Lifschitz, V., van Harmelen, F., Porter, F., eds.: The Handbook of Knowledge Representation. Elsevier (2007)
12. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems **2**(4) (1990) 255–299
13. Doherty, P., Kvarnström, J., Heintz, F.: A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. Journal of Autonomous Agents and Multi-Agent Systems **19**(3) (2009)
14. Heintz, F., Kvarnström, J., Doherty, P.: A stream-based hierarchical anchoring framework. In: Proc. of IROS. (2009)
15. Valle, E.D., Ceri, S., Barbieri, D., Braga, D., Campi, A.: A First step towards Stream Reasoning. In: Proceedings of the Future Internet Symposium. (2008)
16. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A new model and architecture for data stream management. VLDB Journal (August 2003)
17. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of PODS'02. (2002)
18. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system. In: Proceedings of CIDR'03. (2003)
19. OMG: The data-distribution service specification v 1.2 (jan 2007)

20. Carzaniga, A., Rosenblum, D.R., Wolf, A.L.: Challenges for distributed event services: Scalability vs. expressiveness. In: Engineering Distributed Objects. (1999)
21. Pietzuch, P.: Hermes: A Scalable Event-Based Middleware. PhD thesis, University of Cambridge (2004)
22. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems **19**(3) (2001) 332–383
23. Cugola, G., Nitto, E.D., Fuggetta, A.: The jedi event-based infrastructure and its application to the development of the opss wfms. IEEE Trans. Softw. Eng. **27**(9) (2001) 827–850
24. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston, MA, USA (2002)
25. Gyllstrom, D., Wu, E., Chae, H.J., Diao, Y., Stahlberg, P., Anderson, G.: Sase: Complex event processing over streams. In: Proceedings of CIDR'07. (2007)
26. Demers, A., Gehrke, J., Biswanath, P., Riedewald, M., Sharma, V., White, W.: Cayuga: A general purpose event monitoring system. In: Proceedings of CIDR'07. (2007)

Self-Maintenance for Autonomous Robots in the Situation Calculus

Stefan Schiffer Andreas Wortmann Gerhard Lakemeyer

Knowledge-Based Systems Group
RWTH Aachen University
Aachen, Germany
andreas.wortmann@gmail.com
(schiffer,gerhard)@cs.rwth-aachen.de

Abstract. In order to make a robot execute a given task plan more robustly we want to enable it to take care of its self-maintenance requirements during online execution of this program. This requires the robot to know about the (internal) states of its components, constraints that restrict execution of certain actions and possibly also how to recover from faulty situations. The general idea is to implement a transformation process on the plans, which are specified in the agent programming language ReadyLog, to be performed based on explicit (temporal) constraints. Afterwards, a 'guarded' execution of the transformed program should result in more robust behavior.

1 Introduction

Today's artificial intelligence provides a rich framework for the development of "intelligent" autonomous agents. Several branches explore improvements of these agents, dealing with perception, human-robot-interaction, locomotion, reasoning, planning, and more. One aspect of current robotics research is "the study of the knowledge representation and reasoning problems faced by an autonomous robot (or agent) in a dynamic and incompletely known world" [1], coined as *cognitive robotics* by Ray Reiter. The central effort of Reiter's vision [2] "is to develop an understanding of the relationship between the knowledge, the perception, and the action of such a robot". This is outlined by through several questions the research program of *cognitive robotics* is supposed to answer, especially "what does the robot need to know about its environment" and "when should the inner workings of an action be available to the robot for reasoning". We approach a specialization of their intersection, namely "what does the robot need to know about itself and its requirements". This is especially interesting as present agents are often unable to explicate their requirements (e.g., calibration of manipulators before usage) relative to a plan. They usually need these requirements to be considered externally and in advance, otherwise they fail during plan execution. Therefore, we propose a *constraint based self-maintenance framework*, which will enable an agent to monitor its self-maintenance requirements during program execution. Whenever the self-maintenance framework determines

unsatisfied requirements, appropriate recovery measures are performed online. This behaviour increases agent autonomy and robustness. We do so by adding a program transformation step in ReadyLog, a logic-based robot programming language (with planning support) based on the Situation Calculus. This transformation uses explicitly formulated constraints that express dependencies between task actions and the robot itself. These are important at run-time and we cannot and do not want to consider them at planning time already. Thus we also alleviate the costs for planning.

2 Foundations

In the following, we briefly sketch the foundations our approach builds on. For one, that is the Situation Calculus and our robot control language ReadyLog, for another that is a formulation of temporal constraints.

2.1 Situation Calculus & ReadyLog

The Situation Calculus [3] is a sorted logical language with sorts situations, actions, and objects. Properties of the world are described by relational and functional fluents that change over time (situation dependent). Actions have preconditions, and effects of actions are described by successor state axioms. The world evolves from situation to situation, e.g., $s' = do(a, s)$ means that the world is in situation s' after performing action a in situation s. GOLOG [4] is a logic based robot programming (and planning) language based on the Situation Calculus. It allows for Algol-like programming but it also offers some non-deterministic constructs. It uses an evaluation semantics: $Do(\delta, s, s')$ means that executing program δ transforms situation s to s'.

There exist various extensions and dialects to the original Golog interpreter, one of which is ReadyLog [5]. It provides an online interpreter and integrates several extensions like interleaved concurrency, sensing, exogenous events, and online decision-theoretic planning (following [6]) into one framework. We use ReadyLog to specify our agents and the approach presented here is an extension to ReadyLog. As programs in ReadyLog represent task plans, we will use the term program from now on instead of plan.

2.2 Temporal Constraints

To formulate (temporal) constraints we obviously require a notion of (temporal) relations between actions (or more generally, between states). Since we are interested in constraints that should be easy to formulate for the designer we prefer a qualitative description of these relations. We consider this sufficient for most cases we intend to handle and spare computing explicit timing values. We therefore chose Allen's Interval Algebra [7] as our basis. For an overview on important relations in this algebra see Fig. 1. An example of a constraint that we

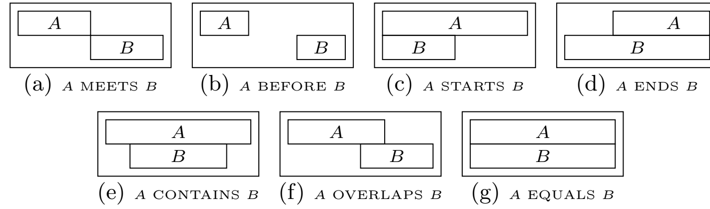


Fig. 1. Seven of Allen's interval relations.

want to formulate could be

$$calibrate_arm \text{ BEFORE } manipulate$$

to indicate that the manipulator has to be calibrated before we can actually use it. We are not the first to consider an interval formulation in Golog [8], however, our use is not targeted at flexible interval planning but more to formulate the constraints and augment a given program according to these.

2.3 Durative Actions

Usually actions are durative, i.e., they consume time. The original Situation Calculus only knows 'instantaneous' actions. There are, however, some extensions that we are going to adopt to represent durative actions [9, 10]. In these approaches, actions with a duration are considered *activities* that are bounded by *instantaneous* start/stop-actions. The fact that such an activity is currently being performed is indicated by a fluent for each activity. See Fig. 2 for an example.

