concurrency, priorities etc. and, at the same time, supports rigorous projections of plans[4] because it is entirely based on the situation calculus (McCarthy 1963).

It turns out, however, that despite many similarities, ConGolog in its current form is not suitable to represent robot controllers such as the example above. The main problem is that the existing temporal extensions of the situation calculus such as (Pinto 1997; Reiter 1996) require that the execution time of an action is supplied explicitly, which seems incompatible with event-driven specifications. To solve this problem we proceed in two steps. First we present a new extension of the situation calculus which, besides dealing with continuous change, allows us to model actions which are event-driven by including *waitFor* as a special action in the logic. We then turn to a new variant of ConGolog called cc-Golog, which is based on the extended situation calculus. We study issues arising from the interaction of *waitFor*-actions and concurrency and show how the example-program can be specified quite naturally in cc-Golog with the additional benefit of supporting projections firmly grounded in logic.

The rest of the paper is organized as follows. In the next section, we briefly review the basic situation calculus. Then we show how to extend it to include continuous change and time. After a very brief summary of ConGolog, we present cc-Golog, which takes into account the extended situation calculus. This is followed by a note on experimental results and conclusions.

The Situation Calculus

One increasingly popular language for representing and reasoning about the preconditions and effects of actions is the situation calculus (McCarthy 1963). We will not go over the language in detail except to note the following features: all terms in the language are one of three sorts, ordinary objects, actions or situations; there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation are called relational *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; similarly, functions varying across situations are called functional fluents and are denoted analogously; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form (Levesque, Pirri, & Reiter 1998):

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as a function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem (Levesque, Pirri, & Reiter 1998).
- Domain closure and unique name axioms for the actions.
- Foundational, domain independent axioms (Levesque, Pirri, & Reiter 1998).

1. $\forall P.P(S_0) \wedge [\forall s \forall a.(P(s) \supset P(do(a, s)))] \supset \forall s P(s)$;
2. $do(a, s) = do(a', s') \supset a = a' \wedge s = s'$;[5]
3. $\neg(s \sqsubset S_0)$;
4. $s \sqsubset do(a, s') \equiv s \sqsubseteq s'$, where $s \sqsubseteq s'$ stands for $(s \sqsubset s') \vee (s = s')$.
5. $s \prec s' \equiv s \sqsubset s' \wedge \forall a, s^*. s \sqsubset do(a, s^*) \sqsubseteq s' \supset Poss(a, s^*)$

The first is a second-order induction axiom ensuring that the only situations are those obtained from applying *do* to S_0. The second is a unique names axiom for situations. $\sqsubseteq$ defines an ordering relation over situations. Intuitively, $s \sqsubseteq s'$ holds if s' can be obtained from s by performing a finite number of actions. Finally, $s \prec s'$ holds when there is a *legal* sequence of actions leading from s to s', where legal means that each action is possible.

Continuous Change and Time

Actions in the situation calculus cause discrete changes and, in its basic form, there is no notion of time. In robotics applications, however, we are faced with processes such as navigation which cause properties like the robot's location and orientation to change continuously over time. In order to model such processes in the situation calculus in a natural way, we add continuous change and time directly to its ontology.

As demonstrated by Pinto and Reiter (Pinto 1997; Reiter 1996), adding time is a simple matter. We add a new sort *real* ranging over the real numbers and, for mnemonic reasons, another sort *time* ranging over the reals as well.[6] In order to connect situations and time, we add a special unary functional fluent *start* to the language with the understanding that $start(s)$ denotes the time when situation s begins. We will see later how *start* obtains its values and, in particular, how the passage of time is modeled.

A fundamental assumption of the situation calculus is that fluents have a fixed value at every given situation. In order to see that this assumption still allows us to model continuous change, let us consider the example of a mobile robot moving along a straight line in a 1-dimensional world, that is, the robot's location at any given time is simply a real number. There are two types of actions the robot can perform, $startGo(v)$, which initiates moving the robot with speed *v*, and *endGo* which stops the movement of the robot. Let us denote the robot's location by the fluent *robotLoc*. What should the value of *robotLoc* be after executing $startGo(v)$ in situation s? Certainly it cannot be a fixed real value, since the position should change over time as long as the robot moves. In fact, the location of the robot at any time after $startGo(v)$ (and before the robot changes its velocity) can be

[4] In this paper we will use the terms program and plan interchangeably, following McDermott (McDermott 1992) who takes plans to be programs whose execution can be reasoned about by the agent who executes the program.

