Self-Maintenance for Autonomous Robots controlled by READYLOG

Stefan Schiffer Andreas Wortmann Gerhard Lakemeyer

Knowledge-Based Systems Group
RWTH Aachen University, Aachen, Germany
andreas.wortmann@rwth-aachen.de
(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 plan. This requires the robot to know about the (internal) states of its components, constraints that restrict execution of actions and 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 qualitative temporal constraints. Afterwards, a 'guarded' execution of the transformed program results in more robust behavior.**

I. INTRODUCTION

To arrive at truly intelligent agents such as robots, many issues need to be adressed such as perception, locomotion, manipulation, human-robot interaction, planning and reasoning. In this paper we are concerned with the latter in the sense of *cognitive robotics* as introduced by the late Ray Reiter, meaning "the study of the knowledge representation and reasoning problems faced by an autonomous robot (or agent) in a dynamic and incompletely known world" [9].

The central effort of Reiter's vision [10] "is to develop an understanding of the relationship between the knowledge, the perception, and the action of such a robot". This is outlined by several questions the research in *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 the intersection of these questions, 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 in an external process and in advance, otherwise they fail during plan execution.

We propose a *constraint-based self-maintenance framework*, which enables an agent to monitor its self-maintenance requirements during program execution. Whenever the framework determines unsatisfied requirements, appropriate recovery measures are performed online. This behaviour increases agent autonomy and robustness. We achieve this by adding a program transformation step in READYLOG, a logic-based robot programming language based on the Situation Calculus. This program transformation uses explicitly formulated constraints that express dependencies between task actions and the robot itself. These are only important at run-time and we cannot and do not want to consider them at planning time already. Thus, by decoupling the run-time requirements, we also alleviate the costs for planning.

II. FOUNDATIONS

In the following, we briefly sketch the foundations of our approach. For one, that is the Situation Calculus and our robot control language READYLOG, for another that is a formulation of temporal constraints.

A. *Situation Calculus* & READYLOG

The Situation Calculus [12] 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 [11] 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. A *Basic Action Theory* (BAT), which is a set of axioms describing properties of the world, axioms for actions and their preconditions and effects, and some foundational axioms, then allows for reasoning about a course of action. To run a program the original GOLOG uses an evaluation semantics: $Do(\delta, s, s')$ means that executing program δ transforms situation s to s'. The program is evaluated as a whole and then executed in one go.

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 [2]) into one framework. In READYLOG a program is interpreted in a step-by-step fashion where a transition relation defines

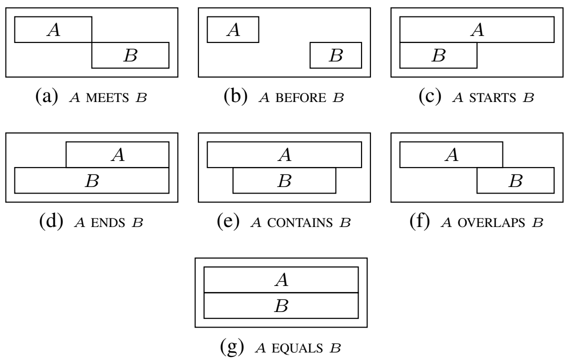

Fig. 1. Seven of Allen's interval relations [1]. We omitted the inverse relations here for simplification purposes.

the transformation from one step to another. In this so-called *transition semantics* a program is interpreted from one configuration $\langle \delta, s \rangle$, a program δ in a situation s, to another configuration $\langle \delta', s' \rangle$ which results after executing the first action of δ, where δ' is the remaining program and s' the situation resulting from the execution of the first action of δ.

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*.

B. Temporal Constraints

To formulate temporal constraints between actions 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 [1] as our basis. For an overview on important relations in this algebra see Fig. 1. An example of a constraint that we 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 [6]. However, our approach is not targeted at flexible interval planning but more to formulate the constraints and augment a given program according to these.

C. 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 [4], [3]. 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.