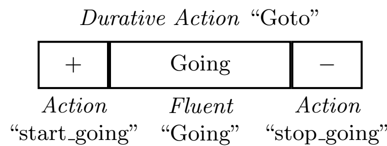


Fig. 2. Exemplary decomposition of a durative action

3 Approach

The general idea is to implement a program transformation process based on temporal constraints and the program to be performed. Fig. 3 depicts how we propose to integrate the components of our self-maintenance framework into existing agent controllers.

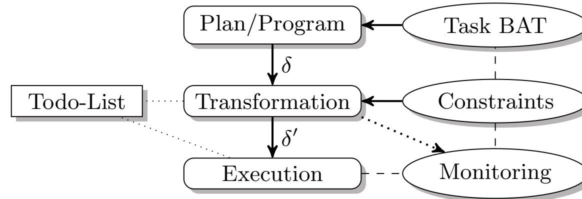


Fig. 3. Architectural overview of our approach

We propose to intervene between decision-theoretic planning, whose output is an executable program, and its online execution. Before any action of the program is performed in the real world, a self-maintenance interpreter checks whether there are unsatisfied constraints for this action. If such constraints are found, the program execution is delayed and the program is augmented with maintenance and recovery measures. The augmented program is only then passed on for execution. Depending on the constraint(s) the transformation also includes monitoring markers, e.g., making sure the *locomotion_module is off* throughout the execution of *manipulating* something.

3.1 Constraints

For this transformation to work, we need to make one important restriction, though. Since the self-management may not invalidate the program, we separate the task from the maintenance domain and restrict the constraints to only map elements from the latter to the former.

Our approach is similar to [11] who propose a framework for online plan repair and execution control based on temporal constraints. Their work is motivated by the same problems than ours, namely that "taking into account run-time failures and timeouts" requires online plan recovery, However, they rely on partial order planning and assume temporal flexible plans. Their objective is to execute a plan under resource and timing constraints by grounding time points at execution time. We, on the other hand, are interested in interleaving self-maintenance and task actions at execution time on a qualitative level. Time points in the Situation Calculus are only characterized by actions. Still, we borrow their notion of (non) preemptive actions and the idea that the system sends some form of report about action completion and the systems' components' states.

Our constraint syntax is $\mathcal{A} \otimes \mathcal{B}$ where

- $\mathcal{A}$ is from self-management domain only. It can be (a) a *instantaneous action* which corresponds to an interval end point, (b) a *durative action* that needs to be decomposed to its end points, or (c) a *fluent formula* that needs to be checked with respect to the interval.
- $\otimes$ is one of Allen's relations.
- $\mathcal{B}$ is from task domain only. The same cases as described above for $\mathcal{A}$ also apply for $\mathcal{B}$.

Table 1. Translation of a constraint to an order on situations

$\mathcal{A}$ BEFORE $\mathcal{B}$		Task ($\mathcal{B}$)		
		b	B	ψ
Mgmt ($\mathcal{A}$)	a	$a < b$	$a < B^+$	$a < \Delta_\psi^+$
	A	$A^- < b$	$A^- < B^+$	$A^- < \Delta_\psi^+$
	ϕ	$\Delta_\phi^- < b$	$\Delta_\phi^- < B^+$	$\Delta_\phi^- < \Delta_\psi^+$

3.2 Online Program Transformation

We transform the program (i.e., a plan generated by ReadyLog) using the set of constraints available for the next task action to be executed. The set of constraints is translated to a Constraint Satisfaction Problem (CSP) by resolving each constraint to an order on situations described by primitive or start/stop-actions. An example is given in Table 1. Small case letters denote instantaneous actions, capital letters stand for complex actions, and Δ_ϕ^- and Δ_ϕ^+ represent a fluent formula ϕ becoming false or true in a certain situation, respectively. The solution of the CSP then dictates the transformation. It inserts maintenance actions and monitoring markers at appropriate positions in the program.

3.3 Inheritance

It is an often seen bad practice to duplicate constraints for related actions. To alleviate this and provide a more convenient way of formulating the constraints, it should be possible to give constraints for actions classes, e.g., we would like to have an action inheritance about several move actions. Building on [12], we employ a modular BAT that allows for inheritance of constraints along the hierarchy of actions. See Fig. 4 for an example.

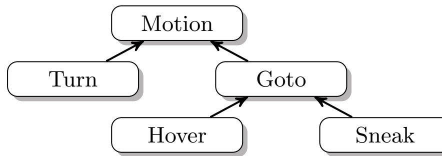


Fig. 4. Inheritance of constraints in a hierarchy of actions

3.4 A Simple Example

To clarify our approach we depict a simplistic example of the general process in the following. The single steps of this process are depicted in Fig. 5.

In Step 1 we show the state of affairs before our process is about to kick in. Then, as soon as the task program features an action that appears in any of the constraints, the CSP solve is triggered. The solution forces us to insert a `start_beep` action before we can actually execute `start_goto`. After executing

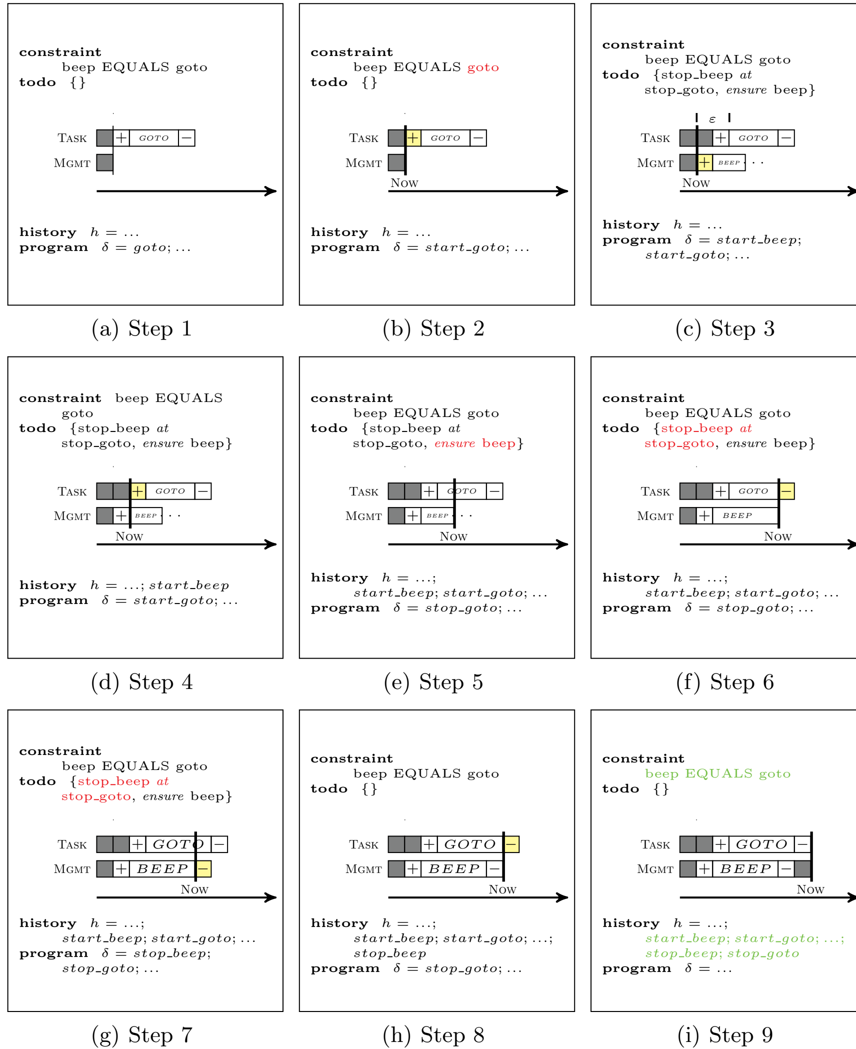


Fig. 5. Exemplary Maintenance Process

start_beep the execution of start_goto succeeds. Throughout the run-time of the goto action we ensure that beep is also running. Then, when stop_goto is about to be executed, when detect that we have to stop_beep. Only after we do this, stop_goto can actually be executed.