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

[6] For simplicity, the reals are not axiomatized and we assume their standard interpretations together with the usual operations and ordering relations.

characterized (in a somewhat idealized fashion) by the function $x + v \times (t - t_0)$, where x is the starting position and t_0 the starting time. The solution is then to take this *function of time* to be the value of *robotLoc*. We call functional fluents whose values are continuous functions *continuous fluents*.

The idea of continuous fluents, which are often called *parameters*, is not new. Sandewall (Sandewall 1989) proposed it when integrating the differential equations into logic, Galton (Galton 1990) investigated similar issues within a temporal logic, and Shanahan considers continuous change in the event calculus (Shanahan 1990). Finally, Miller and Pinto (Miller 1996; Pinto 1997) formulate continuous change in the situation calculus. Here we essentially follow Pinto, in a somewhat simplified form.

We begin by introducing a new sort t-function, whose elements are meant to be functions of time. We assume that there are only finitely many function symbols of type t-function and we require *domain closure and unique names axioms* for them, just as in the case of primitive actions. For our robot example, it suffices to consider two kinds of t-functions: constant functions, denoted by $\mathit{constant}(x)$ and the special linear functions introduced above, which we denote as $\mathit{linear}(x, v, t_0)$.

Next we need to say what values these functions have at any particular time t. We do this with the help of a new binary function *val*. In the example, we would add the following axioms:

$\mathit{val}(\mathit{constant}(x), t) = x$;
$\mathit{val}(\mathit{linear}(x, v, t_0), t) = x + v \times (t - t_0)$.

Let us now turn to the issue of modeling the passage of time during a course of actions. As indicated in the introduction, motivated by the treatment of time in robot control languages like RPL, RAP, or COLBERT, we introduce a new type of primitive action $\mathit{waitFor}(\phi)$. The intuition is as follows. Normally, every action happens immediately, that is, the starting time of the situation after doing a in s is the same as the starting time of s. The only exception is $\mathit{waitFor}(\phi)$: whenever this action occurs, the starting time of the resulting situation is advanced to the earliest time in the future when ϕ becomes true. Note that this has the effect that actions always happen as soon as possible. One may object that requiring that two actions other than *waitFor* must happen at the same time is unrealistic. However, in robotics applications, actions often involve little more than sending messages in order to initiate or terminate processes so that the actual duration of such actions is negligible. Moreover, if two actions cannot happen at the same time, they can always be separated explicitly using *waitFor*.

For the purposes of this paper, we restrict the argument of *waitFor* to what we call a t-form, which is a Boolean combination of closed atomic formulas of the form $(F\ op\ r)$, where F is a continuous fluent with the situation argument suppressed, $op \in \{<, =\}$,[7] and r is a term of type real (not mentioning *val*). An example is $\phi = (\mathit{robotLoc} \geq 1000)$. To evaluate a t-form at a situation s and time t, we write $\phi[s, t]$ which results in a formula which is like ϕ except that every continuous fluent F is replaced by $\mathit{val}(F(s), t)$. For instance, $(\mathit{robotLoc} \geq 1000)[s, t]$ becomes $(\mathit{val}(\mathit{robotLoc}(s), t) \geq 1000)$. For reasons of space we completely gloss over the details of reifying t-forms within the language[8] except to note that we introduce t-forms as a new sort and that $\phi[s, t]$ is short for $\mathit{Holds}(\phi, s, t)$, where *Holds* is appropriately axiomatized.

To see how actions are forced to happen as soon as possible, let $\mathit{ltp}(\phi, s, t)$ be an abbreviation for the formula

$$\phi[s, t] \wedge t \geq \mathit{start}(s) \wedge \forall t'.\mathit{start}(s) \leq t' < t \supset \neg\phi[s, t'],$$

that is, t in $\mathit{ltp}(\phi, s, t)$ is the least time point after the start of s where ϕ becomes true. Then we require that a *waitFor*-action is possible iff the condition has a least time point. In practice, this means that it is up to the user to ensure that ϕ has a least time point.

$$\mathit{Poss}(\mathit{waitFor}(\phi), s) \equiv \exists t.\mathit{ltp}(\phi, s, t).$$

It is not hard to show that, if $\exists t.\mathit{ltp}(\phi, s, t)$ is satisfied, then t is unique.

We remark that Reiter (1996) introduced a related concept of *least natural time point*.[9] Note, however, that Reiter only considers natural actions (like the bouncing of a ball) which happen whenever they are (physically) *possible*. In contrast, we are concerned with actions which are under the control of the agent including deliberately waiting for an event to occur or to ignore it.

