ECAI 2016
G.A. Kaminka et al. (Eds.)

doi:10.3233/978-1-61499-672-9-1265

Interruptible Task Execution with Resumption in Golog

Gesche Gierse and **Tim Niemueller** and **Jens Claßen** and **Gerhard Lakemeyer**[1]

Abstract. Mobile robots should perform a growing number of tasks and react to time-critical events. Thus, the ability to interrupt a task and resume it later is crucial. While interleaved execution occurs often in robotics, existing approaches do not consider the fact that interrupting a task and resuming an interrupted task often requires intermediate steps. In this paper we present an approach to interruptible task execution with resumption. We propose INTRGOLOG which extends INDIGOLOG by task interruption and resumption through introducing new constructs to determine and fulfill the requirements of tasks. Our experiments on a service robot and in simulation show that the ability to switch to another task enables a robot to react in a swift and reliable fashion to new events.

1 INTRODUCTION

Mobile robots are envisioned to fulfill a growing number of tasks in the future—in domestic as well as in industrial scenarios. A required capability then is *task switching*. The problem turns out to be more involved than is obvious. For example, while a robot is cleaning up the breakfast table putting tableware into the dishwasher in the kitchen, the ringing of the doorbell adds the higher prioritized task to answer the door. Completing the initial task first would take too long (the mailman might already be gone by the time the robot answers the door). Hence the robot must *interrupt its current task* as soon as possible. But the robot may not be able to perform the new task right away, e.g., it might be carrying an object that it needs to put down first in order to have its hand free to open the door. Therefore, it needs to reason about the necessary steps to resolve conflicts between requirements when switching tasks. After completing the high priority task, the robot is expected to *resume the interrupted task*. This might involve remembering parameters of the steps performed when switching: if the robot held an object when the doorbell rang and placed it down on the quickly accessible kitchen counter, it must later retrieve the object from there to complete cleaning up the table.

In this paper, we formalize task switching as an extension of INDIGOLOG accounting for a number of problems that arise. We introduce a new kind of interruption. In contrast to CONGOLOG interrupts which may yield control to other programs at arbitrary points, we define an extended transition semantics that explicitly handles the addition, removal, and switching of tasks, which may involve the introduction of intermediate steps for resolving requirement conflicts. We also provide constructs to avoid task switching in certain parts of a program, to force the re-execution of sub-programs marked indivisible, and a well-defined interruption of durative actions.

For this purpose, we define a new transition relation $TTrans$ for task operations (adding, removing, switching) on top of INDIGOLOG's original $Trans$ which defines how to execute the respective current task. To determine the intermediate steps necessary on a task switch, we introduce the new concept of *promises*. These are terms denoting asserted or required conditions (of the running or the interrupting task) used to determine conflicts during task switching. Promises are formulated with regard to actions, i.e., whether an action asserts or revokes a promise. We rely on *durative* actions, such that long-running actions can be interrupted. They are defined as a start and an (exogenous) end action. Finally, *re-execution sequences* allow to specify parts of a program that must be repeated in full on resumption, e.g., to repeat a sensing action before grasping if the latter was interrupted.

Figure 1. *Evaluation scenario.* The robot moves cups from the table to the dishwasher, when the doorbell rings. It should then answer the door.

Our main contribution is the formalization of task interruption and resumption as an extension of INDIGOLOG's transition semantics and the implementation of INTRGOLOG that can deal with cases where procedures for intermediate steps can be pre-programmed.

We have evaluated the approach both in simulation and on a real robot. The results show that the proposed system enables the robot to react to new events in a swift fashion and that it indeed reduces high priority task latency.

In Section 2, we briefly introduce the Situation Calculus and GOLOG. Then we explain our approach in detail (Section 3). The evaluation is described in Section 4. Some related work is discussed in Section 5 before concluding in Section 6.

2 THE SITUATION CALCULUS AND GOLOG

In this section, we will describe the situation calculus and GOLOG and some of its variants, upon which our approach builds.

2.1 Situation Calculus

The situation calculus [16] is a dialect of first-order logic that describes a changing world. It has the sorts *situations* and *actions*. A *situation* is defined by nested terms of the form $do(a, s)$ starting from

[1] Knowledge-Based Systems Group, RWTH Aachen University, Germany, email: {gierse,niemueller,classen,lakemeyer}@kbsg.rwth-aachen.de

the initial situation S_0, where $do(a, s)$ describes the situation that occurs after executing *action* a in situation s. *Actions* are encoded by functions and may have preconditions, e.g., the action `pick_up(x)` is only possible if the robot has an empty hand. The status of the world is described by *fluents*, e.g., `holding(x,s)` denotes that the agent is holding object `x` in situation `s`. Effects of actions are formalized by *successor state axioms*. These concepts can be formalized in a basic action theory $\mathcal{D}$ as described in [19].

2.2 GOLOG

GOLOG [14] is a procedural programming language based on the situation calculus. The situation calculus is used to find a legal sequence of actions to satisfy a GOLOG program. The programming language features different programming constructs, such as sequence, while loops, and conditions. Additionally, it exhibits non-deterministic constructs, for example $\pi x.\delta$, where x is chosen non-deterministically such that δ can be legally executed. As an example, consider a GOLOG program to clear a table:

while $\exists object.$`on_table`$(object)$ **do**
  $\pi x.$`on_table`$(x)?$; `pick_up`(x); `put_on_floor`(x)
endWhile

As long as at least one object is on the table, the program non-deterministically select one, picks it up and puts it down on the floor.

In GOLOG, programs can only be executed offline. That means, given a GOLOG program, a sequence of actions is computed that fulfills the program. Then the whole sequence of actions is executed. In real-world applications this is unrealistic: a robot has to be able to sense its surroundings and make decisions based on this. Also, actions may fail and thus a pre-computed sequence cannot be executed. Therefore, extensions have been formalized to deal with these problems, two of which we are going to describe in more detail. In the following GOLOG will denote the family of programming languages.

2.2.1 CONGOLOG *and* INDIGOLOG

CONGOLOG [2] is an extension that adds concurrent execution (multiple programs are executed step-wise interleaved), prioritized interrupts (with a higher priority interrupt handling program urgent actions can be executed), and exogenous actions (events in the environment). However, execution still happens offline.

INDIGOLOG [3] extends CONGOLOG by adding sensing and online execution.[2] It interprets programs in an incremental way, such that modifications to fluents can alter the trace of situations to program completion. Such traces may then depend on data sensed during execution. These sensing results are stored in a history.

Our work is an extension of INDIGOLOG. It relies on incremental execution in order to be able to interrupt and resume tasks and to interleave the execution of multiple tasks.