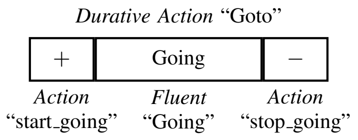

Fig. 2. Exemplary decomposition of a durative action

III. 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 the existing agent controller.

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. It possibly also requires a list of commitments to actions in the future.

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 former to the latter.

A. Constraints

Our approach is similar to [8] who propose a framework for online plan repair and execution control based on temporal constraints. Their work is motivated by the same problems as ours, namely that "taking into account run-time failures

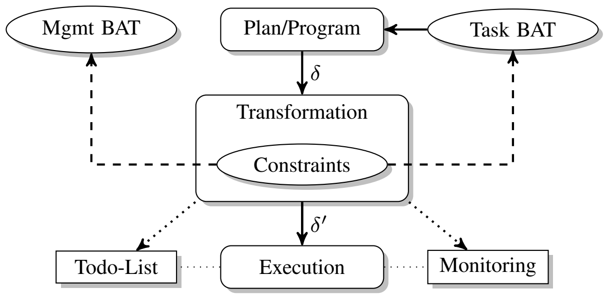

Fig. 3. Architectural overview of our approach. The program δ, which only uses the *Basic Action Theory* for the task, is passed to our transformation process. This process uses constraints, which link elements from the Task BAT to elements from the self-maintenance domain. The latter are specified in the Mgmt BAT. The transformation yields a modified program δ' which is passed on for execution together with instructions for monitoring and commitments to future action. These commitments are maintained in a todo-list to ensure that they will eventually be satisfied.

TABLE I
TRANSLATION OF A CONSTRAINT TO AN ORDER ON SITUATIONS

$\mathcal{A}$ BEFORE $\mathcal{B}$		Mgmt ($\mathcal{B}$)		
		b	B	ψ
Task ($\mathcal{A}$)	a	$a < b$	$a < B^+$	$a < \Delta^+_\psi$
	A	$A^- < b$	$A^- < B^+$	$A^- < \Delta^+_\psi$
	ϕ	$\Delta^-_\phi < b$	$\Delta^-_\phi < B^+$	$\Delta^-_\phi < \Delta^+_\psi$

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.

Our constraint syntax is $\mathcal{A} \otimes \mathcal{B}$ where

$\mathcal{A}$	is from the task domain. 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 the self-management domain. The same cases as described above for $\mathcal{A}$ also apply for $\mathcal{B}$.

B. 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 I. 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 schedules maintenance actions and augments the program with appropriate *occurrence* and *persistence commitments*, i.e., commitments to future situations. Before we detail the construction of the CSP in Sec. III-B.2, we first elaborate on the concept of commitments mentioned above.

1) Commitments: There are maintenance constraints that require a commitment to future points in time. Consider, for example, the constraint A BEFORE B, which denotes that B has to be performed sometime after B or the constraint A EQUALS B, which, amongst others, demands that B has to stay active until A terminates. The only way to handle such commitments so far is to introduce auxiliary fluents that record these commitments. To determine the value of these fluents we would need to use regression [13], which traces the fluent's value back to the initial situation and applies all changes made since then up to the current situation. As regression is quite expensive with an increasing number of actions performed already, we follow a different approach. We extend the READYLOG language with two sets of conceptually different commitments: (i) *occurrence commitments*, which denote that some action has to be performed in the future, and (ii) *persistency commitments*, which denote that some situation calculus expression holds for some timespan, e.g., for a subsequence of a situation term.

a) Occurrence Commitments: Occurrence commitments $\mathcal{O}$ are commitments about the future program and ought to ensure that a certain action happens in the future and under certain circumstances. We consider four forms of occurrence commitments which represent the different circumstances under which we may require an action to be performed in the future.

- *sometime* (β) denotes that action β has to be performed some time in the future,
- *after* (α, β) denotes that β has to be performed some time in the future, but not before the next occurrence of action α.
- *within* (β, N) denotes that β has to be performed within the next N actions.
- *then* (α, β, N) denotes that β has to be performed within the next N actions, after the next occurrence of α.