Finally, we need to characterize how the fluent *start* changes its value when an action occurs. The following successor state axiom for *start* captures the intuition that the starting time of a situation changes only as a result of a $\mathit{waitFor}(\phi)$, in which case it advances to the earliest time in the future when ϕ holds.

$$\begin{aligned} &Poss(a, s) \supset [\mathit{start}(do(a, s)) = t \equiv \\ &\quad \exists \phi.a = \mathit{waitFor}(\phi) \wedge \mathit{ltp}(\phi, s, t) \vee \\ &\quad [\forall \phi.a \neq \mathit{waitFor}(\phi) \wedge t = \mathit{start}(s)]]. \end{aligned}$$

Let AX be the set of foundational axioms of the previous section together with the domain closure and unique names axioms for t-functions, the axioms required for t-form's, the precondition axiom for *waitFor*, and the successor state axiom for *start*. Then the following formulas are logical consequences of AX.

Proposition 1:

1. *The starting time of legal action sequences is monotonically nondecreasing:*
   $\forall s, s'.s \prec s' \supset \mathit{start}(s) \leq \mathit{start}(s')$.
2. *Actions happen as soon as possible:*
   $[\forall a, s.\mathit{start}(do(a, s)) = \mathit{start}(s)] \vee [\exists \phi.a = \mathit{waitFor}(\phi) \wedge \mathit{ltp}(\phi, s, \mathit{start}(do(a, s)))]$

To illustrate the approach, let us go back to the robot example. First, we can formulate a successor state axiom for *robotLoc*:

$$\begin{aligned} &Poss(a, s) \supset [\mathit{robotLoc}(do(a, s)) = y \equiv \\ &\quad \exists t_0, v, x.x = \mathit{val}(\mathit{robotLoc}(s), t_0) \wedge t_0 = \mathit{start}(s) \wedge \\ &\quad ([a = \mathit{startGo}(v) \wedge y = \mathit{linear}(x, v, t_0)] \\ &\qquad \vee [a = \mathit{endGo} \wedge y = \mathit{constant}(x)] \vee [y = \mathit{robotLoc}(s) \wedge \\ &\qquad \neg(\exists v.a = \mathit{startGo}(v) \vee a = \mathit{endGo})])] \end{aligned}$$

In other words, when an action is performed *robotLoc* is assigned either the function $\mathit{linear}(x, v, t_0)$, if the robot starts moving with velocity v and x is the location of the robot at situation s, or it is assigned $\mathit{constant}(x)$ if the

[7] We freely use $\leq$, $\geq$, and $>$ as well.

[8] See, for example, (Giacomo, Lesperance, & Levesque 1999) for details how this can be done.

[9] Similar ideas occur in the context of delaying processes in real-time programming languages like Ada (Burns & Wellings 1991).

robot stops, or it remains the same as in s. Note that $val(robotLoc(s), t_0)$ is well-defined since every t-function has a name (*constant* or *linear*) with corresponding axioms for *val* as given above.

Let Σ be AX together with the axioms for *val*, the successor state axiom for *robotLoc*, precondition axioms stating that *startGo* and *endGo* are always possible, and the fact $(robotLoc(S_0) = constant(0))$, that is, the robot initially rests at position 0. Let us assume the robot starts moving at speed 50 (cm/s) and then waits until it reaches location 1000 (cm), at which point it stops. The resulting situation is $s_1 = do(endGo, do(waitFor(robotLoc = 1000), do(startGo(50), S_0)))$. Then

$\Sigma \models start(s_1) = 20 \wedge robotLoc(s_1) = constant(1000)$.

In other words, the robot moves for 20 seconds and stops at location 1000, as one would expect.

In summary, to model continuous change and time in the situation calculus, we have added four new sorts: real, time, t-function (functions of time), and t-form (temporal formulas). In addition, we introduced a special function *val* to evaluate t-functions, a new kind of primitive action *waitFor* together with a domain-independent precondition axiom, and a new fluent *start* (the starting time of a situation) together with a successor state axiom.

ConGolog

ConGolog (Giacomo, Lesperance, & Levesque 1999), an extension of GOLOG (Levesque *et al.* 1997), is a formalism for specifying complex actions and how these are mapped to sequences of atomic actions assuming a description of the initial state of the world, action precondition axioms and successor state axioms for each fluent. Complex actions are defined using control structures familiar from conventional programming language such as sequence, while-loops and recursive procedures. In addition, parallel actions are introduced with a conventional interleaving semantics. Here we confine ourselves to the deterministic fragment of ConGolog. (While nondeterministic actions raise interesting issues, we ignore them for reasons of space. Also note that nondeterminism plays little if any role in languages like RPL.)