2.2.2 GOLOG *and* INDIGOLOG *Programs*

A GOLOG program δ can execute an action (a) or verify a condition ($\phi?$). These basic operations can be combined with the following control structures: Sequence ($\delta_1; \delta_2$), non-deterministic branch ($\delta_1|\delta_2$), non-deterministic choice of argument ($\pi x.\delta$), non-deterministic iteration (δ^*), if-conditions, while-loops, and procedure calls. To search for a plan ahead of execution, the search operator $\Sigma(\delta)$ is used. Concurrently executing two programs is expressed by $\delta_1||\delta_2$. Additionally, there are prioritized concurrency ($\delta_1 \rangle\rangle \delta_2$), and concurrent iteration ($\delta^{||}$). To interrupt a program whenever condition ϕ holds, one writes $\langle\phi \rightarrow \delta\rangle$.

2.2.3 Transition Semantics

Both CONGOLOG and INDIGOLOG use a transition semantics which defines how a program δ in some situation s can evolve to a program δ' and a situation s'. As an example consider the programming construct a which executes an action:

$$Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a, s)$$

Here an action a transforms to the program nil which denotes the empty program. The transition may only be used if it is possible to execute action a in the current situation s. The resulting situation s' is the situation that occurs when action a is executed in situation s.

A program δ may terminate if it is final, i.e., $Final(\delta, s)$ holds. For example, the program nil is final, while the program a is not final – an action still needs to be executed. The predicate is defined recursively, e.g., a sequence of actions $\delta_1; \delta_2$ may terminate if and only if both δ_1 and δ_2 are final.

A *history* is used to represent a sequence of actions with their sensing results. A history σ is of the form $\sigma = (a_1, \mu_1) \cdot \ldots \cdot (a_n, \mu_n)$ if actions $a_1, \ldots, a_n$ were executed and each action a_i returned the sensing result μ_i. It is convenient to use $end[\sigma]$ as an abbreviation for the end situation of history, defined by: $end[\epsilon] = S_0$; and inductively, $end[\sigma \cdot (a, x)] = do(a, end[\sigma])$ [3]. Additionally, $Sensed[\sigma]$ denotes the formula that contains all sensing results of history σ [3].

2.2.4 Online Execution

While GOLOG tries to find a sequence of actions beforehand and then tries to execute all actions, INDIGOLOG can execute one action and afterwards decide how to execute the remaining program. This is especially useful since we can delay decisions to the execution time and use information gained by sensing actions. The transition semantics allow to search for the next step of the program. Assume we started a program and at some later point the program δ is remaining to be executed. Additionally, we have executed some actions and obtained corresponding sensing results. In the online execution the interpreter will now either stop, if the remaining program δ is final, or find one transition from δ and situation s to δ' and some situation s'. If s and s' are not equal, s' will be of the form $do(a, s)$. In this case the action a is executed in the real world. The next step is determined by the basic action theory, the transition and final rules, and the sensing results. We also monitor exogenous actions, i.e., actions the agent observes but does not execute itself. These are added to the history of executed actions.

2.2.5 Concurrency and Interrupts

In INDIGOLOG, programs which are run concurrently are executed in an interleaved fashion. An interrupt $\langle\phi \rightarrow \delta\rangle$ executes its program δ if the condition ϕ holds and no process with higher priority is executed. It will suspend processes of lower priority. After execution of the interrupt, suspended processes are continued. A major

[2] Since INDIGOLOG includes CONGOLOG we focus on the former.

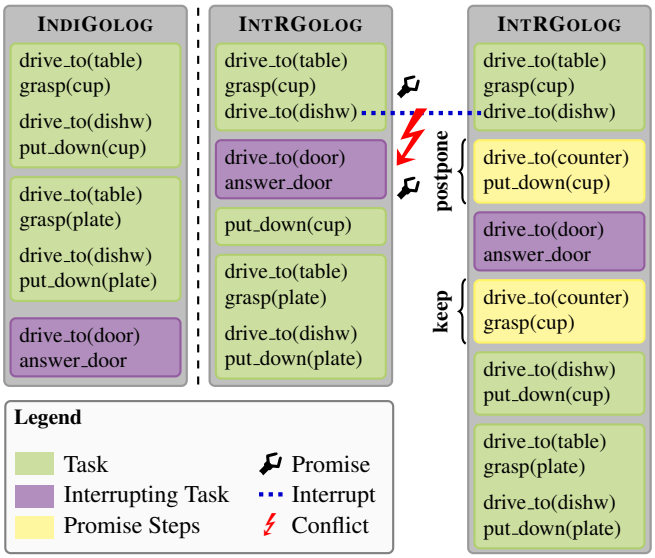

Figure 2. *Task interruption.* The left shows how the interrupting task is simply appended in INDIGOLOG, while INTRGOLOG inserts it as soon as the interrupt event occurs. The conflict caused by the `hand_used` promise is resolved by introducing suitable steps to postpone and keep the promise.

problem with interrupts is that they yield control to interrupts at arbitrary points. This is a problem for interruptible task execution with resumption, since additional actions might be necessary before the switch can be performed, and again when resuming the original task.

As an example why GOLOG interrupts do not suffice for task interruption consider the following program:

$$\langle \texttt{doorbell\_rings} \rightarrow \texttt{answer\_door} \rangle \rangle \; \texttt{clean\_up}$$

If the doorbell rings, the robot will interrupt and answer the door, otherwise it will clean up. This is a simple approach to interleaved task execution and it does not work as expected in the envisioned situations: Consider that the doorbell rings while the robot is about to put an object in the dishwasher. The switch will be immediate, thus the robot drives to the door, but it cannot open it, because it hand is still full. Therefore, the higher prioritized program will yield control and the clean up may continue. However, the robot is not in front of the dishwasher anymore, thus the precondition of the put-in-dishwasher action does not hold anymore. So, without performing intermediate steps, the combined interruptible program may no longer be executable. To avoid this, one would have to anticipate all problems that may potentially arise, along with possible solutions, and include those in the interrupt. This is not only tedious, but also error-prone, particularly when the interplay between multiple interleaved tasks has to be considered. Instead, we introduce a new notion of interrupts where conflicts are avoided by executing intermediate steps on interruption.

2.2.6 *Durative actions*

To represent the concurrency of real-world actions *durative actions* [19] can be used. It essentially means splitting up an action in an instantaneous start and an (exogenous) instantaneous end action that occurs when the action is finished. As long as the action is being executed, a fluent for this action is true. In between, a program may continue or interrupts may be handled. Another possible representation has been used in CC-GOLOG [7], where an action $waitFor(condition)$ is used to have the agent wait until some formula $condition$ (which depends on continuously changing fluents such as the robot position) becomes true, and then continue executing further actions (e.g., stopping the movement). Here however, we want to allow the interruption of durative actions through exogenous events. Therefore, we have:

$$\texttt{exec\_dur}(d) \stackrel{\text{def}}{=} \texttt{dur\_act\_start}(d); \texttt{wait\_for}(d)$$

$$\texttt{wait\_for}(d) \stackrel{\text{def}}{=} \textbf{while } \texttt{dur\_flu}(d) \textbf{ do } \texttt{wait} \textbf{ endWhile}$$

This is the typical case where nothing except waiting (and thereby checking for interrupts) happens while one durative action is running. Our formalization also allows to state explicitly in which order to start durative actions and wait for their termination, and thus interleave such processes.