Note that in Step 3, when inserting the start_beep action we make use of something we call the ε-slot. We consider two actions happening simultaneously if they happen within a time span not exceeding the ε-slot. This is due to the fact that ReadyLog only supports *interleaved concurrency* [9] which executes two action sequences concurrently by interleaving them. This is opposed to *true concurrency* [13] where sets of actions may be executed 'truly concurrently' between any two situations.

4 Discussion

In this paper we sketched our approach to self-maintenance for autonomous robots controlled by ReadyLog. We modify a given program at run-time using explicitly formulated temporal constraints that relate self-maintenance actions with actions from the task domain. This way we achieve more robust and enduring operation and take care of maintenance when it is relevant: at execution time. Keeping our approach in one framework allows to use all of ReadyLog's features in maintenance and recovery.

In future work we will consider two extensions. *Explanation:* Since the robot knows which constraint(s) failed in a particular situation and it probably does not have means to take care of it itself the robot can at least exhibit to the user what went wrong. *Interaction:* Alternatively, if the robot can not handle a constraint itself (e.g., *no_emergency_off* while *drive*) but knows, that a human user could do, it can simply trigger an interaction, e.g., ask *"Could you please release my emergency button?"*.

References

1. Levesque, H., Lakemeyer, G.: Cognitive Robotics. Handbook of Knowledge Representation. Elsevier (2007)
2. Levesque, H., Reiter, R.: High-level robotic control: Beyond planning. a position paper. In: AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap. (1998)
3. McCarthy, J.: Situations, Actions, and Causal Laws. Technical Report Memo 2, AI Lab, Stanford University, California, USA (1963) Published in Semantic Information Processing, ed. Marvin Minsky. Cambridge, MA: The MIT Press, 1968.
4. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A Logic Programming Language for Dynamic Domains. Journal of Logic Programming **31** (1997) 59–83
5. Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains. Robotics and Autonomous Systems **56** (2008) 980–991

6. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence, AAAI Press / The MIT Press (2000) 355–362
7. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26** (1983) 832–843
8. Finzi, A., Pirri, F.: Flexible interval planning in concurrent temporal golog. In: Working notes of the 4th Int. Cognitive Robotics Workshop. (2004)
9. de Giacomo, G., Lespérance, Y., Levesque, H.J.: Congolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121** (2000) 109–169
10. Claßen, J., Hu, Y., Lakemeyer, G.: A Situation-Calculus Semantics for an Expressive Fragment of PDDL. In: AAAI'07: Proceedings of the 22nd National Conference on Artificial Intelligence, AAAI Press (2007) 956–961
11. Lemai, S., Ingrand, F.: Interleaving temporal planning and execution in robotics domains. In: AAAI'04:Proceedings of the 19th National Conference on Artifical Intelligence, AAAI Press / The MIT Press (2004) 617–622
12. Gu, Y., Soutchanski, M.: Reasoning about Large Taxonomies of Actions. In Fox, D., Gomes, C.P., eds.: AAAI'08: Proceedings of the 23rd National Conference on Artificial Intelligence. Volume 2., AAAI Press (2008) 931–937
13. Reiter, R.: Natural actions, concurrency and continuous time in the situation calculus. In: In Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96), Cambridge, Massachusetts, U.S.A. (1996) 2–13

Improving the Performance of Complex Agent Plans Through Reinforcement Learning

Matteo Leonetti and Luca Iocchi

Sapienza University of Rome
Department of Computer and System Sciences
via Ariosto 25, 00185
Rome, Italy

Abstract. Agent programming in complex, partially observable, and stochastic domains usually requires a great deal of understanding of both the domain and the task in order to provide the agent with the knowledge necessary to act effectively. While symbolic methods allow the designer to specify declarative knowledge about the domain, the resulting plan can be brittle since it is difficult to supply a symbolic model that is accurate enough to foresee all possible events in complex environments, especially in the case of partial observability. Reinforcement Learning (RL) techniques, on the other hand, can learn a policy and make use of a learned model, but it is difficult to reduce and shape the scope of the learning algorithm by exploiting a priori information. We propose a methodology for writing complex agent programs that can be effectively improved through experience. We show how to derive a stochastic process from a partial specification of the plan, so that the latter's perfomance can be improved solving a RL problem much smaller than classical RL formulations. Finally, we demonstrate our approach in the context of Keepaway Soccer, a common RL benchmark based on a RoboCup Soccer 2D simulator.

1 Introduction

Despite the great deal of research on planning over many years and in many different domains, planning in dynamic and uncertain domains is still a challenging task. In many applications, agents operate in highly dynamic and uncertain environments where most of the changes are not a consequence of the agent behavior. They usually have limited knowledge of the environment and noisy sensors. Many approaches rely on a transitional model of the domain; in these cases the knowledge about the environment is encoded and exploited for planning either offline or online.

As stated by Bonet and Geffner [2], creating a controller that maps observations into actions has been mainly achieved in three ways:

- the *programming* approach, where the controller is programmed by hand in a suitable high-level procedural language;
- the *planning* approach, where the controller is derived automatically from a suitable description the actions, sensors and goals;

– the *learning* approach, where the controller is derived from experience.

The programming approach allows to encode procedural information about how the task must be performed, but it makes improving the agent's behavior quite difficult, leaving little or no room for automation. The planning approach, on the other hand, allows to provide the agent with declarative knowledge about the environment, but is sensitive to inaccuracy of the model: in the class of environments we are considering, a declarative model cannot in general be able to foresee all possible events that can cause the plan to fail. This issue, especially in robotics, leads to the need for *execution monitoring* [14], that constitutes a whole research field. Finally, the learning approach can learn both a model of the environment and/or a *policy*, but it is particularly difficult for the designer to shape the search space, even when his/her knowledge could reduce the learning burden significantly.

In spite of many efforts to planning and learning in complex domains, hand-crafted plans still have a major role in many applications, even though they require a lot of effort from the designer and the results are of limited use in highly dynamic and uncertain domains.

Some relevant works in the direction of integrating a priori knowledge into a learning framework are present (cf Section 5). However, these works have limited applicability and do not scale to the class of problems we consider.

In this paper we introduce a novel use of Reinforcement Learning (RL) to improve planning from experience, while still allowing the designer to write a knowledge base or a set of plans. The proposed approach allows to convey prior knowledge to the agent in a straightforward way, more specifically in the form of a partially specified plan (or a set of plans). This is in contrast with standard approaches to learning to perform a specific task, which usually require a non negligible effort in the definition of the features of the environment to feed the learning algorithm, a careful choice of a function approximator and also the definition of proper actions.

The novelty of the approach is in the application of well established RL theory and methods in a novel learning state space, which is obtained directly from the plan, and it is considerably smaller with respect to standard formulations, thus not requiring function approximation. The proposed approach is targeted at all the "real world" applications in which the knowledge about the domain, even from the designer perspective, does not allow him/her to establish which plan (in a set of admissible ones) is going to perform better. In this context the agent behavior can automatically adapt during plan execution gaining benefit from experience.