α	primitive action
$\phi?$	test action[10]
$seq(\sigma_1, \sigma_2)$	sequence
$if(\phi, \sigma_1, \sigma_2)$	conditional
$while(\phi, \sigma)$	loop
$par(\sigma_1, \sigma_2)$	concurrent execution
$prio(\sigma_1, \sigma_2)$	prioritized execution
proc $\beta(x)\sigma$	procedure definition

The semantics of ConGolog is defined using the so-called transition semantics, which defines single steps of computation. There is a relation, denoted by the predicate $Trans(\sigma, s, \delta, s')$, that associates with a given program σ and situation s a new situation s' that results from executing a primitive action in s, and a new program δ that represents what remains of σ after having performed that action. Furthermore, we need to define which configurations (σ, s) are final, meaning that the computation can be considered completed when a final configuration is reached. This is denoted by the predicate $Final(\sigma, s)$.[11].

For space reasons, we only list a few of the axioms for $Final$ and $Trans$. Note that the semantics is defined for the *non-temporal* situation calculus. Adapting the semantics to the temporal situation calculus of the previous section will be the subject of the next section.

$Final(\alpha, s) \equiv false$, where α is a primitive action
$Final(nil, s) \equiv true$, where nil is the empty program
$Final(\phi?, s) \equiv false$
$Final(if(\phi, \sigma_1, \sigma_2), s) \equiv$
  $\phi[s] \wedge Final(\sigma_1, s) \vee \neg\phi[s] \wedge Final(\sigma_2, s)$
$Final(par(\sigma_1, \sigma_2), s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$
$Final(prio(\sigma_1, \sigma_2), s) \equiv Final(\sigma_1, s) \wedge Final(\sigma_2, s)$

$Trans(\alpha, s, \delta, s') \equiv$
  $Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s)$
$Trans(nil, s, \delta, s') \equiv false$
$Trans(\phi?, s, \delta, s') \equiv \phi[s] \wedge \delta = nil \wedge s' = s$
$Trans(seq(\sigma_1, \sigma_2), s, \delta, s') \equiv$
  $Final(\sigma_1, s) \wedge Trans(\sigma_2, s, \delta, s') \vee$
  $\exists\delta'.Trans(\sigma_1, s, \delta', s') \wedge \delta = seq(\delta', \sigma_2)$
$Trans(if(\phi, \sigma_1, \sigma_2), s, \delta, s') \equiv$
  $\phi[s] \wedge Trans(\sigma_1, s, \delta, s') \vee$
  $\neg\phi[s] \wedge Trans(\sigma_2, s, \delta, s')$
$Trans(par(\sigma_1, \sigma_2), s, \delta, s') \equiv$
  $\exists\gamma.\delta = par(\gamma, \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee$
  $\exists\gamma.\delta = par(\sigma_1, \gamma) \wedge Trans(\sigma_2, s, \gamma, s')$

Intuitively, a program cannot be in its final state if there is still a primitive action to be done. Similarly, a concurrent execution of two programs is in its final state if both are. As for $Trans$, let us just look at par: a transition of two programs working in parallel means that one action of one of the programs is performed.

A final situation s' reachable after a finite number of transitions from a starting situation is identified with the situation resulting from a possible execution trace of program σ, starting in situation s; this is captured by the predicate $Do(\sigma, s, s')$, which is defined in terms of $Trans^*$, the transitive closure of $Trans$:

$Do(\delta, s, s') \equiv \exists\delta'.Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$
$Trans^*(\delta, s, \delta', s') \equiv \forall T[... \supset T(\delta, s, \delta', s')]$

where the ellipsis stands for the universal closure of the conjunction of the following formulas:

$T(\delta, s, \delta, s)$
$Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s')$

Given a program δ, proving that δ is executable in the initial situation then amounts to proving $\Sigma \models \exists s Do(\delta, S_0, s)$, where Σ consists of the above axioms for ConGolog together with a basic action theory in the situation calculus.

cc-Golog: a Continuous, Concurrent Golog

Let us now turn to cc-Golog, which is a variant of deterministic ConGolog and which is founded on our new extension of the situation calculus.

[10]Here, ϕ stands for a situation calculus formula with all situation arguments suppressed. $\phi[s]$ will denote the formula obtained by restoring situation variable s to all fluents appearing in ϕ.

[11]Again, we gloss over the issue of reifying formulas and programs in the logical language and refer to (Giacomo, Lesperance, & Levesque 1999) for details.