3 EXTENDED TRANSITION SEMANTICS

The robot should to be able to execute a set of tasks instead of just one program. Therefore, we define tasks which consist of a program, a priority, and a unique ID. We extend the transition semantics of INDIGOLOG to work on a set of tasks. Instead of directly using the transition predicate $Trans$, we introduce $TTrans$ whose semantics is added on top of the original $Trans$ predicate. With it a task can be added or removed. It facilitates task switching and uses the original $Trans$ to execute the current task.

As described in Section 2.2.5 CONGOLOG interrupts are not sufficient to handle task switches, since intermediate steps may be necessary to interrupt and later resume a task. To ensure that these necessary intermediate steps are performed when switching tasks we introduce two additional concepts.

Re-execution sequences ensure that actions that need to be executed together are fully repeated on resumption. For example, the robot uses a sensing action to detect an object on a table and is about to grasp it. When interrupted in this situation it needs to sense if the object is still there on resumption. Essentially, an interrupted task is modified by restoring an already executed part of the program.

Promises are used to specify necessary conditions for (parts of) programs and to determine conflicts during task switches. The basic action theory defines which action asserts or retracts a promise during execution. When switching to a new task, the system determines if the promises are *conflicting*, that is whether any currently asserted promise is required by the new task. In this case, domain-specific procedures are used to *postpone* and later *keep* the promise. The postpone program is executed before starting a different task and the keep procedure is executed before resuming the interrupted task. Task-specific information may be stored during postponing to be used when a promise is to be kept again.

As an example, consider Figure 2. The left column shows a classic INDIGOLOG implementation that requires the two objects to be cleaned up before answering the door, as the cleanup program was already running. The two right columns show INTRGOLOG's behavior. Once the interrupt event occurs (blue dotted line), the respective action is interrupted and the new higher priority task inserted. But the robot has grasped an object with its only hand and needs this very hand free in order to successfully execute the new, higher prioritized task of answering the door. A promise *hand_used* is defined by the user. Finishing grasping an object or starting to open a door would assert the promise, while finishing putting down an object or finishing opening the door retracts it. When switching after grasping

to opening the door, the robot would postpone the promise *hand_used* by driving to a nearby counter and putting down what it is carrying. However, to later resume the task, it then needs to have the object in its hand again. The *keep* procedure will handle this by remembering where it put the object and then retrieve it (yellow boxes). Afterwards the previous task can be resumed.

3.1 Task Definition

A task is defined as a tuple $\tau = \langle \delta, \texttt{prio}, \texttt{id} \rangle$, where δ is a program as before (that will be modified by $Trans$), `prio` is the task's priority and `id` is a unique ID. As a shorthand notation we will use $prog(\tau)$, $prio(\tau)$ and $id(\tau)$ to access the program, the priority and the ID of the task τ, respectively.

We write finite sets $\tau_1, \ldots, \tau_n$ of tasks as $\Omega = \{\tau_1, \ldots, \tau_n\}$. We will write $\Omega[\tau_i = \tau_i']$ to denote the task set that is like Ω except with τ_i replaced by τ_i'. Furthermore, $\Omega[prog(\tau_i) = \delta']$ will denote Ω where the program of the task τ_i is changed to δ'. Also, $\Omega[\{\tau_1 = \tau_1', \ldots, \tau_k = \tau_k'\}]$ will denote that all tasks τ_i are changed to τ_i' for $1 \leq i \leq k$ and the other tasks remain identical to Ω.

3.2 Transition Semantics Extension Idea

We define an extension of $Trans$ called $TTrans$. It will handle tasks, while referring to the standard INDIGOLOG transition semantics to execute a single task. Intuitively, $Trans$ defines transitions between configurations consisting of a program δ and a situation s: $(\delta, s) \vdash (\delta', s')$. In contrast $TTrans$ yields transitions of the form $(i, \Omega, s) \vdash (i', \Omega', s')$, where i, i' are the indexes denoting the current task, Ω, Ω' are task sets and s, s' are situations. A modification of Ω to Ω' can be either *adding* a new task, *removing* a task, *executing* the current task or *switching* to the task with highest priority.

Generally, our new top-level transition predicate $TTrans$ uses fluents that are set by exogenous actions to decide which of the above possibilities are used in each step of an online execution.

In the following, we will present a number of sufficient conditions for doing a transition step $TTrans(i, \Omega, s, i', \Omega', s')$. The actual axiom defining $TTrans$ is then understood as the completion of those conditions, i.e., $TTrans(i, \Omega, s, i', \Omega', s')$ is equivalent to the disjunction over all right-hand sides of the rules given below. All situation calculus terms (e.g., fluents, actions, procedures) are set in `monospace`, while internal formulas are set in *italics*.

3.3 Adding a Task

A task is added by an exogenous action `add_task`(τ). We use the fluent `taskupdate` which will denote whether there is a new task. Additionally, if there is a new task, the fluent `new_task` will contain the new task. If this fluent is set, we will update the set of tasks Ω.

$$\begin{aligned} &TTrans(i, \Omega, s, i', \Omega', s') \subset \\ &\quad \texttt{taskupdate}[s] \wedge \exists \tau_{new}.\, \texttt{new\_task}[s] = \tau_{new} \\ &\quad \wedge \Omega' = \Omega \cup \{\tau_{new}\} \wedge s = s' \wedge i = i' \end{aligned} \tag{1}$$

3.4 Removing a Task

A task can also be removed from Ω. If a task is final, it can be removed. Similar to $Trans$ we also add an extended version of $Final$ called $TFinal$ to define if tasks are final. A set of tasks is final if it is empty and a single task is final if its program is final regarding the original CONGOLOG's $Final$-rules:

$$\begin{aligned} TFinal(\Omega, s) &\equiv \Omega = \emptyset \\ TFinal(\tau, s) &\equiv Final(prog(\tau)) \end{aligned}$$

Besides removing final tasks, a task can also be deleted using an exogenous action `remove_task`(id). We use the fluent `taskremove` to denote that a task deletion is requested. The fluent `delete_task` will contain the ID of the task that should be removed. The index i denoting the current task is set to zero to express that currently no task is selected to execute (the selection of the new task will be handled by the task switch case described below).