To verify the effectiveness of our solution we implemented and tested it in the *Keepaway* domain [16, 8], a benchmark for learning algorithms at the edge of what RL can currently face. Our learning method could learn a behavior that significantly outperforms former results in the same setting and converges to the optimal solution several times faster.

2 Plan Representation

Our approach addresses the planning problem in complex, dynamic environments and is suited for reactive systems. In the rest of the paper we consider reactive plans represented as generic state machines, like state charts [6], in which every state corresponds to a set of *actions* and each transition corresponds to an *event*.

A *plan state* is a configuration of the machine that encodes the plan, as opposed to an environment state that is a configuration of the variables that represent the agent knowledge. Each plan state is associated the set of *actions* that is being executed in that point of the plan. Notice that the same set of actions may occur several times in different plan states. The state of the whole system is the Cartesian product of the plan and the environment state spaces.

An *event* is a general happening in the environment that can be whatever the agent is capable of detecting, for instance: a condition that becomes true, a message received from another agent, a timeout expired or a joint that reached its target position.

The representation of plans considered in this paper is based on a transition system defined over plan states and events. Such a transition system determines a set of plans, or *plan schemas*, as formally stated in the following definition.

Definition 1 (Plan Schema). *A Plan Schema is a tuple* $\langle S, s_0, F, E, \Phi, A, L, T\rangle$ *where:*

- S *is a finite set of plan states*
- s_0 *is the initial plan state*
- $F \subset S$ *is a set of final plan states*
- E *is a finite set of events*
- Φ *is a set of conditions*
- A *is a set of actions*
- $L : S \rightarrow \wp(A)$ *is a total labeling relation that maps plan states on actions*
- $T : S \times E \times \Phi \rightarrow S$ *is a transition relation augmented with a triggering event and a condition. For each* $s \in S$, $e \in E$ *and* $\phi \in \Phi$, ϕ *must entail the pre-conditions of all the actions in* $L(T(s, e, \phi))$

The underlying assumption is that all actions are indeed procedures that involve some actuators and then take time to execute. During the execution of an action the environment state changes continuously while the plan state does not. Indeed this representation does not model explicitly the agent's knowledge, but only the execution state of the plans.

The outcome of actions may be uncertain and we assume that a knowledge base (KB) exists such that at any moment it is possible to check whether or not it entails a certain condition. We also assume that appropriate modules keep such a KB updated with respect to the agent's perceptions.

In addition to events, edges are labeled with guard conditions that must hold for the edge to be enabled. The behavior of the machine is the following: the current state contains the currently executed set of actions which is performed until one of the events associated to the outgoing edges happens. Whenever such an event is sensed by the agent we say that the event *triggers* the transition which

makes it available for execution. For the edge to be actually enabled at that time another condition must be met, namely the guard of the transition must hold. When an edge is *triggered* (the associated event happens) and *enabled* (its guard condition holds) it is allowed to be followed and the next state represents the set of actions the agent is to execute next. If an action was present in the previous plan state and it is not in the next one that action must be terminated. On the other hand, if an action appears in the next state it must be started. All actions that are both in the previous and next plan state keep being executed. To make the operational semantic clearer we assume that all events are external (i.e., they cannot be generated by the machine itself) and transitions are instantaneous, so that no event can be lost during a transition execution. Final states are absorbing states that cannot be left once entered and determine the execution termination.

If the state machine is deterministic (it can never happen that two transitions are triggered and enabled at the same time), then the plan schema is actually a single plan since no choices are left to the executor and the entire behavior is specified. On the other hand, if the machine is non-deterministic the plan schema represents multiple plans and each non-deterministic choice is a fork among them. Nothing prevents different plans from sharing common paths and depart only where their behavior differs.

Such a state machine can easily represent any reactive, conditional plan with also while-loops. Transformation from plans in classical state-based representation to plan schemas as defined above is straightforward, since events may model post-conditions that become true and the guards can easily represent the pre-condition of the following action. But events can do much more, they can represent communication among agents (recall that an event can be associated to the receipt of a message), allowing the specification of multi-agent plan schemas. Events can also represent unexpected conditions (not necessarily the awaited post conditions), so that the plan may also account for interrupts. Finally, it is possible to easily extend the representation for hierarchical plans in which actions can be low level behaviors or state machines themselves, even if for this paper we limit the analysis to non-hierarchical plans. Thus, the proposed plan representation is quite general and we do not pose any restriction on the origin of plans, they can come from anything between automatic generation and handcrafting. The only assumption we require is that plans must be *correct*, in the sense that they should reach goal situations without violating action preconditions or domain constraints. Checking correctness of input plan schemas is outside the scope of the proposed approach.

2.1 Keepaway Soccer example

In order to make the plan representation and execution clear, we show a simple example borrowed by the *Keepaway Soccer* domain proposed by Stone and Sutton [16, 8]. Keepaway Soccer is a subtask of RoboCup Soccer in which one team, the *keepers*, must keep possession of the ball in a limited region as long as possible while another team, the *takers*, tries to gain possession. The task is *episodic* and one episode ends whenever the takers manage to catch the ball or the ball leaves the region.

Keepaway soccer retains some of the complexity of real world for the sensors are noisy, the environment is highly dynamic, also due to adversarial agents, and the communication among agents is limited.

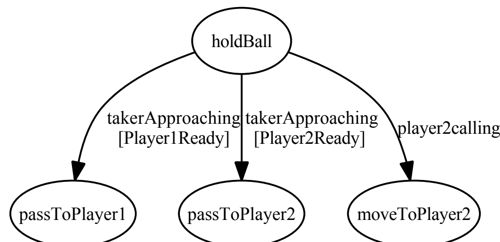


Fig. 1. An example of a simple plan. Actions label states, events and guards label transitions.

As an example, consider the plan schema in Figure 1. In the initial state the agent simply holds the ball until an event occurs. If *takerApproaching* happens two transitions are triggered. When at least one transition is triggered the post-conditions are checked, and if a transition is also enabled (its condition at that moment holds), it is followed setting the plan in a new state. Of course not necessarily there must be at least one enabled transition when an event happens and some events may be uncaught. In that case, the system remains in its current state waiting for the next event to happen. Notice that, if the guards *Player1Ready* and *Player2Ready* are not mutually exclusive, two transitions can be triggered and enabled at the same time. Thus, the transition system is non-deterministic and, in the same situation, two plans are available: the first one is $\langle holdBall, passToPlayer1\rangle$ while the second one is $\langle holdBall, passToPlayer2\rangle$.

In other words, in general, plan schemas are a compact way of representing multiple plans providing for different alternatives to achieve a goal.

3 Learning Framework

The learning framework is focused on exploiting the non determinism of a plan schema to make an informed choice.

Reinforcement Learning allows us to make use of experience to improve an agent's performance over time and seems a reasonable choice to achieve our goal. RL has been thoroughly studied within the MDP framework, since this framework provides a formal and neat mathematical notation for studying an important class of sequential decision problems. In traditional RL applications it is assumed that all relevant knowledge about an agent's environment can be encoded in a structure, usually a Markov Decision Process (MDP). Moreover, both in "model-free" and "model-based" RL techniques, it is assumed that even

though the agent might not know exactly what the structure of the MDP is (e.g. the transition matrix, etc), all sample observations are drawn from some underlying MDP. In the class of problems we are considering, however, assuming the existence of a fully observable MDP, or even trying to come up with a reasonable possible encoding for the states, which could somehow guarantee that the Markovian assumption is respected, might be infeasible. One reason for that is that it can be quite hard, or even impossible, to represent all the required information about other agents, their policies, unpredictable events, parallel action execution, unexpected changes in the task or in the environment, etc. In other words, it might be unreasonable or infeasible to assume that the task being solved can be well represented by an MDP. This is still true despite the sophisticated work on function approximation.