First, for reasons discussed below we slightly change the language by replacing the instructions par and $prio$ by the constructs $tryAll$ and $withPol$, respectively. Intuitively, $tryAll(\sigma_1, \sigma_2)$ starts executing both σ_1 and σ_2; but unlike par, which requires both σ_1 and σ_2 to reach a final state, the parallel execution of $tryAll$ stops as soon as *one* of them reaches a final state. As for $withPol(\sigma_1, \sigma_2)$, the idea is that a low priority plan σ_2 is executed, which is interrupted whenever the program σ_1, which is called a *policy*, is able to execute. The execution of the whole $withPol$ construct ends as soon as σ_2 ends. (Note that $prio$ is just like $withPol$ except that for $prio$ to end both σ_1 and σ_2 need to have ended.)

$tryAll$ and $withPol$ are inspired by similar instructions in RPL where they have been found very useful in specifying complex concurrent behavior. In particular, $withPol$ is useful to specify the execution of a plan while guarding certain constraints. As we will see later, it is quite straightforward to define par and $prio$ using the new instructions. On the other hand, defining $tryAll$ and $withPol$ in terms of par and $prio$ appears to be more complicated. Hence we decided to trade par and $prio$ for their siblings.

Let us now turn to the semantics of cc-Golog, which means finding appropriate definitions for $Final$ and $Trans$. To start with, the semantics remains exactly the same for all those constructs inherited from deterministic ConGolog. Note that this is also true for the new *waitFor*(ϕ), which is treated like any other primitive action.[12] Hence we are left to deal with $tryAll$ and $withPol$.

It is straightforward to give $Final$ its intended meaning, that is, $tryAll$ ends if one of the two programs ends and $withPol$ ends if the second program ends:

$$Final(tryAll(\sigma_1, \sigma_2, s)) \equiv Final(\sigma_1, s) \lor Final(\sigma_2, s)$$
$$Final(withPol(\sigma_1, \sigma_2), s) \equiv Final(\sigma_2, s)$$

When considering the transition of concurrent programs, care must be taken in order to avoid conflicts with the assumption that actions should happen as soon as possible, which underlies our new version of the situation calculus. To see why let us consider the following example, where we want to instruct our robot to run a backup at time 8 or 20, whichever comes first. Let us assume we have a continuous fluent *clock* representing time[13] and let *runBackup* be always possible. Given our intuitive reading of $tryAll$, we may want to use the following program:

$$seq(tryAll(waitFor(clock = 8), waitFor(clock = 20)), runBackup)$$

If we start the program at time 0 we would expect to see

[*waitFor*(*clock*=8),*runBackup*]

as the only execution trace, since time 8 is reached first. (Recall that $tryAll$ finishes as soon as one of its arguments finishes.) However, this is not necessarily guaranteed. In fact, the obvious adaptation of ConGolog's $Trans$-definition of par to the case of $tryAll$[14] also yields the trace

[*waitFor*(*clock*=20),*runBackup*].

This is because there simply is no preference enforced between the two *waitFor*-actions. As the following definition shows, it is not hard to require that actions which can be executed earlier are always preferred, restoring the original idea that actions should happen as soon as possible.

$$\begin{aligned}
&Trans(tryAll(\sigma_1, \sigma_2), s, \delta, s') \equiv \\
&\quad \neg Final(\sigma_1, s) \land \neg Final(\sigma_2, s) \land \\
&\qquad \exists \delta_1 . Trans(\sigma_1, s, \delta_1, s') \land \delta = tryAll(\delta_1, \sigma_2) \land \\
&\qquad \forall \delta_2, s_2 . Trans(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)] \\
&\quad \lor \exists \delta_2 . Trans(\sigma_2, s, \delta_2, s') \land \delta = tryAll(\sigma_1, \delta_2) \land \\
&\qquad \forall \delta_1, s_1 . Trans(\sigma_1, s, \delta_1, s_1) \supset start(s') \leq start(s_1)]
\end{aligned}$$

We are left with defining $Trans$ for $withPol$. To see what is involved, let us consider the following example

$withPol(watchB, deliverMail)$, where

$watchB = seq(waitFor(battLevel \leq 46), chargeBatt)$.

The idea is to deliver mail and, with higher priority, watch for a low battery level, at which point the batteries are charged. In the discussion of a similar scenario written in RPL, we already pointed out that the *waitFor*-action should not block the mail delivery even though it belongs to the high priority policy. On the other hand, once the routine for charging the batteries starts, it should not be interrupted, that is, it should run in blocking mode, which should also hold for possible *waitFor*-actions it may contain such as waiting for arrival at the docking station. It turns out that it suffices to arrange in the semantics of $Trans$ that occurrences of *waitFor* within a policy are considered non-blocking. As we will see below, the effect of a policy running in blocking mode is definable by other means.