$$\begin{aligned} &TTrans(i, \Omega, s, i', \Omega', s') \subset \\ &\quad s = s' \wedge \exists \tau \in \Omega\, \exists id.(Final(\tau, s) \vee \texttt{taskremove}[s] \\ &\quad \wedge \texttt{delete\_task}[s] = id) \wedge id(\tau) = id \\ &\quad \wedge (id = i \supset i' = 0) \wedge (id \neq i \supset i' = i) \\ &\quad \wedge \Omega' = \Omega \setminus \{\tau\} \end{aligned} \tag{2}$$

3.5 Executing a Task

We can execute the current task if the following conditions hold: A proper current task is defined by the id i (zero indicates that this is not the case, either because the current task was finished or removed by the user). There is neither a new task to be added nor a task to be removed, and if task switching is allowed, the current task has highest priority. A step of the program of the current task is the executed according to the original $Trans$:

$$\begin{aligned} &TTrans(i, \Omega, s, i', \Omega', s') \subset \\ &\quad \exists \tau_i \in \Omega.id(\tau_i) = i \wedge i \neq 0 \wedge i' = i \wedge \neg \texttt{taskupdate}[s] \\ &\quad \wedge \neg \texttt{taskremove}[s] \wedge (\neg \texttt{switching\_allowed}[s] \\ &\qquad \vee \forall \tau_k \in \Omega.prio(\tau_k) \leq prio(\tau_i)) \\ &\quad \wedge \exists \gamma.Trans(prog(\tau_i), s, \gamma, s') \wedge \Omega' = \Omega[prog(\tau_i) = \gamma] \end{aligned} \tag{3}$$

3.6 Switching Tasks

Switching a task contains some technicalities, so we will discuss the corresponding rule in an incremental manner. We will first present a simple version that does not use any intermediate steps and then describe the formalities needed for re-execution sequences and promises and how they are integrated in $TTrans$. For now consider switching to the task with highest priority:

$$\begin{aligned} &TTrans(i, \Omega, s, i', \Omega', s) \subset \\ &\quad \exists \tau_j \in \Omega. \forall \tau_k \in \Omega.prio(\tau_k) \leq prio(\tau_j) \\ &\quad \wedge i' = id(\tau_j) \wedge \exists \tau_i \in \Omega.\, id(\tau_i) = i \\ &\quad \wedge prio(\tau_j) > prio(\tau_i) \wedge id(\tau_i) \neq id(\tau_j) \\ &\quad \wedge \Omega' = \Omega \end{aligned} \tag{4}$$

Next, we include the fluent `switching_allowed` that enables and disables task switching. The fluent is toggled by the actions `enable_switching` and `disable_switching`. In the definition of removing tasks we introduced the case where currently no task is selected, indicated by $i = 0$. This is a special case, because no task can be executed by the $TTrans$-rule for execution. So even

if switching is disabled, we need to select a new current task if $i = 0$. The resulting rule looks as follows, where the new part is underlined:

$$\begin{aligned} &TTrans(i, \Omega, s, i', \Omega', s) \subset \\ &\quad \underline{(i = 0 \vee \texttt{switching\_allowed}[s])} \\ &\quad \wedge \exists \tau_j \in \Omega. \forall \tau_k \in \Omega. prio(\tau_k) \leq prio(\tau_j) \\ &\quad \wedge i' = id(\tau_j) \wedge (\exists \tau_i \in \Omega. id(\tau_i) = i \wedge prio(\tau_j) > prio(\tau_i) \\ &\quad \wedge id(\tau_i) \neq id(\tau_j) \underline{\vee i = 0}) \wedge \Omega' = \Omega \end{aligned} \tag{5}$$

3.6.1 Re-execution Sequences

A sequence of actions can be marked as a re-execution sequence. It is then executed as usual, however, in the case of a task switch within that sequence, the whole re-execution sequence is executed from scratch at the resumption of the task.

The programmer uses the construct **reexec**(δ) in their procedures. $Trans$ then transforms this to **re**(δ, δ). Transitions from **re**(δ', δ) will only modify δ', while δ remains the original δ. We extend (CONGOLOG's original) $Trans$ and $Final$ by the following:

$$\begin{aligned} &Trans(\textbf{reexec}(\delta), s, \textbf{re}(\delta, \delta), s) \equiv True \\ &Trans(\textbf{re}(\delta, \delta_{re}), s, \delta', s') \equiv \\ &\quad \exists \gamma. Trans(\delta, s, \gamma, s') \wedge \delta' = \textbf{re}(\gamma, \delta_{re}) \\ &\quad \vee Final(\textbf{re}(\delta, \delta_{re})) \wedge \delta' = nil \\ &Final(\textbf{reexec}(\delta), s) \equiv False \\ &Final(\textbf{re}(\delta, \delta_{re}), s) \equiv Final(\delta, s) \end{aligned}$$

As an example consider the following program:

reexec(exec_dur(detect_cup);
if cups_on_table **then**
exec_dur(grasp) **else** noop **endIf**)

where the robot grasps a cup, if it detected one. If the robot is interrupted when starting to grasp the object, the robot would continue with the whole program above on resumption. Let the above program be denoted by **reexec**(δ_{SG}). Internally, this is transformed to **re**(δ_{SG}, δ_{SG}). When executing it in a situation where cups are sensed, we would eventually reach **re**(exec_dur(grasp), δ_{SG}). If interrupted at that time, we want to execute **re**(δ_{SG}, δ_{SG}) again on resumption. Thus, we now extend Formula 5 to replace all occurrences of **re**(δ, δ_{re}) in the current task by **re**(δ_{re}, δ_{re}) before switching. This is done by the formula $reset\_re(\tau_i, \tau_i')$.

$$\begin{aligned} &TTrans(i, \Omega, s, i', \Omega', s) \subset \\ &\quad (i = 0 \vee \texttt{switching\_allowed}[s]) \\ &\quad \wedge \exists \tau_j \in \Omega. \forall \tau_k \in \Omega. prio(\tau_k) \leq prio(\tau_j) \\ &\quad \wedge i' = id(\tau_j) \wedge (\exists \tau_i \in \Omega. id(\tau_i) = i \wedge prio(\tau_j) > prio(\tau_i) \\ &\quad \wedge id(\tau_i) \neq id(\tau_j) \vee i = 0) \\ &\quad \underline{\wedge \exists \tau_i'. reset\_re(\tau_i, \tau_i')} \\ &\quad \underline{\wedge \Omega' = \Omega[prog(\tau_i) = \tau_i']} \end{aligned} \tag{6}$$

To handle cases where intermediate steps are also necessary before starting the new task we introduce the concept of *promises*.

