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

Dagstuhl Seminar Proceedings 10081
Cognitive Robotics
http://drops.dagstuhl.de/opus/volltexte/2010/2636

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

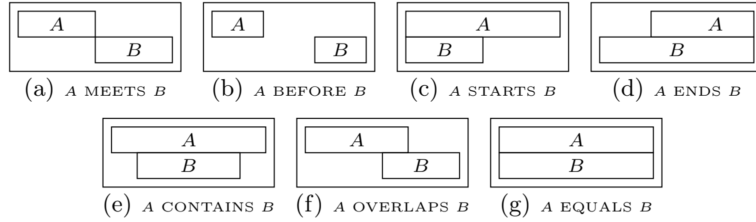


Fig. 1. Seven of Allen's interval relations.

want to formulate could be

$$\textit{calibrate_arm} \text{ BEFORE } \textit{manipulate}$$

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.

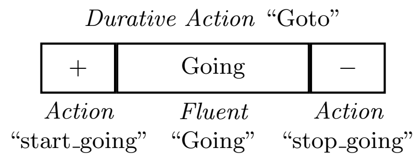


Fig. 2. Exemplary decomposition of a durative action

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.

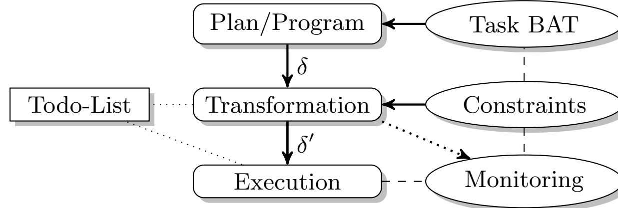


Fig. 3. Architectural overview of our approach

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.

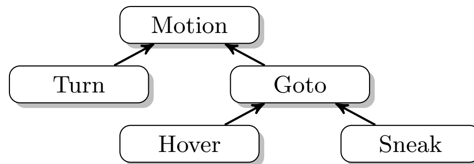
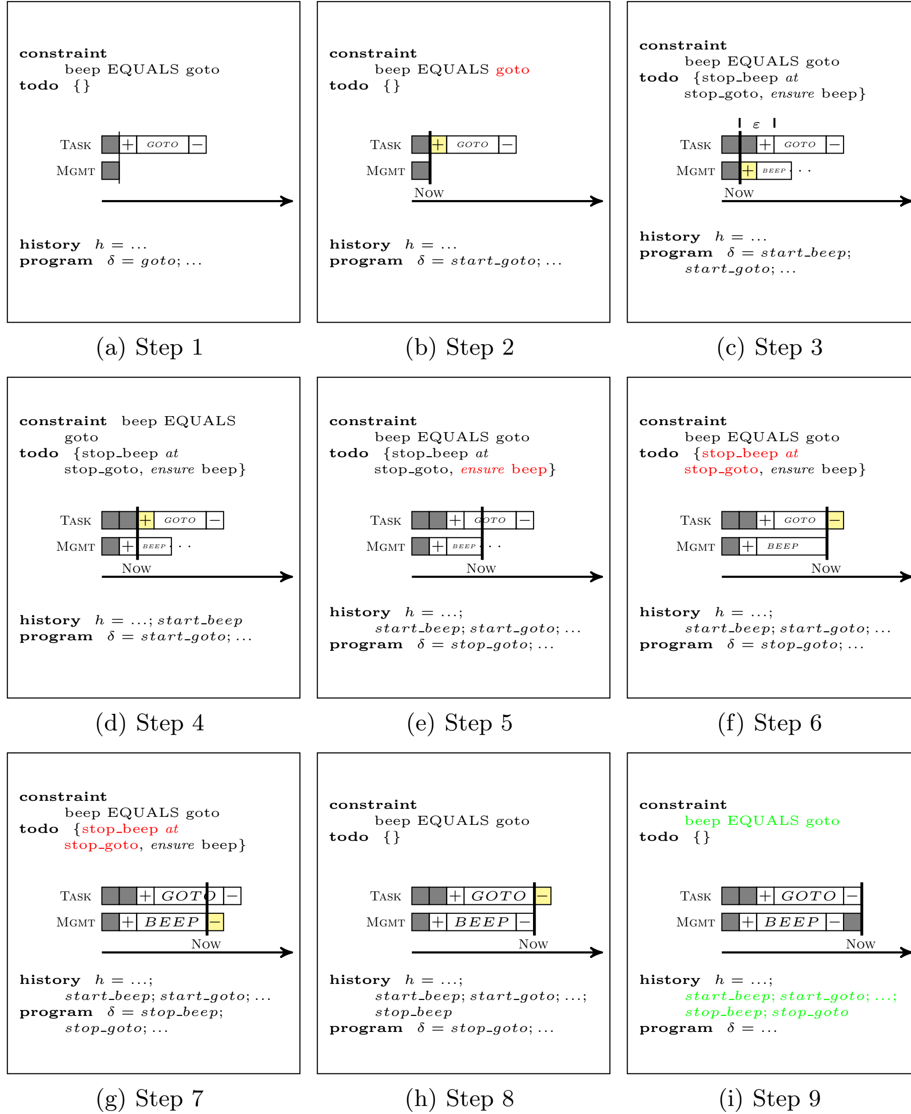


Fig. 4. Inheritance of constraints in a hierarchy of actions

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



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

(g) Step 7 (h) Step 8 (i) Step 9

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