Interestingly, the resulting axiom is almost identical to that of $tryAll$: the main difference is that $\leq$ is replaced by $<$ in the last line. This ensures that σ_1 takes precedence if both σ_i are about to execute an action at the same time.

$$\begin{aligned}
&Trans(withPol(\sigma_1, \sigma_2), s, \delta, s') \equiv \neg Final(\sigma_2, s) \land \\
&\qquad \exists \delta_1 . Trans(\sigma_1, s, \delta_1, s') \land \delta = withPol(\delta_1, \sigma_2) \land \\
&\qquad \forall \delta_2, s_2 . Trans(\sigma_2, s, \delta_2, s_2) \supset start(s') \leq start(s_2)] \\
&\quad \lor \exists \delta_2 . Trans(\sigma_2, s, \delta_2, s') \land \delta = withPol(\sigma_1, \delta_2) \land \\
&\qquad \forall \delta_1, s_1 . Trans(\sigma_1, s, \delta_1, s_1) \supset start(s') < start(s_1)]
\end{aligned}$$

This then ends the discussion of the semantics of $Trans$ in cc-Golog. $Trans^*$ and $Do(\delta, s, s')$ are defined the same way as in ConGolog.

One issue left open is to show how a policy can run in blocking mode. This can be arranged using the macro $withCtrl(\phi, \sigma)$, which stands for σ with every primitive action or test α replaced by $if(\phi, \alpha, false?)$.[15]

Intuitively $withCtrl(\phi, \sigma)$ executes σ as long as ϕ is true, but gets blocked otherwise. As the following example shows, the effect of a policy in blocking mode is obtained by having the truth value of ϕ be controlled by the policy and using the $withCtrl(\phi, \sigma)$-construct in the low priority program.

This leads us, finally, to the specification of our initial example in cc-Golog. In the following we assume a fluent $wheels$, which is initially *true*, set *false*

[12]The reader familiar with ConGolog may wonder whether a test action $\phi?$ is the same as *waitFor*(ϕ). This is not so. Roughly, the main difference is that tests have no effect on the world while *waitFor* advances the time.

[13]This can be modeled using a simple linear function, but we ignore the details here.

[14]Roughly, replace par by $tryAll$ and add $\neg Final(\sigma_1, s)$ and $\neg Final(\sigma_2, s)$ as additional conjuncts in the definition's R.H.S.

[15]We remark that $if(\phi, \alpha, false?)$ can only lead to a transition if ϕ is true in the current situation at which point α is executed immediately. This is essentially due to the fact that $false?$ is neither $Final$ nor can it ever lead to a transition.

by $grabWhls$, and reset by the action $releaseWhls$. We also use $whenever(\phi, \sigma)$ as shorthand for $while(true, seq(waitFor(\phi), \sigma))$.

$$withPol(whenever(battLevel \leq 46, \\ seq(grabWhls, \mathit{chargeBatteries}, releaseWhls)^{16}), \\ withPol(whenever(nearDoor\text{A-118}, \\ seq(say(hello), waitFor(\neg nearDoor\text{A-118}))) \\ withCtrl(wheels, deliverMail)))$$

Figure 2: The introductory example as a cc-Golog plan.

In this program, the outermost policy is waiting until the battery level drops to 46. At this point, a $grabWheels$ is immediately executed, which blocks the execution of the program $deliverMail$. It is only after the complete execution of *chargeBatteries* that $wheels$ gets released so that $deliverMail$ may resume execution (if, while driving to the battery docking station, the robot passes by Room A-118, it would still say "hello").

Note that the cc-Golog-program is in a form very close to the original RPL-program we started out with. Hence we feel that cc-Golog is a step in the right direction towards modeling more realistic domains which so far could only be dealt with in non-logic-based approaches. Moreover, with their rigorous logical foundation, it is now possible to make provable predictions about how the world evolves when executing cc-Golog-programs. (See also the next section on experimental results.)

Finally, let us briefly consider how par and $prio$ which we dropped in favor of $tryAll$ and $withPol$ are definable within cc-Golog. Let us assume fluents flg_i which are initially *false* and set *true* by $setFlg_i$. Then we can achieve what amounts to $par(\sigma_1, \sigma_2)$ by $tryAll(seq(\sigma_1, setFlg1, flg2?), seq(\sigma_2, setFlg2, flg1?))$. Note that the testing of the flags at the end of each program forces that both σ_i need to finish. Similarly, $prio$ can be defined as $withPol(seq(\sigma_1, setFlg), seq(\sigma_2, flg?))$.