3.6.2 Promises

Promises are terms that denote asserted or required conditions of programs. They are defined as part of the basic action theory. We introduce new situation independent predicates to describe promises and which actions assert and retract a promise. Procedures to *postpone* and *keep* a promise need to be defined. These are the intermediate steps executed before starting the interrupting task and before resuming the interrupted task. We use promise($prom, order$) to denote that the term $prom$ is a promise and is associated with a numeric value (natural number) $order$ to indicate in which order to postpone and keep promises in case of an interruption. To denote that an action a starts a promise $prom$, we write asserts_promise(a, $prom$). For example, a successful exogenous end_grasp starts the promise has_obj. A promise stops being asserted when an action retracts it, written retracts_promise(a, $prom$). For example, a successful exogenous end_grasp starts the promise has_obj and the action end_put_down ends the promise has_obj.

Internally, the fluent asserted_promise($prom, s$) indicates that the promise $prom$ is asserted in situation s. The successor state axiom of this fluent is described using the predicates from above:

$$\begin{aligned} &\texttt{asserted\_promise}(p, do(a, s)) \equiv \\ &\quad \texttt{asserts\_promise}(a, p) \vee \\ &\quad \texttt{asserted\_promise}(p, s) \wedge \neg \texttt{retracts\_promise}(a, p) \end{aligned}$$

Now we need to define procedures to perform intermediate steps when switching a task. They are called *postpone* and *keep* procedures. In the case of the promise hand_used the robot would need to put away what it is holding with the *postpone* procedure. To later resume the interrupted task, the robot needs to hold the object again before continuing the task. Thus, in the *keep* procedure it has to find the object again. Our extension provides two actions and a fluent to help with remembering necessary parameters for keeping a promise. We introduce a fluent has_param($i, prom, key, s$)=$value$ that indicates that the task with ID i has the parameter $value$ for the postponed promise $prom$ and the key key in situation s. When postponing a promise, we can use the action set_param($id, prom, key, value$) to set this fluent. In the case of hand_used we would for example save the table where we put the object. In the *keep* procedure we can then read this value from the fluent. After successfully using the information, the action used_param($id, prom, key$) resets the fluent has_param($id, prom, key$) to $false$.

As an example consider the procedures to keep and postpone the promise hand_used. As domain independent fluents, we first define which procedure keeps and postpones the promise:

postpone_promise(hand_used, put_somewhere(Id)).
keep_promise(hand_used, get_back_obj(Id)).

To *postpone* we define a procedure which will ensure that it is actually holding something, in that case it will sense the nearest table, go there, put the object down and remember the place.

proc put_somewhere(Id)
sense_holding_obj;
if holding_obj **then**
sense_nearest_table;
?(nearest_table = $Table$);
exec_dur(drive_to($Table$));
exec_dur(put_down);
set_param($hand\_used, Id, table, Table$)
endIf endProc

Before resuming the task, we will check if a location for an object to grasp was saved (the promise `hand_used` could have been in effect because the robot was opening a door when interrupted, in that case we do not want to pick up any object. Re-execution sequences will take care of restarting durative actions that were interrupted during execution). In that case the robot will go to the table, detect the object and pick it up. Afterwards it will indicate that it used the information stored by the postpone procedure, which will reset the fluent for that promise, ID and key to *false*.

proc `get_back_obj`(Id)
  if $\neg$`has_param`$(\texttt{hand\_used}, Id, table) = false$ **then**
    ?(`has_param`$(\texttt{hand\_used}, Id, table) = Table$);
    `exec_dur`(`drive_to`$(Table)$);
    `exec_dur`(`detect_obj`);
    if `obj_on_table` **then** `exec_dur`(`grasp`) **endIf**;
    `used_param`$(\texttt{hand\_used}, Id, table)$
  endIf endProc

So, in case of a task interruption, we need to check if there is a promise that is currently asserted but may be needed in the interrupting task (like a hand holding an object and the interrupting task needs the hand to open the door). To define this, we will look ahead what promises could be asserted by the interrupting task by extending the definition of `asserts_promise` to programs. For example:

$$\texttt{asserts\_promise}(\delta_1;\delta_2,p) \stackrel{\text{def}}{=} \texttt{asserts\_promise}(\delta_1,p) \vee \texttt{asserts\_promise}(\delta_2,p)$$

$$\texttt{asserts\_promise}(\pi v.\delta,p) \stackrel{\text{def}}{=} \exists v.\, \texttt{asserts\_promise}(\delta,p)$$

The rules for the remaining constructs are defined in a similar manner. With this lookahead we can describe if a promise is *contradicting*, i.e., it is asserted and the interrupting task τ could assert it, too.

$$\begin{aligned} &\mathit{is\_contradicting\_promise}(\mathit{prom},\mathit{order},\tau,s) \stackrel{\text{def}}{=} \\ &\quad \texttt{promise}(\mathit{prom},\mathit{order}) \wedge \texttt{asserted\_promise}(\mathit{prom},s) \\ &\quad \wedge \texttt{asserts\_promise}(\tau,\mathit{prom}) \end{aligned}$$

For all contradicting promises we want to prepend the program of the interrupting task with the corresponding *postpone* procedure and the interrupted task with the respective *keep* procedure. We define two helper functions to prepend the current task with these procedures given the current situation s, the set of tasks Ω and the ID j of the interrupting task:

$$\begin{aligned} &\mathit{postpone\_contradicting\_promises}(s,\Omega,j) = \Omega' \stackrel{\text{def}}{=} \\ &\quad \exists \tau \in \Omega.\ id(\tau) = j \\ &\quad \wedge \Omega' = \Omega[\mathit{prog}(\tau) = \texttt{disallow\_switching}; \delta_p; \\ &\qquad\qquad\qquad \texttt{allow\_switching}; \mathit{prog}(\tau)] \end{aligned}$$

where $\delta_p = \delta_1;\ldots;\delta_n$ and $\delta_1,\ldots,\delta_n$ are exactly those δ_i such that

$$\begin{aligned} &\exists \mathit{prom}_i, \mathit{order}_i \\ &\quad \mathit{is\_contradicting\_promise}(\mathit{prom}_i,\mathit{order}_i,\tau,s) \wedge \\ &\quad \texttt{postpone\_promise}(\mathit{prom},\delta_i) \end{aligned}$$

and $\mathit{order}_i > \mathit{order}_j$ for all $i > j$.

In a similar manner, we define a function to prepend each task that asserted a conflicting promise with the associated *keep* procedure:

$$\begin{aligned} &\mathit{keep\_postponed\_promises}(\Omega,s) = \Omega' \stackrel{\text{def}}{=} \\ &\quad \Omega' = \Omega[\{\mathit{prog}(\tau) = \delta_k^{\tau}; \mathit{prog}(\tau) \mid \tau \in \Omega\}] \end{aligned}$$

where $\delta_k^{\tau} = \delta_n;\ldots;\delta_1$ and $\delta_1,\ldots,\delta_n$ are exactly those δ_i such that

$$\begin{aligned} &\exists \mathit{prom}_i, \mathit{order}_i \\ &\quad \mathit{is\_contradicting\_promise}(\mathit{prom}_i,\mathit{order}_i,\tau,s) \\ &\quad \wedge \texttt{keep\_promise}(\mathit{prom},\delta_i) \\ &\quad \wedge \texttt{promise\_made\_to}(\mathit{prom},\mathit{id}(\tau),s) \end{aligned}$$

and $\mathit{order}_i > \mathit{order}_j$ for all $i > j$. Notice that the order of the keep and postpone procedures is reversed to allow dealing with promises that depend upon other promises.