For this reason we rely on a generic knowledge base that reflects the beliefs of the agent about the environment, without building a dynamic model of it. In the following, we will define a stochastic decision process by deriving it from the plan and we will use this model to gather the experience to use in subsequent trials.

The state of the system is composed by both the state of the plan and the state of the environment but the latter is in general not completely known. The reward depends on how the state of the environment is perceived by the agent. In order to make a decision in non-deterministic choice points, we want to look forward in the plan having a value function associated with plan states, but not looking forward in the environment state space trying to predict the outcome of actions (i.e. the next environment state).

The plan executor must adhere to the state machine operational semantic as long as the choices are deterministic. Whenever a non-deterministic choice must be taken, the executor can refer to the value function to evaluate the alternatives and then exploit or explore as usual in RL.

3.1 Problem Definition

In order to properly characterize the stochastic process associated to the previously described state machine, and to set the proposed method in the RL framework, we define it in terms of a Semi Non-Markov Decision Process (SN-MDP), since the actions do not have the same duration and the process is in general non Markovian.

We first show the construction of the SNMDP with an example and then we provide its formal definition. Suppose that at some point of the plan schema you have a choice point like the one previously described (Figure 1). The nodes that allow for non-determinism (i.e., that have more than a transition associated with the same event, and whose guard conditions are not mutually exclusive) are split into a number of nodes equal to the constituent events of the condition. In the example, the event *takerApproaching* (abbreviated as *ta*) is associated to the conditions Player1Ready (*p1r*) and Player2Ready (*p2r*). This gives raise to four possible constituents, namely: only *p1r* is true, only *p2r* is true, both are true or none is. To the first three we associate a state and an arc from the

original *holdBall* state. The last situation, in which none of the conditions holds, translates into a loop on the *holdBall* state.

The resulting graph is represented in Figure 2. All the created edges correspond to the same non-deterministic action of the SNMDP reported as *ta*. Since it is caused by the perception of the event *ta*, the result of this action depends on the environment and cannot be chosen by the agent. In this section we make use of the term "action" as it is in the literature of stochastic processes when we refer to the SNMDP. Therefore, while an action in the plan schema is the actual intervention of the agent in the environment, an *action* in the SNMDP is an instantaneous transition available to the controller. An *action* in the SNMDP causes a change in the state of the process but cannot modify the state of the environment while this is the primary intention of an action in the plan schema.

Each node associated to a constituent of the conditions is connected to the action node containing the actions enabled by that constituent. In our example, *p1r* is connected to the node representing the action *passToPlayer1* (*pp1*), while *p2r* is connected to *passToPlayer2* (*pp2*) and *p1r*&*p2r* is connected to both. At this level the edges reaching different action nodes are associated to different *actions* of the SNMDP. The resulting graph has a choice point in the state *p1r*&*p2r* since in that case two actions are simultaneously available.

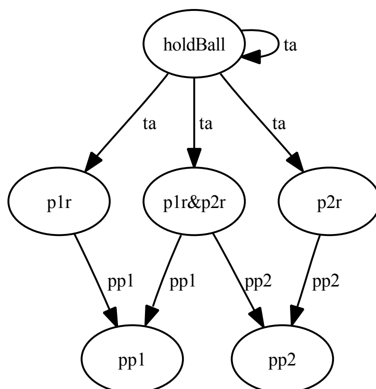


Fig. 2. Creation of the SNMDP. The original node with the action *holdBall* is split creating nodes to represent the conditions

The number of nodes in which a choice point in the original plan is split is exponential in the number of conditions. This is not surprising, and in the case of full observability and discrete state space this number would be equal to the number of states storing an entire Q-function. Nonetheless, the underlying assumption is that the domain is continuous and partially observable so that there is no notion of single state and considering single states or many small regions is not possible nor desirable. Hence, even if it is possible to consider

function approximation, it is not necessary for the number of nodes generated in practice.

To give a formal definition of the SNMDP we have informally previously introduced, we define the set $C_{cnd}(s,e)$ of the constituent events generated by overlapping conditions in a specific choice point (denoted as $\langle s,e\rangle$) of a plan schema $PS = \langle S, s_0, F, E, \Phi, T, A, L\rangle$ as follows: if there exist k conditions $\phi_1 \dots \phi_k$ and a state s_j s.t. $\langle s,e,\phi_i,s_j\rangle \in T$ for each $i \in \{1,\dots,k\}$ then $C_{cnd}(s,e) = \wp(\{\phi_k\}) \setminus \emptyset$ while $C_{cnd}(s,e) = \emptyset$ otherwise. In our example $C_{cnd}($ $holdball$, $takerApproaching) = \{\{p1r\}, \{p2r\}, \{p1r, p2r\}\}$.

Next, we define the set S_c of the states generated by condition overlapping in all choice points:

$$S_c = \{\langle s,e,cond\rangle | s \in S, e \in E, cond \in C_{cnd}(s,e)\}$$

In our example

$$\begin{aligned} S_c = \{ & \langle holdball, takerApproaching, \{p1r\}\rangle, \\ & \langle holdball, takerApproaching, \{p2r\}\rangle, \\ & \langle holdball, takerApproaching, \{p1r, p2r\}\rangle, \\ & \langle holdball, player2calling, \{true\}\rangle\} \end{aligned}$$

Those states constitute the second layer of Figure 2 except for the last one since the event *player2calling* has been omitted for simplicity. Finally, we also define a utility function S_c^e to select in S_c the states that are generated by a specific choice point as follows:

$$S_c^e(s,e) = \{\langle s,e,cond\rangle \in S_c | cond \in C_{cnd}(s,e)\}$$

Time has not been addressed yet so far. We consider time in discrete timesteps and actions can take multiple timesteps to complete. We use the following notation:

- t_k: the time of occurrence of the kth transition. By convention we denote $t_0 = 0$
- $s_k = s(t_k)$ where $s(t) = s_k$ for $t_k \le t \le t_{k+1}$
- $a_k = a(t_k)$ where $a(t) = a_k$ for $t_k \le t \le t_{k+1}$

We define a Semi Non-Markov Decision Process $SNMDP = \langle\ S_{sp}$, A_{sp}, P_{sp}, $r_{sp}\rangle$ such that:

- $S_{sp} = S_c \cup S$, is the state set. The set S_c is the set generated by overlapping conditions, whereas S is borrowed directly from the plan schema and accounts for *action states*, that is states that are not the result of a choice point split but are associated to actions in execution. The first and last layer of Figure 2 are an example of the states in S while the intermediate layer is an example of the states in S_c.
- $A_{sp} = \{a \in \wp(A) | \exists s \in S\ s.t.\ L(s) = a\} \cup E$, is the action set. The first part is the co-domain of the labeling function in the plan schema. We create an action for each possible set that labels the states of the plan schema. Notice

that those actions are deterministic and we give them the same name of their target state. In our example of Figure 1 the co-domain of labeling function is $\{\{holdBall\}, \{pp1\}, \{pp2\}, \{mp2\}\}$. In this example there is no parallelism, so all sets are singletons. You can spot the corresponding actions in Figure 2. The set E (events in the plan schema) is used to define the actions on which the agent has no control. These actions are non-deterministic and their outcome depends on the environment. Again, in Figure 2, *ta* is an example of such an action.