We end this section with some remarks on Reiter's proposal for a temporal version of GOLOG (Reiter 1998),[17] which makes use of a different temporal extension of the situation calculus (Pinto 1997; Reiter 1996). Roughly, the idea is that every primitive action has as an extra argument its execution time. E.g., we would write *endGo*(20) to indicate that *endGo* is executed at time 20. It turns out that this explicit mention of time is highly problematic when it comes to formulating programs such as the above. Consider the part about saying "hello" whenever the robot is near Room A-118. In Reiter's approach, the programmer would have to supply a temporal expression as an argument of the *say*-action. However, it is far from obvious what this expression would look like since it involves analyzing the mail delivery subprogram as well as considering the odd chance of a battery recharge. In a nutshell, while Reiter's approach forces the user to figure out when to act, we let cc-Golog do the work. — As a final aside, we remark that *waitFor*-actions allow us to easily emulate Reiter's approach within our framework.

[16] $seq(\sigma_1, \sigma_2, \sigma_3)$ is a shorthand for $seq(\sigma_1, seq(\sigma_2, \sigma_3))$. We will also use a similar shorthand for $tryAll$.

[17] While the paper is about sequential GOLOG, the extension to ConGolog is straightforward.

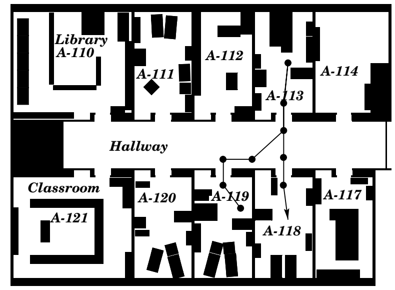

Figure 3: The robot environment

Experimental Results

Although the definition of cc-Golog requires second-order logic, it is easy to implement a PROLOG interpreter for cc-Golog, just as in the case of the original ConGolog.[18] In order to deal with the constraints implied by the $waitFor$ instruction, we have made use of the ECRC Common Logic Programming System Eclipse 4.2 and its built-in constraint solver library (similar to Reiter (Reiter 1998)).

In order to evaluate the performance of our cc-Golog interpreter, we used it to project the (slightly modified) example of (Beetz & Grosskreutz 1998), where the XFRM framework (McDermott 1992; 1994) is used to deal with continuous change. Here, a mobile robot is to deliver letters in the environment depicted in Figure 3. At the same time, it has to monitor the state of the environment, that is, it has to check whether doors are open. As soon as it realizes that the door to A-113 is open, it has to interrupt its actual delivery in order to deliver an urgent letter to A-113. This is specified as a policy that leads the robot inside A-113 as soon as the opportunity is recognized. Similar to (Beetz & Grosskreutz 1998)), we approximated the robot's trajectory by polylines, consisting of the starting location, the goal location and a point in front of and behind every passed doorway.

```
withPol(whenever(inHallway,
  [say(enterHW),
    tryAll(
      [whenever(nearDoor(a-114+117),
        [chkDr(a114),chkDr(a117),false?])],
      [whenever(nearDoor(a-113+118),
        [chkDr(a113),chkDr(a118),false?])],
        ...
      [waitFor(leftHallway),say(leftHW)])]),
   withPol([useOpp?,gotoRoom(a-113),
           deliverUrgentMail],
     [gotoRoom(a-118),giveMail(gerhard)]))).
```

Note that in the implementation, we used PROLOG lists (`[a1,a2,...]`) instead of $seq(a_1, a_2, ...)$. The outer policy is activated whenever the robot enters the hallway. It concurrently monitors whether the robot reaches a location near two opposite doors, at which point it checks whether the doors are open or not. If A-113 is detected to be open, the fluent `useOpp` is set true (by procedure `chkDr`). The policy is deactivated when the robot leaves the hallway. The inner policy is activated as soon as `useOpp` gets true. Its purpose is to use the opportunity to enter A-113 as soon as

[18] The subtle differences between the second order axiomatization of ConGolog and a PROLOG implementation are discussed in (Giacomo, Lesperance, & Levesque 1999).

possible. Figure 3 illustrates the projected trajectory starting in Room A-119, assuming that the door to A-113 is indeed open.

The projection of this plan took 0.5 seconds using cc-Golog, while the projection of the corresponding RPL program took 3.6 seconds. Both cc-Golog and XFRM ran on the same machine (a Linux Pentium III Workstation), under Allegro Common Lisp resp. Eclipse 4.2. We believe that cc-Golog owes this somewhat surprising advantage to the fact that it lends itself to a simple implementation with little overhead, while XFRM relies on the rather complex RPL-interpreter involving many thousand lines of Lisp code. Finally, and maybe most importantly, the cc-Golog implementation is firmly based on a logical specification, while XFRM relies on the procedural semantics of the RPL interpreter.