We can now put both helper functions together to modify the set of tasks with all *postpone* and *keep* procedures:

$$\begin{aligned} &\mathit{handle\_proms}(j,\Omega,s) = \Omega' \stackrel{\text{def}}{=} \\ &\quad \wedge \Omega'' = \mathit{keep\_postponed\_promises}(\Omega,s) \\ &\quad \wedge \Omega' = \mathit{postpone\_contradicting\_promises}(s,\Omega'',j) \end{aligned}$$

To handle promises when switching a task, we then extend Formula 6 to use this function after handling re-execution sequences. This leads us to the final version of the $TTrans$-rule for switching a task:

$$\begin{aligned} &TTrans(i,\Omega,s,i',\Omega',s) \subset \qquad (7) \\ &\quad (i = 0 \vee \texttt{switching\_allowed}[s]) \\ &\quad \wedge \exists \tau_j \in \Omega. \forall \tau_k \in \Omega. \mathit{prio}(\tau_k) \leq \mathit{prio}(\tau_j) \\ &\quad \wedge i' = \mathit{id}(\tau_j) \wedge (\exists \tau_i \in \Omega. \mathit{id}(\tau_i) = i \wedge \mathit{prio}(\tau_j) > \mathit{prio}(\tau_i) \\ &\quad \wedge \mathit{id}(\tau_i) \neq \mathit{id}(\tau_j) \vee i = 0) \\ &\quad \wedge \exists \tau_i'. \mathit{reset\_re}(\tau_i,\tau_i') \\ &\quad \underline{\wedge \Omega' = \mathit{handle\_proms}(\mathit{id}(\tau_j), \Omega[\tau_i = \tau_i'], s)} \end{aligned}$$

We then take the disjunction over all right hand sides of the formulas for adding, removing, executing and switching tasks to gain a formula for $TTrans$.

3.7 Online Execution

We adapt the definition of online execution presented in [3] to use $TTrans$ instead of $Trans$ and $TFinal$ instead of $Final$: Assume we started with a set of tasks Ω_0 and a current task i_0 (with $\tau_{i_0} \in \Omega_0$) in situation S_0. Then at some later point, we have executed actions $a_1,\ldots,a_k$ and have obtained sensing results $\mu_1,\ldots,\mu_k$ and therefore are now in history $\sigma = (a_1,\mu_1)\cdot\ldots\cdot(a_k,\mu_k)$ with the set of tasks Ω left to be executed, where one task $\tau_i \in \Omega$ is the current task. At each execution step, we need to decide what to do next. This can be either stopping execution if all tasks are finished, executing a step of the current task, or changing the set of tasks.

- stop, if
  $\mathcal{D} \cup \mathcal{C} \cup \mathcal{D}_{\mathcal{IL}} \cup \mathcal{C}_{\mathcal{IL}} \cup \mathit{Sensed}[\sigma] \models TFinal(\Omega, end[\sigma])$;
- return the set of remaining tasks Ω' and id i' of the current task, if
  $\mathcal{D} \cup \mathcal{C} \cup \mathcal{D}_{\mathcal{IL}} \cup \mathcal{C}_{\mathcal{IL}} \cup \mathit{Sensed}[\sigma] \models$
  $TTrans(i,\Omega,end[\sigma],i',\Omega',end[\sigma])$,
  and no action is required in this step;

- return action a, i and Ω', if
  $\mathcal{D} \cup \mathcal{C} \cup \mathcal{D}_{\mathcal{IL}} \cup \mathcal{C}_{\mathcal{IL}} \cup Sensed[\sigma] \models$
  $TTrans(i, \Omega, end[\sigma], i, \Omega', do(a, end[\sigma]))$.

Here $\mathcal{D}$ is an action theory for INDIGOLOG and $\mathcal{C}$ is a set of axioms defining the predicates $Trans$ and $Final$ as in INDIGOLOG. $\mathcal{D}_{\mathcal{IL}}$ contains the additional basic action theory axioms needed for the new built-in actions, fluents and procedures. $\mathcal{C}_{\mathcal{IL}}$ contains the axioms for $TTrans$ as described above and the new $TFinal$-rules described in Section 3.4. It also contains the extension of $Trans$ and $Final$ for re-execution sequences from Section 3.6.1. If the online execution returned an action, that action will be executed on the robot and sensing results are added to the history.

So analogously to [3] an online execution of a set of programs Ω and a current task number i starting from a history σ is defined as a sequence of online configurations $(i_0 = i, \Omega_0 = \Omega, \sigma_0 = \sigma), \ldots, (i_n, \Omega_n, \sigma_n)$ such that for $j = 0, \ldots, n-1$:

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{D}_{\mathcal{IL}} \cup \mathcal{C}_{\mathcal{IL}} \cup Sensed[\sigma_j] \models TTrans(i_j, \Omega_j, end[\sigma_j], i_{j+1}, \Omega_{j+1}, end[\sigma_{j+1}])$$

$$\sigma_{j+1} = \begin{cases} \sigma_j & \text{if } end[\sigma_{j+1}] = end[\sigma_j] \\ \sigma_j \cdot (a, \mu) & \text{if } end[\sigma_{j+1}] = do(a, end[\sigma_j]) \\ & \text{and } a \text{ returns } \mu. \end{cases}$$

With this formalization we can now prove that our approach is able to simulate INDIGOLOG.

Theorem 1. *Online execution of the program δ in* INDIGOLOG *is equivalent to executing the single task $\langle \delta, \texttt{prio}, 1 \rangle$ in* INTRGOLOG*, given the same sensing results and that no tasks are added or removed.*

Proof. We show, given the online execution of a program δ in INDIGOLOG starting from a history σ as a sequence of configurations $(\delta = \delta_0, \sigma = \sigma_0), \ldots, (\delta_n, \sigma_n)$, it holds for all $0 \leq i < n$ that

$$\mathcal{D} \cup \mathcal{C} \cup Sensed[\sigma_i] \models Trans(\delta_i, end[\sigma_i], \delta_{i+1}, end[\sigma_{i+1}])$$

if and only if

$$\mathcal{D} \cup \mathcal{C} \cup \mathcal{D}_{\mathcal{IL}} \cup \mathcal{C}_{\mathcal{IL}} \cup Sensed[\sigma_i] \models TTrans(1, \{\langle \delta_i, \texttt{prio}, 1 \rangle\}, end[\sigma_i], 1, \{\langle \delta_{i+1}, \texttt{prio}, 1 \rangle\}, end[\sigma_{i+1}]),$$

where $\mathcal{D}$, $\mathcal{C}$, $\mathcal{D}_{\mathcal{IL}}$ and $\mathcal{C}_{\mathcal{IL}}$ are defined as above.