The self-maintenance interpreter updates $\mathcal{O}$ according to the constraints involved. In any future step, it also needs to be notified about the existence of actions to be integrated. We therefore introduce a new construct TODO (α) to the READYLOG language which causes both, updating $\mathcal{O}$ according to the last action α performed and managing to check and schedule an action from $\mathcal{O}$ if applicable.

b) Persistency Commitments: Persistency commitments $\mathcal{P}$ denote that some situation calculus expression needs to hold for some timespan. These commitments follow similar ideas as the occurrence commitments, i.e., a set of situation calculus expressions should hold under certain circumstances in the future. As satisfying these commitments is considered vital to program execution, the self-maintenance interpreter may terminate the program if such a persistency commitment is violated.

- *between* (α, β, φ) denotes that the expression φ has to hold between the next occurrence of α and the first subsequent occurrence of β.
- *until* (β, φ) denotes that φ has to hold until the next occurrence of β.

The corresponding construct MONITOR follows the same idea as the TODO construct above, i.e., the persistency commitments are updated relative to the last action performed and then the resulting persistency commitments are checked relative to the current situation. We introduce two auxiliary relations UPDATEMONITOR $(\alpha, \mathcal{P}, \mathcal{P}')$ and CHECKMONITOR $(s, \delta', \mathcal{P})$ which perform these tasks.

2) Constructing the CSP: If, at any point in program execution, we have more than one constraint associated with the next action α, we have to schedule the corresponding maintenance actions appropriately. We do so by using transformation rules which we will present in the next section.

The constraints satisfaction problem that we construct is a fixed length CSP. It is built over variables $\{t_{\alpha_1}, \ldots, t_{\alpha}, \ldots, t_{\alpha_k}\}$, where each t_{α_i} represents a time-step in the sequence of actions induced by the maintenance constraints. The number of variables, i.e., the length of the CSP, is infered from the transformation rules. For every action that needs to be scheduled we need one time-step, hence one variable, and we need one time-step for the task action α itself. If there are occurrence commitments of types *within* (β, N) or *then* (α, β, N), these are also added to the CSP by adding additional variables t_{α_j} for each action required in any of these commitments. The domain of all these variables are the natural numbers $[1, 2, \ldots, n]$, where n is the length of the CSP, namely, the number of actions to be scheduled around the task action α plus one (for α itself). The restrictions on the variable assignments are taken from the transformation rules and the commitments. This completes the construction of the CSP. The solution to the CSP is an assignment of values to the variables t_{α_i}. It represents an ordering on the actions to be scheduled and it can then be translated to the sequence of actions $[\alpha_{i_1}, \ldots, \alpha_{i_n}]$.

3) Transformation Rules: The illustration of our maintenance constraints in Table I was a slight simplification. As already mentioned, we actually use a set of transformation rules per constraint. A transformation rule is a six-tupel $(A \otimes B, \alpha, \Phi, \mathcal{C}_{CSP}, \mathcal{O}, \mathcal{P})$, where

- $A \otimes B$, is a (maintenance) constraint as described in Sec. III-A
- $\alpha \in A^P$, i.e., an action from the task domain
- Φ is a situation calculus formula qualifying the scope of the transformation rule, e.g., apply different rules depending on whether a fluent F holds or not.
- $\mathcal{C}_{CSP}$ is a multi-set of restrictions of the form $T \times T \times \Psi$, where T is the set of CSP variables and Ψ is an arithmetical or logical expression on these variables. For example, $(t_i, t_j, \Psi_{i,j})$ denotes that t_i und t_j are constraint by the expression $\Psi_{i,j}$.
- $\mathcal{O}$ is a set of occurence commitments
- $\mathcal{P}$ is a set of persistency commitments