Conclusions

In this paper we proposed an extension of the situation calculus which includes a model of continuous change due to Pinto and a novel approach to modeling the passage of time using a special *waitFor*-action. We then considered cc-Golog, a deterministic variant of ConGolog which is based on the extended situation calculus. A key feature of the new language is the ability to have part of a program wait for an event like the battery voltage dropping dangerously low while other parts of the program run in parallel. Such mechanisms allow very natural formulations of robot controllers, in particular, because there is no need to state explicitly in the program when actions should occur. In addition to the sound theoretical foundations on which cc-Golog is built, experimental results have shown a superior performance in computing projections when compared to the projection mechanism of the plan language RPL, whose expressive power has largely motivated the development of cc-Golog.

Finally, a few words are in order regarding the use of projections in cc-Golog. They should be understood as a way of assessing whether a program is executable *in principle*. The resulting execution trace of a projection is not intended as input to the execution mechanism of the robot. This is because the time point of a *waitFor*-condition like a low battery level is computed based on a *model* of the world which includes a model of the robot's energy consumption. In reality, of course, the robot should react to the *actual* battery level by periodically reading its voltage meter. In the runtime system of RPL for an actual robot (Thrun *et al.* 1999) this link between *waitFor*-actions and basic sensors which are immediately accessible to the robot has been realized. One possibility to actually execute cc-Golog-programs on a robot would be to combine this idea of executing *waitFor*'s with an incremental interpreter along the lines of (de Giacomo & Levesque 1999). We leave this to future work. Another research issue is uncertainty, which plays a central role in the robotics domain which should be reflected in a plan language as well. Based on foundational work within the situation calculus (Bacchus, Halpern, & Levesque 1995) first preliminary results have been obtained regarding an integration into ConGolog (Grosskreutz & Lakemeyer 2000).

References

Bacchus, F.; Halpern, J.; and Levesque, H. 1995. Reasoning about noisy sensors in the situation calculus. In *IJCAI'95*.

Beetz, M., and Grosskreutz, H. 1998. Causal models of mobile service robot behavior. In *AIPS'98*.

Burns, A., and Wellings, A. 1991. *Real-time systems and their programming languages*. Addison-Wesley.

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

de Giacomo, G.; Lesperance, Y.; and Levesque, H. J. 1997. Reasoning about concurrent execution, prioritized interrupts, and exogeneous actions in the situation calculus. In *IJCAI'97*.

Firby, J. 1987. An investigation into reactive planning in complex domains. In *Proc. of AAAI-87*, 202–206.

Galton, A. 1990. A critical examination of Allen's theory of action and time. *Artificial Intelligence* 42:159–188.

Giacomo, G. D.; Lesperance, Y.; and Levesque, H. J. 1999. Congolog, a concurrent programming language based on the situation calculus: foundations. Technical report, University of Toronto, `http://www.cs.toronto.edu/cogrobo/`.

Grosskreutz, H., and Lakemeyer, G. 2000. Turning high-level plans into robot programs in uncertain domains. In *Submitted*. `http://www-i5.informatik.rwth-aachen-bonn.de/kbsg/publications/`.

Konolige, K. 1997. Colbert: A language for reactive control in sapphira. In *KI'97*, volume 1303 of *LNAI*.

Levesque, H. J.; Reiter, R.; Lesprance, Y.; Lin, F.; and Scherl, R. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

Levesque, H.; Pirri, F.; and Reiter, R. 1998. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science* Vol. 3(1998): nr 018. URL: http://www.ep.liu.se/ea/cis/1998/018/.

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

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

McDermott, D. 1994. An algorithm for probabilistic, totally-ordered temporal projection. Research Report YALEU/DCS/RR-1014, Yale University, `www.cs.yale.edu/AI/Planning/xfrm.html`.

Miller, R. 1996. A case study in reasoning about actions and continuous change. In *ECAI'96*.

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

Reiter, R. 1996. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. KR'96*, 2–13.

Reiter, R. 1998. Sequential, temporal golog. In *Proc. KR'98*.

Sandewall, E. 1989. Combining logic and differential equations for describing real-world systems. In *KR'89*, 412–420.

Shanahan, M. 1990. Representing continuous change in the event calculus. In *ECAI'90*.

Thrun, S.; Bennewitz, M.; Burgard, W.; Dellaert, F.; Fox, D.; Haehnel, D.; Rosenberg, C.; Roy, N.; Schulte, J.; and Schulz, D. 1999. Minerva: A second-generation museum tour-guide robot. In *ICRA'99*.