We can show this by induction over the configuration steps $1 \ldots n$. The idea here is that because no new tasks are added, only the $TTrans$-rule for execution can be used: Since by prerequisite of the theorem no tasks are added or removed, the conditions on the right hand side of the Formulas 1 and 2 do not hold. Also, Formula 7 cannot be used since there is only one task that may not be removed and thus no task τ_j exists that fulfills this formula (e.g., $id(\tau_j) \neq id(\tau_i)$).

Thus, only the rule to execute the current task can be used. Since it uses the original $Trans$-rules this will yield the same results as applying the semantics of INDIGOLOG. □

4 EVALUATION

As a basis for our implementation we used the standard Prolog implementation of INDIGOLOG.[3] To show the possibilities offered by our approach we choose a domestic service robot scenario shown in Figure 1. The setting consists of two tasks:

Clean up The robot has to clean up a table in the living room by moving all cups on it to the dishwasher in the kitchen. Since the robot has only one arm it can only transport one cup at a time.

Answer Door The robot drives to the blue door and opens it.

We focus on the situation, where the robot is currently on the way from the table to the dishwasher with a cup in its hand when the doorbell rings. With INTRGOLOG the robot can drive to a nearby counter, put down the cup and then answer the door. As discussed in Section 2.2.5 it would be tedious, error prone and not scalable to use INDIGOLOG to successfully interrupt the clean-up task. Thus, the robot will drive to the dishwasher and finish loading the cup inside before answering the door. We compare the times needed to reach the door after the doorbell rang.

We tested the scenario on a real robot [4] as well as in simulation. The simulation is based on Gazebo [13] using a Robotino 3 robot. Benchmarks where made with the simulation on an Intel Core i7-3770 at 3.40 GHz with 4 cores and hyper-threading enabled.

To determine intermediate steps we use a promise `hand_used` and procedures to *postpone* and *keep* that promise as described in Section 3.6.2. For the clean up task the robot drives to a specified table. As long as there are cups on the table it grasps one and puts it in the dishwasher. Then it returns to the table and checks if there are any cups left (or new ones were put there):

```
proc(clean_up(Table), [
  exec_dur(drive_to(Table)),
  exec_dur(detect_cup),
  while(cups_on_table, [
    reexec([exec_dur(detect_cup),
            if(cups_on_table, [grasp], [])]),
    if(holding_cup, [
      reexec([exec_dur(drive_to(kitchen)),
              exec_dur(put_in_dishwasher)]),
      reexec([exec_dur(drive_to(Table)),
              exec_dur(detect_cup)])],
    [exec_dur(detect_cup)])])).
```

When answering the door, the robot drives to the door and opens it.

```
proc(answer_door, [ exec_dur(drive_to(door)),
                    exec_dur(open_door)]).
```

The doorbell is modeled as an exogenous action, that adds a task with the program `answer_door` and high priority.

To measure the improvement in reaction time, we compare to a simplified version of the above clean up program in INDIGOLOG. There, the robot only drives to the table once, picks up an object and puts it in the dishwasher. Afterwards, it answers the door:

```
proc(clean_up_once_indigolog(Table), [
  exec_dur(drive_to(Table)),
  exec_dur(detect_cup),
  if(cups_on_table, [grasp], []),
  if(holding_cup, [
    exec_dur(drive_to(kitchen)),
    exec_dur(put_in_dishwasher),
    answer_door], [])]).
```

The signal of the doorbell was given to both INDIGOLOG and INTRGOLOG randomly (but with a similar distribution) after the robot picked up the cup but had not yet placed it in the dishwasher.

[3] Project page at `https://bitbucket.org/ssardina/indigolog`

While INTRGOLOG is able to switch to the new task inserted by the exogenous doorbell action, INDIGOLOG performs its simplified procedure. The results are shown in Figure 3. Blue dots indicate the time INDIGOLOG needed from receiving the doorbell until arriving at the door, red squares the times for INTRGOLOG. The X-axis denotes time since the program start. When the doorbell rings early, the INTRGOLOG time is significantly shorter. In contrast, when the doorbell sounds close to arriving at the dishwasher, no significant time advantage can be achieved by putting the cup somewhere else than the dishwasher. On average, reaction times were about 15 % shorter. The experiment was run 700 times for both systems.

To make the comparison challenging we simplified the task significantly for the INDIGOLOG agent. Otherwise, it would have needed to drive back to the table and put away all cups before answering the door. This simple scenario already shows the potential of our approach. With INTRGOLOG the robot is able to react to new events promptly. In addition, it enables the robot to resume tasks after interruption. This makes the robot more reliable since given instructions are finished while unexpected events are handled appropriately.

5 RELATED WORK

Interleaved execution of processes has been used in operating systems to simulate parallel execution on a single CPU [21]. In contrast to operating systems, we do not want to simulate parallelism, because context switches in the real world are very time consuming. Our approach interleaves a task with a more important one.

Scheduling algorithms can be formalized in GOLOG [18]. However, the approach uses offline computation and cannot take into account new tasks or unexpected duration of tasks. Another scheduling-based approach is Temporal Flexible Golog [5], where low-level components of a robot are scheduled such that the execution fulfills time constraints between the components. However, INTRGOLOG focuses on tasks with intermediate steps between switching tasks. In our work the term *promise* denotes that we promise the programmer that a specified condition will hold when the program is executed, even if the task was interrupted in between. The term has already been used in a GOLOG context [15]. There the word is used in a multi-agent setting to describe that one agent promises another one that it will perform a request. Schiffer et al. [20] propose a self-maintenance system in READYLOG to ensure that during plan execution some predefined constraints are satisfied. They monitor the execution, detect events and try to fix occurring problems. In contrast, we focus on interleaving tasks such that resumption is possible.

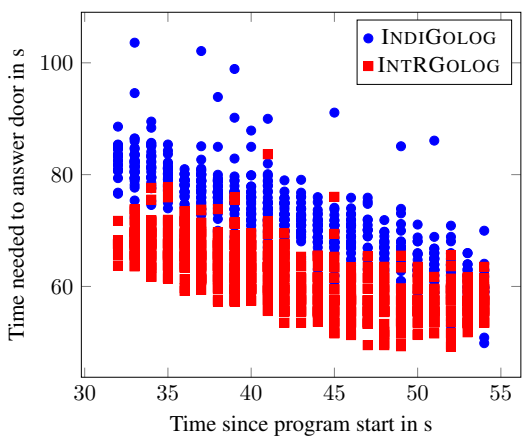

Figure 3. Time needed to arrive at the door since the doorbell rang.