The transformation rules have to be read as follows. If there is a maintenance constraint associated with the next task action α to be executed in situation s, and the condition Φ holds, then (1) the restrictions $\mathcal{C}_{CSP}$ have to be met in the CSP, that is built, and (2) the CSP has to also meet both, the occurence commitments $\mathcal{O}$ and the persistency commitments $\mathcal{P}$. That is to say, every set of pairwise restrictions on variable assignments $\{(t_i, t_j) \mid \Gamma_{i,j}\}$ in the final CSP is made up of the Ψ in $(t_i, t_j, \Psi_{i,j})$ from $\mathcal{C}_{CSP}$. As an example, consider the constraint A MEETS B with transformation rule $(A \text{ Meets } B, A^-, \text{True}, \{(t_{A^-}, x_{B^+}, t_{A^-} \approx_\varepsilon t_{B^+})\}, \emptyset, \emptyset)$. If the next action to be performed is A^-, and the formula Φ holds in that situation, this rule causes the variables t_{A^-}, t_{B^+} to be added to the CSP with the restriction $t_{A^-} \approx_\varepsilon t_{B^+}$ on their ordering. It also causes the occurence commitments to be updated with $\mathcal{O}$, and the persistency commitments to be updated with $\mathcal{P}$.

C. Extended Configurations

The transition semantics mentioned in Sec. II so far described transitions over configurations of the form (δ, s). This does not allow iterative modifications of our commitments $\mathcal{O}$ and $\mathcal{P}$ without introducing procedural notions of variable assignments. This seems indesirable. Therefore, we extend the transition semantics of READYLOG from configurations (δ, s) to configurations $(\delta, s, \mathcal{O}, \mathcal{P})$ which include both, occurence commitments $\mathcal{O}$ and persistency commitments $\mathcal{P}$. Then, in each transition, i.e., in one step of the interpreter, we can update not only the program and the situation but the commitments needed for successful self-maintenance as well.

D. Special Features

Our approach features two particularities, namely (non-) preemptive actions and constraint inheritance, which we elaborate on in the following.

1) (Non-)Preemptive Actions: By introducing a decomposition of complex actions (i.e., activities) into an action starting the activity and an action stopping it, we skipped the fact that some of the actions that an agent can take may not be terminated by the agent itself in a meaningful way. Following [8], we use their notion of *preemptive* and *non-preemptive* actions, i.e., we distinguish between actions which can be terminated at any time without ruining the outcome and actions which are expected to fail if terminated early. We will also borrow the idea that the underlying execution system sends some form of report about the end of *non-preemptive* actions.

Within a coffee delivery scenario we might consider a self-maintenance action *beep*, that makes an annoying sound while the agent transports coffee, to be preemptive, while the task action *pickup* clearly is not. Since we want to treat both types of action similarly during program transformation, we introduce a construct $\text{waitForEnd}(A, \tau)$. It takes a complex action A and a deadline τ and terminates the action A if it is preemptive. If the action is non-preemptive, it blocks the program execution until either the execution system signals that the action has finished or the deadline is met, at which point program execution is aborted.

2) 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 define constraints for classes of actions, e.g., we would like to have an action inheritance for several move actions. Building on [7], we employ a modular BAT that allows for inheritance of constraints along the hierarchy of actions. See Fig. 4 for an example.

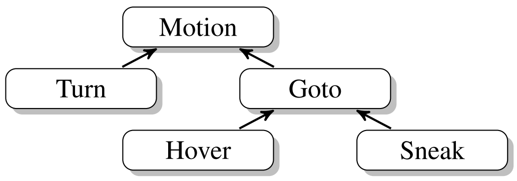

Fig. 4. Inheritance of constraints in a hierarchy of actions

E. Concurrency

The transformation, and hence the execution, may require several actions to be running concurrently. With our notion of durative actions this is possible, but the maintenance may even require two or more instantaneous actions to be executed simultaneously. READYLOG, however, currently only supports *interleaved concurrency* [4], which executes two action sequences concurrently by interleaving them. This is opposed to *true concurrency* [14], where sets of actions may be executed 'truly concurrently' between any two situations. To deal with this, we make use of something we call the ε-slot. Since, with a real execution system, concurrent calls to execute something will be serialized eventually, we consider two (or more) actions happening simultaneously if they happen within a time span not exceeding the ε-slot. ε is a natural number that denotes, how many subsequent instantaneous actions we think can be considered to happen simultaneously.