– $P_{sp}(s', a, \tau, s) = Pr(t_{k+1} - t_k \leq \tau, s_{k+1} = s' | s_k = s, a_k = a)$ is the probability for action a to take τ time steps to complete, and to reach state s' from state s
  - if $a \notin E$: the action is deterministic. An action that *is not in E* connects a state in S_c to the state in S (second to third layer in the example) labeled with the actions enabled by the condition in that state. Moreover, those actions do not reflect any change in the environment so they always complete in zero time. That is,

$$P_{sp}(s', a, \tau, s) \begin{cases} = 1 \text{ if } \exists\, s_i, e, \phi. \\ \qquad \langle s_i, e, \phi, s' \rangle \in T \\ \qquad \wedge\, s \in C^e_s(s_i, e) \\ \qquad \wedge\, L(s') = a \wedge\ \tau = 0 \\ = 0 \quad \text{otherwise} \end{cases}$$

    A state s is connected to the state s' by a iff s is a state generated by a condition constituent, it is linked to s' by the plan schema, and a is the label of that link.
  - if $a \in E$: the action is non-deterministic. These actions take the time spent in the previous state waiting for the event. An action that *is in E* connects a state in S to itself and to all the condition states that its split generate (first to second layer in the example). Therefore, events cannot connect all pairs of states, which translates into:

$$P_{sp}(s', a, \tau, s) \begin{cases} = 0 \text{ if } s \notin S\, \vee \\ \qquad s' \notin C^e_s(s, a) \cup \{s\} \\ = \int_H p(t_{k+1} - t_k \leq \tau, s_{k+1} = \\ \qquad s' | s_k = s, a_k = a, \boldsymbol{h}) \\ \qquad p(\boldsymbol{h})\, d\boldsymbol{h} \\ \text{otherwise} \end{cases}$$

    If a connection between s and s' through e exists according to the plan schema, the value of the transition function is the probability for the event a to happen in the state $\langle s, \boldsymbol{h} \rangle \in S \times H$ where H is the domain of (continuous) hidden variables. Since those variables are not observable, the sample distribution is the (hidden) underlying one marginalized over the hidden variables. This makes the stochastic process non Markovian due to partial observability.

– $r_{sp}(s', a, k, s_t)$ is the reward function. It is Markovian (but the total reward in general is not) and we define its value to be 0 if $a \notin E$. Therefore the

immediate reward is non-zero only for events. Since events can take time (the time spent waiting in the previous state for the event to happen) the reward is defined in terms of immediate rewards as:

$$r_{sp}(s', a, k, s_t) = \sum_{i=1}^{k} \gamma^{i-1} r_{t+i}$$

where the r_{t+i} are the instantaneous rewards collected during the action execution, and γ such that $0 \leq \gamma \leq 1$ is the discount factor. Instantaneous rewards are defined over perceptions, that is they are a function of the state of the knowledge base.

In order to define a decision problem, we establish a performance criterion that the controller of the stochastic decision process tries to maximize. As such, we consider the expected discounted cumulative reward, defined for a stochastic policy $\pi(s, a)$ and for all $s \in S_{sp}$ and $a \in A_{sp}$ as:

$$\begin{aligned} Q_\pi(s, a) &= E\{\sum_{i=1}^{\infty} \gamma^{i-1} r_i\} \\ &= \sum_{s' \in S_{sp}} \sum_{\tau=0}^{\infty} \pi(s, a) P_{sp}(s', a, \tau, s) \cdot \\ &\cdot \Big(r_{sp}(s', a, \tau, s) \; + \\ &+ \gamma^{\tau} \sum_{a' \in A_{sp}(s')} \pi(s', a') Q_\pi(s', a') \} \Big) \end{aligned} \tag{1}$$

where $A_{sp}(s)$ is the set of actions for which $P_{sp}(s', a, \tau, s) > 0$ for some value of τ. The optimal discounted reward function is defined as:

$$Q^*(s, a) = max_\pi \; Q_\pi(s, a), \; s \in S_{sp}, \; a \in A_{sp} \tag{2}$$

3.2 Learning Algorithm

Since the stochastic process is in general non Markovian, extra attention must be paid at the algorithm used to evaluate the expected reward of a given policy. Usual algorithms based on a value function for MDPs make use of temporal difference (TD) methods to compute the expected reward from a state onward. The actual proof of convergence for TD relies on the Markov property and, even if Sarsa(λ) can be quite robust to partial observability [9], it is in general not guaranteed to converge. It has also been shown that adding memory to the observations can solve some POMDPs [11] and the plan schema allows to add arbitrary memory: if the plan schema is a tree the whole history is considered, but loops can create any situation in between memory-less and full memory. Practically, Sarsa(λ) should converge to a policy that, even if suboptimal, can allow for behavior improvement. A sound algorithm for the general case is MCESP by

Perkins [13], and good other candidates can be found in policy search methods, whose evaluation on our framework we leave for future work. For a brief review of results we can leverage, please refer to Section 5

The value function, that is the cumulative discounted reward from each state executing each action onward, will converge to the *expected value* of the reward influenced by the experience. It might happen that a choice point in the stochastic process corresponds to a region of the actual state space in which no action is in most of the cases better than any other. In such a case the value of all the available actions in that choice point would average out each other giving no reliable estimation of the expected reward. For this reason, a method (such as the aforementioned MCESP and Sarsa(λ)) that performs some form of Monte Carlo update must be preferred, so that is does not spoil the estimation of the former states. If the available knowledge does not allow to separate the conflicting regions in the actual state space, the agent cannot do any better.

4 Experimental Evaluation

The learning approach described in this paper has been tested in *Soccer Keepaway* on the 3 vs 2 task, i.e. with three keepers and two takers. Although the agents learn separately and there is no communication involved in the task, Keepaway is still a multi-agent task since the agents share the reward signal and each agent's action has an impact on all the others. Thus, credit assignment is particularly difficult since the reward for the whole *team behavior* is received by each agent as if it was its own.

In our work, we focus on the keepers and leave the takers' behavior to their predefined policy that consists in both of them following the ball. We refer to Stone at al. [16] and especially to the more recent work by Kalyanakrishnan and Stone [8] as representatives of the "RL way" to face Keepaway Soccer and we show our methodology applied to this task. As in that last reference, we consider the problem of learning both a behavior for the agent in possession of the ball and a behavior for the agents that are far from the ball. This is an additional challenge since the two behaviors interact making credit assignment even more problematic.

The first step consists in devising a proper set of actions. We consider three actions for the agent closer to the ball and three for the other two agents. The actions available to an agent close enough to kick are *holdBall* that just keeps possession of the ball, *passToCloser* that passes the ball to the agent that is closer to the kicker and *passToFree* that passes the ball to the agent whose line of pass is further from the takers. The first action is clearly wrong since a player cannot hold the ball indefinitely without being reached by the takers but we added it as a control, to make sure that our algorithm assigns the correct value to it and never chooses that action after convergence. The actions available to the agents far from the ball are *searchForBall* that just turns in place, *getOpen* that is the default handcoded behavior provided by the framework described as "move to a position that is free from opponents and open for a pass from

the ball's current position", and *goToTheNearestCorner* that goes to the corner closer to the agent.