A GOLOG-based approach is presented by Kelly and Pearce in [12], where the focus is on extending GOLOG with true concurrency in order to allow to coordinate multiple agents acting in parallel. However, they do not present a mechanism to deal with conflicts arising during the execution of concurrent programs akin to our promises.

Belief-Desire-Intention (BDI) systems like PRS [10] or AgentSpeak(L) [17] allow to execute multiple plans in an interleaved or parallel manner. In this context the challenges that arise with parallel execution of multiple plans have been discussed. In [1] summaries of concurrent hierarchical plans are used to identify associated preconditions and effects that can be used to reduce backtracking. Their formalism however only considers propositional, STRIPS-style actions and is less expressive than GOLOG, in particular regarding loops and recursive procedures. Harland et al. [8] propose to perform clean-up steps when suspending and resuming plans. The corresponding clean-up methods have to be defined for every plan. In comparison, our *promises* are action-specific; the intermediate steps necessary to suspend and resume a certain task are then implicit.

Haythem et al. [11] use a sequence of acts called *cascade* to detect failures and to remember what is left to be done after an interruption. They resume a cascade exactly where stopped and handle failures as new interrupts. In comparison, we determine intermediate steps in advance to avoid failures caused by interruptions.

6 CONCLUSION

In this work we described the problem of task interruption and resumption. We presented a first formalization of an extension of INDIGOLOG called INTRGOLOG to address some of the simpler cases of interruption and resumption. The introduction of the semantics predicates $TTrans$ and $TFinal$ allow our programming language to handle a set of tasks instead of one program. We introduced new constructs to determine intermediate steps when switching a task: The concept of *promises* defines asserted or required conditions of a running or interrupting task and *Re-execution sequences* allow to repeat certain parts of a program in full.

The evaluation showed that INTRGOLOG can react quickly to new events in real–world scenarios. As an example, interrupting a clean up task with an object in hand and putting it somewhere on the way leads on average to a 15 % latency reduction compared to bringing the object to its destination first. After reacting to an interrupting task, the robot can resume previous tasks without user intervention. Thereby, INTRGOLOG enables the robot to adapt to changing demands and new events in a swift and reliable fashion.

Opportunities for future work lie for instance in comparing the estimated time necessary for switching to the time for finishing the current task. For example, Gianni et al. use an estimation of *switching costs* to decide whether to react to stimuli with task switching [6]. This would avoid unnecessary or inefficient task switching, thus making our approach more efficient. Another promising possibility would be to integrate continual planning in GOLOG [9] with our approach. This would allow to recover promises in a more generic fashion without specific procedures for each promise. It would also provide more robustness if keeping a promise fails, e.g., because a cup cannot be retrieved since it was moved by someone else.

ACKNOWLEDGEMENTS

This work was supported by the German National Science Foundation (DFG) research unit FOR 1513 on Hybrid Reasoning for Intelligent Systems (http://www.hybrid-reasoning.org).

REFERENCES

[1] Bradley J Clement and Edmund H Durfee, 'Theory for coordinating concurrent hierarchical planning agents using summary information', in *AAAI Conference on Artifical Intelligence (AAAI)*, (1999).

[2] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, 'ConGolog, a concurrent programming language based on the situation calculus', *Artificial Intelligence*, **121**(1), (2000).

[3] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina, 'IndiGolog: A high-level programming language for embedded reasoning agents', *Multi-Agent Programming: Languages, Tools and Applications*, (2009).

[4] Alexander Ferrein, Tim Niemueller, Stefan Schiffer, and Gerhard Lakemeyer, 'Lessons learnt from developing the embodied AI platform Caesar for domestic service robotics', in *AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI II*, (2013).

[5] Alberto Finzi and Fiora Pirri, 'Switching tasks and flexible reasoning in the situation calculus', *Department of Computer and System Sciences Antonio Ruberti Technical Reports*, **2**(7), (2010).

[6] Mario Gianni, Panagiotis Papadakis, and Fiora Pirri, 'Shifting and inhibition in cognitive control', in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS) - Workshop on Cognitive Neuroscience Robotics*, (2012).

[7] Henrik Grosskreutz and Gerhard Lakemeyer, 'cc-Golog: Towards more realistic logic-based robot controllers', in *AAAI Conference on Artificial Intelligence (AAAI)*, (2000).

[8] James Harland, David N Morley, John Thangarajah, and Neil Yorke-Smith, 'Aborting, suspending, and resuming goals and plans in BDI agents', *Autonomous Agents and Multi-Agent Systems*, (2015).

[9] Till Hofmann, Tim Niemueller, Jens Claßen, and Gerhard Lakemeyer, 'Continual planning in Golog', in *AAAI Conference on Artificial Intelligence (AAAI)*, (2016).

[10] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert, 'PRS: A high level supervision and control language for autonomous mobile robots', in *IEEE Int. Conf. on Robotics and Automation (ICRA)*, (1996).

[11] Haythem O. Ismail and Stuart C. Shapiro, 'Conscious error recovery and interrupt handling', in *Proc. of the Int. Conf. on Artificial Intelligence (ICAI)*, (2000).

[12] Ryan F. Kelly and Adrian R. Pearce, 'Towards high-level programming for distributed problem solving', in *IEEE/WIC/ACM Int. Conf. on Intelligent Agent Technology (IAT)*, (2006).

[13] Nathan Koenig and Andrew Howard, 'Design and use paradigms for gazebo, an open-source multi-robot simulator', in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, (2004).

[14] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *The Journal of Logic Programming*, **31**(1), (1997).

[15] Yisong Liu, Lili Dong, and Yamin Sun, 'Cooperation model of multi-agent system based on the situation calculus', in *IEEE/WIC/ACM Int. Conf. on Intelligent Agent Technology (IAT)*, (2006).

[16] John McCarthy and Patrick Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', *Readings in Artificial Intelligence*, (1969).

[17] Anand S. Rao, 'AgentSpeak(L): BDI agents speak out in a logical computable language', in *Agents Breaking Away*, Springer, (1996).

[18] Ray Reiter and Zheng Yuhua, 'Scheduling in the situation calculus: A case study', *Annals of Mathematics and Artificial Intelligence*, **21**(2-4), (1997).

[19] Raymond Reiter, *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, MIT press, 2001.

[20] Stefan Schiffer, Andreas Wortmann, and Gerhard Lakemeyer, 'Self-Maintenance for Autonomous Robots controlled by ReadyLog', in *IARP Workshop on Technical Challenges for Dependable Robots in Human Environments*, (2010).

[21] Andrew S. Tanenbaum, *Modern operating systems*, volume 3, Pearson Prentice Hall, 2009.