IV. A DETAILED EXAMPLE

To illustrate our approach we will now step through a detailed example. Consider the following setup: Given a Basic Action Theory that contains at least

- the fluent *Calibrated* indicating whether the robot's laser range finder has already been calibrated,
- the complex task domain action *goto* (x), which causes our robot to move to position x, and
- the two complex maintenance domain actions *calibrate* $=$ (*start_calibrate*, *stop_calibrate*, *Calibrating*), and *viscan* $=$ (*start_viscan*, *stop_viscan*, *Viscaning*). *calibrate* calibrates the laser range finder, i.e., it causes *Calibrated* to hold afterwards, and *viscan* tests for imminent collisions using a visual obstacle detector.

Each of these actions is always possible, and *calibrate* is the only way to set the fluent *Calibrated* $=$ *True*. The set of constraints is

$$C = \{\mathit{goto}\,(x)\ \textsc{Equals}\ \mathit{viscan},\ \mathit{goto}\,(x)\ \textsc{After}\ \mathit{calibrate}\},$$

which denotes that *goto* (x) is performed concurrently with *viscan* and only after the laser range finder was calibrated. These constraints imply – amongst others – three transformation rules. We abbreviate complex actions by their first letter with the usual modifiers, omitting the parameter of *goto* (x) as it is of no relevance for this example. Thus, G denotes the complex action *goto* (x), G^+ denotes *start_goto* (x), G^- denotes *stop_goto* (x) and G^F denotes the fluent *Going* (x). Furthermore, variables t_α refer to the position of an action α in a sequence of actions. The transformation rules are

(1) $(G\ \textsc{After}\ C, G^+, \neg\mathrm{Occurred}\,(C^-) \wedge \neg C^F,$
$\{t_{C^+} < t_{C^-}, t_{C^-} < t_{G^+}\}, \emptyset, \emptyset)$

where Occurred indicates whether C^- was already performed, i.e., $\exists s', a_1, \dots, a_N.\ s = do([a_1, \dots, a_N, C^-], s')$.

(2) $(G\ \textsc{Equals}\ V, G^+, \neg V^F,$
$\{t_{V^+} \approx_\varepsilon t_{G^+}\}, \emptyset, \{\mathit{between}\,(V^+, G^-, V^F)\})$

(3) $(G\ \textsc{Equals}\ V, G^-, V^F,\ \{t_{G^-} \approx_\varepsilon t_{V^-}\}, \emptyset, \emptyset)$

The agent has not performed any actions yet, i.e., the current situation is the initial situation S_0. The ε-slot length is 3 and the program to be performed is $\delta = [\mathit{start\_goto}\,(x), \delta_1, \mathit{stop\_goto}\,(x), \delta_2]$, where δ_1 and δ_2 are sequences of primitive actions that we do not want to detail here. Furthermore, both occurrence commitments $\mathcal{O}$ and persistency commitments $\mathcal{P}$ are empty in S_0. We now show how execution of this program evolves in terms of single steps of the transition semantics:

(I) We start in situation $s = S_0$, with the program

$$\delta = [\mathit{start\_goto}\,(x), \delta_1, \mathit{stop\_goto}\,(x), \delta_2]$$

and commitment sets $\mathcal{O} = \emptyset$ and $\mathcal{P} = \emptyset$.
(i) Inspecting the set of constraint-implied transformation rules reveals that the next action to be performed, *start_goto* (x), requires self-maintenance attention, as both "$\neg\mathrm{Occurred}\,(C^-) \wedge \neg C^F$" (from G AFTER C) and "$\neg V^F$" (from G EQUALS V) hold in s.
(ii) Therefore, the rules (1) and (2), together with *start_goto* are transformed into a constraint satisfaction problem over variables $t_{C^+}, t_{C^-}, t_{V^+}, t_{G^-}$. The domains are $D_{t_{C^+}} = D_{t_{C^-}} = D_{t_{V^+}} = D_{t_{G^-}} = \{1, 2, 3, 4\}$, because we know from the transformation rules, that we need to perform four actions to transform the current situation into a situation that satisfies the self-maintenance requirements. The transformation rules further imply restrictions on the order of three pairs of actions, namely

- $C_{C^+,C^-} = \{(i,j) \mid i < j\}$
- $C_{C^-,G^+} = \{(i,j) \mid i < j\}$
- $C_{V^+,G^+} = \{(i,j) \mid i \approx_\varepsilon j\}$.