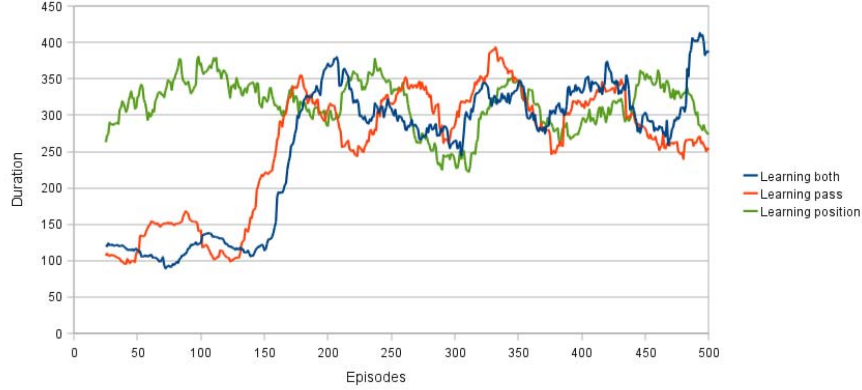


Fig. 3. A representative run of experiments. The x axis is the number of episodes in the run while y axis is the hold time in tenths of seconds

After the definition of the available actions we create a plan schema to accommodate our choice points. The entire plan schema used in these experiments is shown in Figure 4. States labeled with *noaction* have the empty action set associated, while *noevent* is a special event that takes zero time. This event has no impact on learning but it allows to add states in the plan schema convenient for representation and readability. Similarly, when no condition is indicated the guard of that edge is assumed to be *true*, i.e. the condition that is always fulfilled. Again, this is just syntactic sugar and does not affect the method. The leftmost node is the initial state, the control flows from left to right and it reaches the rightmost node within a simulation server cycle. In each cycle the agent must send a command to the server, thus performing an action, therefore every path from the leftmost to the rightmost nodes contains exactly one action. All of the conditions except those that guard the edges with event *takerApproaching* are mutually exclusive and leave no choice to the executor. As already mentioned, in the case of the passing actions since all three of them are triggered by the same event (*takerApproaching*) and their conditions (*true*) always hold at the same time, there is a non-determinism that can be exploited to make an informed choice. *TakerApproaching* is triggered when the agent perceives that a taker is closer than a certain threshold, *actionPerfomed* happens when the previous actions has queued its command for the server, and the conditions are similarly defined over state variables. In a similar way, the three choices for the positioning behavior *getOpen*, *searchForBall* and *goToTheNearestCorner* are taken into account when the player is not the one closest to the ball. In this setting even the simple Sarsa algorithm converged to the optimal solution.

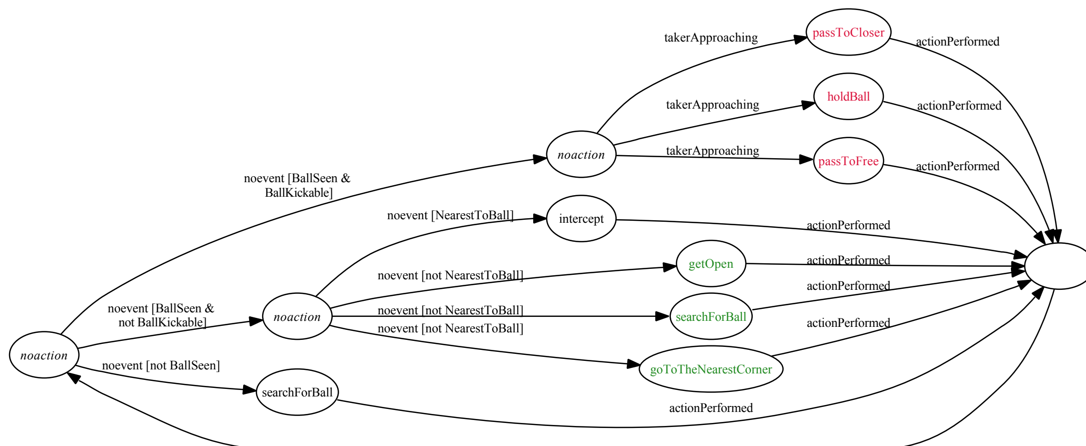


Fig. 4. The plan schema for a keeper with choice points on the passing and positioning behaviors

We performed different trials learning the two behaviors simultaneously and also the passing behavior and the positioning behavior separately. Our implementation used a greedy policy with optimistic initialization, a value of $\alpha = 0.3$ and $\gamma = 1.0$ which is sound since the task is episodic and the cumulative reward is limited. The reward signal is given by the duration of the episode: at every server cycle each agent receives a reward of $1/10$ of second. Even if the immediate reward is the same after every action, the cumulative reward depends on the previous choices and on the behavior of all the agents resulting in being highly non Markovian. Indeed what each agent aims maximize is actually the *team* performance. A representative trial is plotted in Figure 3 where each point is the average over a window of 50 episodes. With our approach we obtain the optimal behavior (that can be manually verified to be when *passToFree* and *goToTheNearestCorner* are chosen) after about 200 episodes in the case of learning both passing and positioning, with an average episode duration after learning of about 31 seconds. In previous works [16, 8] the best results are about 16 seconds of hold time and they take tens of thousands of episodes to be learned. We also show the learning curves of the single behaviors separately when coupled with the optimal choice for the other one. It appears that the passing behavior is the harder to learn, while positioning is learned in the first few episodes. Moving to the nearest corner without the ball then proves to be the crucial action that outperforms its alternatives quite quickly. In Figure 5 we show the box-plot of the distributions of the episodes' duration before (random behavior) and after learning. Both plots are drawn from 250 runs. We first used the Shapiro-Wilk normality test to check whether the two samples come from a normally distributed population, which turned out to be false for the second sample. Then, we used the non-parametric Mann-Whitney U test to confirm that the two samples do not (are extremely

unlikely to: $p = 5.7271 * 10^{-26}$) come from the same distribution. This means that the learning algorithm has indeed had a statistically significant impact on the duration of the episodes. The domain proved to be extremely noisy and the variance of both the samples is quite noteworthy.

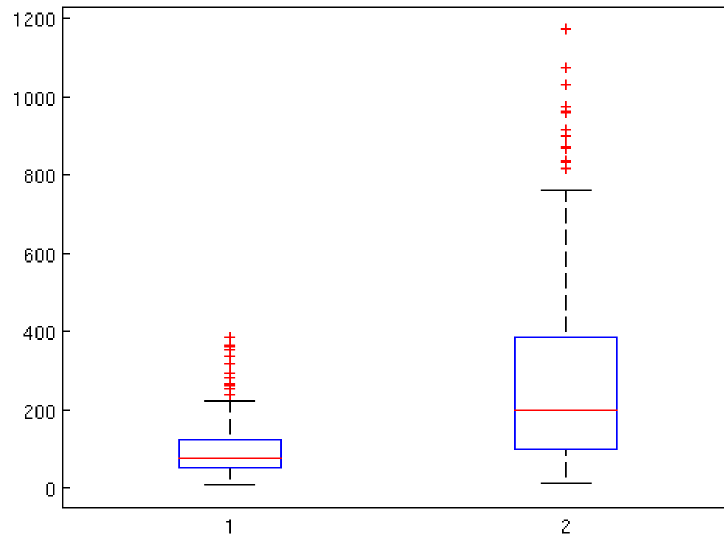


Fig. 5. Box-plot of the distributions of the episodes' duration (1) before and (2) after learning.