Furthermore, there is an additional set of global constraints for each pair of variables that states that their values are different. We illustrate the CSP in Fig. 5.
(iii) This CSP is solved using a simple backtracking CSP solver that yields (possibly amongst others) the solution $t_{C^+} = 1, t_{C^-} = 3, t_{V^+} = 2, t_{G^-} = 4$, which corresponds to the sequence of actions

$$\delta^* = [\mathit{start\_calibrate}, \mathit{start\_viscan}, \mathit{stop\_calibrate}, R\,(\mathit{start\_goto}\,(x))],$$

where $R\,(\mathit{start\_goto}\,(x))$ is an indicator denoting that we already tried to perform self-maintenance for the

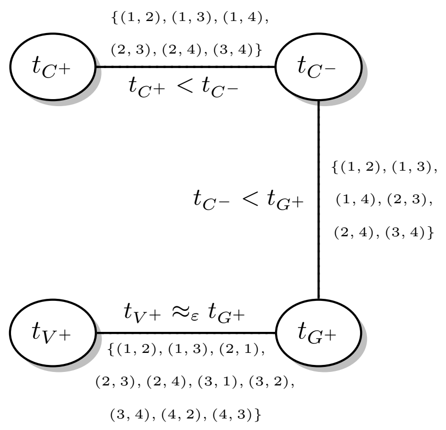

Fig. 5. An exemplary CSP for four variables t_{C^+}, t_{C^-}, t_{G^+}, and t_{V^+} represented as nodes. Edges are labeled (left and below, respectively) with constraints between these variables and the set of possible variable assignments (above and right, respectively).

action *start_goto* (x). This leads to the remaining program

$$\delta' = [\textit{start\_calibrate}, \textit{start\_viscan}, \textit{stop\_calibrate}, R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_gotox}, \delta_2]$$

(iv) Rule (2) also yields the persistency commitment "*between* (V^+, G^-, V^F)", which states that we require the fluent V^F to hold from V^+ to G^-, thus we update $\mathcal{P}$ accordingly to $\mathcal{P}' = \{\textbf{\textit{between}}\,(\textit{start\_viscan}, \textit{stop\_goto}\,(x)\,, \textit{Viscaning})\}$

(II) The second step of the transition semantics now faces the same situation $s = S_0$ as before, a modified program

$$\delta = [\textit{start\_calibrate}, \textit{start\_viscan}, \textit{stop\_calibrate}, R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2]$$

modified persistency commitments $\mathcal{P} = \{\textbf{\textit{between}}\,(\textit{start\_viscan}, \textit{stop\_goto}\,(x)\,, \textit{Viscaning})\}$ and unmodified occurrence commitments $\mathcal{O} = \emptyset$.
(i) *start_calibrate* is performed.
(ii) The remaining program is

$$\delta' = [\textit{start\_viscan}, \textit{stop\_calibrate}, R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2]$$

(iii) Neither $\mathcal{P}$ nor $\mathcal{O}$ require updates.

(III) The third step faces the situation

$$s = \textit{do}\,(\textit{start\_calibrate}, S_0)\,,$$

the program

$$\delta = [\textit{start\_viscan}, \textit{stop\_calibrate}, R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2],$$

and unmodified occurrence commitments $\mathcal{O} = \emptyset$ and persistency commitments $\mathcal{P} = \{\textbf{\textit{between}}\,(\textit{start\_viscan}, \textit{stop\_goto}\,(x)\,, \textit{Viscaning})\}$.
(i) *start_viscan* is performed.
(ii) The remaining program is

$$\delta' = [\textit{stop\_calibrate}, R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2].$$

(iii) As *start_viscan* is performed, $\mathcal{P}$ updates to $\mathcal{P}' = \{\textbf{\textit{until}}\,(\textit{stop\_goto}\,(x)\,, \textit{Viscaning})\}$, which denotes that the fluent *Viscaning* is required to hold from now on until *stop_goto* (x) is performed.

(IV) The fourth step begins with situation

$$s = \textit{do}([\textit{start\_calibrate}, \textit{start\_viscan}], S_0)$$

and program

$$\delta = [\textit{stop\_calibrate}, R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2],$$

$\mathcal{P} = \{\textbf{\textit{until}}\,(\textit{stop\_goto}\,(x)\,, \textit{Viscaning})\}$, and $\mathcal{O} = \emptyset$.
(i) *stop_calibrate* is performed.
(ii) The remaining program is

$$\delta' = [R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2].$$

(iii) Neither $\mathcal{P}$ nor $\mathcal{O}$ require updates.

(V) The situation advanced to

$$s = \textit{do}([\textit{start\_calibrate}, \textit{start\_viscan}, \textit{stop\_calibrate}], S_0)$$

with program remainder

$$\delta = [R\,(\textit{start\_goto}\,(x))\,, \delta_1, \textit{stop\_goto}\,(x)\,, \delta_2]$$

and commitments $\mathcal{O} = \emptyset$ and $\mathcal{P} = \{\textbf{\textit{until}}\,(\textit{stop\_goto}\,(x)\,, \textit{Viscaning})\}$.
(i) Neither "$\neg\text{Occurred}\,(\textit{stop\_calibrate}) \wedge \neg\textit{Calibrating}$" (from *goto* (x) AFTER *calibrate*), nor "$\neg$*Viscaning*" (from *goto* (x) EQUALS *viscan*) hold in s, thus, recovering from the faulty situation where *start_goto* (x) was supposed to be executed, has succeeded and *start_goto* (x) may finally be performed.
(ii) Therefore, the recovery indicator is removed and *start_goto* (x) is performed. Leaving

$$\delta' = [\delta_1, \textit{stop\_goto}\,(x)\,, \delta_2].$$

(iii) $\mathcal{P}$ and $\mathcal{O}$ do not change.

(VI) The situation is now

$$s = \textit{do}([\textit{start\_calibrate}, \textit{start\_viscan}, \textit{stop\_calibrate}, \textit{start\_goto}\,(x)], S_0)$$

with program remainder

$$\delta' = [\delta_1, \textit{stop\_goto}\,(x)\,, \delta_2],$$

and $\mathcal{O} = \emptyset$.
(i) Now the sequence of actions δ_1 – which may require further self-maintenance recovery – is performed. Thus, $\mathcal{P}$ and $\mathcal{O}$ may have been updated to $\mathcal{P}' = \{\textbf{\textit{until}}\,(\textit{stop\_goto}\,(x)\,, \textit{Viscaning})\} \cup \mathcal{P}_{\delta_1}$ and $\mathcal{O}' = \mathcal{O}_{\delta_1}$, respectively.
(ii) $\delta' = [\textit{stop\_goto}\,(x)\,, \delta_2]$.

(VII) The situation has advanced to

$$s = \textit{do}([\textit{start\_calibrate}, \textit{start\_viscan}, \textit{stop\_calibrate}, \textit{start\_goto}\,(x)\,, \delta_1], S_0),$$

the program remainder is

$$\delta = [\mathit{stop\_goto}(x), \delta_2],$$

and commitments are $\mathcal{O} = \mathcal{O}_{\delta_1}$ and $\mathcal{P} = \{\mathit{until}(\mathit{stop\_goto}(x), \mathit{Viscaning})\} \cup \mathcal{P}_{\delta_1}$.
(i) $\mathit{stop\_goto}(x)$ is to be performed, but there is another transformation rule left unsatisfied. Rule (3) states to assure that *stop_viscan* is performed immediately after $\mathit{stop\_goto}(x)$.
(ii) This is achieved by solving a CSP about rule (3), which leads to a program remainder $\delta^* = [R(\mathit{stop\_goto}(x)), \mathit{stop\_viscan}]$ without further modifications to $\mathcal{P}$ and $\mathcal{O}$.
(iii) Then