At the cost of devising a few (quite simple indeed) actions, and creating a partially specified plan exploiting the designer's knowledge about the task, we obtained a performance twice as high as the previous works in a number of learning episodes thousands of times smaller even with an algorithm as simple as Sarsa. The burden of creating the state representation and tailoring the function approximation for traditional RL is quite remarkable compared to the effort required of a designer to define such a plan schema and implement those actions. Also notice we made little use of perceptions, since conditions and events consider only a few aspects of the environment.

5 Related Work

Our work can be considered as part of the field of Hierarchical Reinforcement Learning (HRL). The overall idea of HRL is the ability of expressing constraints

on the behavior of the agent so that knowledge about the task and the environment can be exploited to shrink the search space. The optimal policy in this setting is the best one among those compatible with the constraints. The approaches closest to ours are Parr and Russell's HAM [1] and Andre and Russell's ALisp [10]. A similar approach can also be found in the field of symbolic agent programming, as this is the case of Decision Theoretic Golog (DTGolog) [3, 5]. All of the mentioned works allow to partially define the agent behavior in a high-level language (hierarchical state machines, Lisp and Golog respectively) and learn (or compute, in the case of DTGolog) the best behavior when this is not specified. While we share their motivation, our work departs from the previous ones in at least two aspects: (1) the formalism we adopt allows for dealing with reactive plans, non atomic actions, and continuous state spaces: these aspects are strictly related, leading to the representation of actions as states (instead of transitions) and to the need for events to both determine the end of an action and to mark those perceptions among the continuous infinity of possible ones that the agent should pay attention to; (2) even more importantly, we do not assume the existence of a Markov process as the underlying environment (an assumption common to all of the previous methods), but we derive a controllable process directly from the plan. Notice that the implementation of Kalyanakrishnan and Stone [8] fixes the behavior of the agent everywhere except for the two aspects they want to learn actually implementing a HAM. Therefore, the performance evaluation we carried can also be considered with respect to HAMs.

As a result of giving up the Markov assumption, and since partial observability is an aspect of common applications we consider in our approach, the control of the stochastic process resulting from the plan schema belongs to the class of problems usually referred to as with *hidden states*. The most general formulation of learning with hidden state are Partially Observable Markov Decision Processes (POMDP) [4]. Most of the methods for POMDPs attempt some state estimation, while we do not.

The stochastic process resulting from the observations in a POMDP (ignoring the underlying state space) is non-Markovian, and it is in some sense similar to the process we generate. The literature about N-MDPs is not as extensive as the one about MDPs, nonetheless some interesting results have been proved. A review of the available results is beyond the scope of this section, but we refer to the analysis by Pendrith and McGarity [11] and Singh et al. [15] about the characteristics of optimal policies in N-MDP and the effects of applying *direct* RL to them. An algorithm sound in the general case (although potentially suboptimal) is provided by Perkins [13] and the role of the exploration policy in the convergence of Sarsa and Q-learning is pointed out by Perkins and Pendrith [12]. Moreover, while examples can be constructed to prove that some (extremely simple indeed) implementation of direct RL on N-MDPs can diverge, there are promising empirical results about eligibility traces and partial observability [9]. Thus, although a comprehensive study about classes of N-MDPs and the traditional RL algorithms able to cope with them is still an issue, the lack of general results about convergence in non Markovian environments does not imply that those methods are doomed, it simply entails that further work is still needed.

We have shown through experiments that standard RL on the SNMDP built from a plan schema as shown before converges in a well-known (quite difficult) benchmark domain.

6 Discussion and Future Work

In this paper we have presented a methodology to write agent programs and to improve the agent's behavior through learning for a quite general category of plans. We have defined a proper controllable stochastic process deriving it from a partial specification of plans, in order to use it as a model for RL algorithms to improve the performance of the agent through experience. Finally, we have discussed the applicability of the available RL algorithms to the particular class of stochastic processes that our method generates and we have proved the effectiveness of our approach on a widely adopted test bed.

In our work we used *actions* as procedures that are usually referred in Hierarchical RL as *skills*. A popular model for skills is provided by the *option* framework [17] in which options and basic actions can be simultaneously considered. The role of options and their utility has been regarded as arguable [7] when the focus is on optimality. Nonetheless, in the class of problems we are considering optimality is quite difficult to achieve anyway, and our approach semi-automatically combining a set of handcoded skills proved to be more effective than *flat* RL which, even though is supposed to eventually reach the optimal behavior, was outperformed making use of a number of training episodes several orders of magnitude lower. In this context, having a good set of reusable skills to combine is of the utmost importance, and the work on temporal abstraction to create them automatically can profitably be integrated with our method providing different levels of intervention. Thus, where flat RL suffers in scaling up the search for a global optimum, the role of skills can be less arguable.

We have demonstrated in this paper a simple application of our approach to Keepaway Soccer limiting for simplicity the number of actions, and by no means obtaining the best behavior achievable. Our method is conceived to scale up to domains in which RL has not yet been successfully applied. In future work we plan to face more complicated settings possibly defining a new benchmark for hierarchical methods. We also plan to extend our formalism to multi-agent plans, exploiting events to represent message delivery or, more in general, coordination signals, thus learning team behaviors and coordination. Finally, we will further investigate the properties of the RL algorithms when applied to the stochastic process generated from a plan schema, and how to make use of the structure of plan schemas to obtain processes that favor convergence and/or optimality.

References

1. D. Andre and S. Russell. Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems*, pages 1019–1025, 2001.
2. B. Bonet and H. Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

3. C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the National Conference on Artificial Intelligence*, pages 355–362. AAAI Press / The MIT Press, 2000.
4. A. R. Cassandra. *Exact and approximate algorithms for partially observable markov decision processes*. PhD thesis, Providence, RI, USA, 1998. Adviser-Kaelbling, Leslie Pack.
5. C. Fritz and S. McIlraith. Decision-theoretic golog with qualitative preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK*, 2006.
6. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
7. N. K. Jong, T. Hester, and P. Stone. The utility of temporal abstraction in reinforcement learning. In *The Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008.
8. S. Kalyanakrishnan and P. Stone. Learning Complementary Multiagent Behaviors: A Case Study. In *Proceedings of the 13th RoboCup International Symposium*, 2009.
9. J. Loch and S. Singh. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 141–150. Morgan Kaufmann, 1998.
10. B. Marthi, S. J. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 779–785. Professional Book Center, 2005.
11. M. D. Pendrith and M. McGarity. An analysis of direct reinforcement learning in non-markovian domains. In J. W. Shavlik, editor, *ICML*, pages 421–429. Morgan Kaufmann, 1998.
12. T. Perkins and M. Pendrith. On the existence of fixed points for Q-learning and Sarsa in partially observable domains. In *Proceedings of the Nineteenth International Conference on Machine Learning*, page 497. Morgan Kaufmann Publishers Inc., 2002.
13. T. J. Perkins. Reinforcement learning for pomdps based on action values and stochastic optimization. In *Eighteenth national conference on Artificial intelligence*, pages 199–204, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.
14. O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.
15. S. Singh, T. Jaakkola, and M. Jordan. Learning without state-estimation in partially observable Markovian decision processes. In *Proc. of 11th International Conference on Machine Learning*, 1994.
16. P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.
17. R. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.