$$\delta' = [R(\mathit{stop\_goto}(x)), \mathit{stop\_viscan}, \delta_2]$$

(VIII) Step eight faces situation

$$s = \mathit{do}([\mathit{start\_calibrate}, \mathit{start\_viscan}, \mathit{stop\_calibrate}, \mathit{start\_goto}(x), \delta_1], S_0)$$

with program remainder

$$\delta = [R(\mathit{stop\_goto}(x)), \mathit{stop\_viscan}, \delta_2]$$

and commitments $\mathcal{O} = \mathcal{O}_{\delta_1}$ and $\mathcal{P} = \{\mathit{until}(\mathit{stop\_goto}(x), \mathit{Viscaning})\} \cup \mathcal{P}_{\delta_1}$.
(i) Evaluating the recovery indicator for $\mathit{stop\_goto}(x)$ leads to re-evaluation of rule (3). We notice that *Viscaning* still holds. Since we also know that $\mathit{stop\_goto}(x)$ is about to be executed, we can retract $\mathit{until}(\mathit{stop\_goto}(x), \mathit{Viscaning})$ from $\mathcal{P}$.
(ii) Finally, $\mathit{stop\_goto}(x)$ is performed and $\mathcal{P}$ is updated accordingly to $\mathcal{P}' = \mathcal{P}_{\delta_1}$.

(IX) The current situation is

$$s = \mathit{do}([\mathit{start\_calibrate}, \mathit{start\_viscan}, \mathit{stop\_calibrate}, \mathit{start\_goto}(x), \delta_1, \mathit{stop\_goto}(x)], S_0),$$

the program remainder is

$$\delta = [\mathit{stop\_viscan}, \delta_2],$$

with $\mathcal{P} = \mathcal{P}_{\delta_1}$ and $\mathcal{O} = \emptyset$.
(i) *stop_viscan* is performed.
(ii) Neither $\mathcal{P}$ nor $\mathcal{O}$ require updates.
(iii) $\delta' = \delta_2$.

(X) The execution of δ_2 continues and may require further self-maintenance or not, but eventually the program finishes.

V. Discussion

In this paper we presented 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. Up to now, we have not considered any optimization issues in scheduling the maintenance actions, we just apply one possible scheduling returned as a solution to the CSP. It could be worthwhile looking into methods on how to determine which of a set of solutions is optimal.

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?"*.

Acknowledgments

The authors thank the reviewers for their comments.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[2] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. 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*, pages 355–362. AAAI Press / The MIT Press, 2000.

[3] J. Claßen, Y. Hu, and G. Lakemeyer. A Situation-Calculus Semantics for an Expressive Fragment of PDDL. In *AAAI'07: Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 956–961. AAAI Press, 2007.

[4] G. de Giacomo, Y. Lespérance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[5] A. Ferrein and G. Lakemeyer. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991, 2008.

[6] A. Finzi and F. Pirri. Flexible interval planning in concurrent temporal golog. In *Working notes of the 4th Int. Cognitive Robotics Workshop*, 2004.

[7] Y. Gu and M. Soutchanski. Reasoning about Large Taxonomies of Actions. In D. Fox and C. P. Gomes, editors, *AAAI'08: Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 2, pages 931–937. AAAI Press, 2008.

[8] S. Lemai and F. Ingrand. Interleaving temporal planning and execution in robotics domains. In *AAAI'04:Proceedings of the 19th National Conference on Artifical Intelligence*, pages 617–622. AAAI Press / The MIT Press, 2004.

[9] H. Levesque and G. Lakemeyer. *Cognitive Robotics*. Handbook of Knowledge Representation. Elsevier, 2007.

[10] H. Levesque and R. Reiter. High-level robotic control: Beyond planning. a position paper. In *AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap*, March 1998.

[11] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[12] J. McCarthy. Situations, Actions, and Causal Laws. Technical Report Memo 2, AI Lab, Stanford University, California, USA, 3 July 1963. Published in Semantic Information Processing, ed. Marvin Minsky. Cambridge, MA: The MIT Press, 1968.

[13] F. Pirri and R. Reiter. Some Contributions to the Metatheory of the Situation Calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[14] R. Reiter. 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)*, pages 2–13, Cambridge, Massachusetts, U.S.A., November 1996.