Scheduling in the Situation Calculus: A Case Study

Ray Reiter* and Zheng Yuhua
Department of Computer Science
University of Toronto
Toronto, Canada M5S 3G4
{reiter,zy}@cs.toronto.edu
http://www.cs.toronto.edu/~cogrobo/

Abstract

We illustrate the utility of the situation calculus for representing complex scheduling tasks by axiomatizing a deadline driven scheduler in the language. The actions arising in such a scheduler are examples of natural actions, as investigated in the concurrent situation calculus by Pinto [10], and later by Reiter [13]. Because the deadline driven scheduler is sequential, we must first suitably modify Reiter's approach to natural actions so it applies to the sequential case. Having done this, we then show how the situation calculus axiomatization of this scheduler yields a very simple simulator in GOLOG, a situation calculus-based logic programming language for dynamic domains.

1 Introduction

The situation calculus (McCarthy [7]) has long been the formalism of choice in artificial intelligence for theoretical investigations of properties of actions, but until very recently, it has not been taken seriously as a specification or implementation language for practical problems in dynamic world modeling. One of the few exceptions to this is the situation calculus-based programming language GOLOG (Levesque et al. [3]), and some of its applications to robotics (Lespérance et al. [2]) and agent programming (Marcu et al. [6]). The purpose of this paper is to explore the usefulness of the situation calculus for a new class of applications in dynamic modeling, namely temporal scheduling problems. Such problems differ from those "traditionally" considered in the literature on the situation calculus; they involve time, and the actions taken by the scheduler are *natural* actions (Pinto [10], Reiter [13]), which are actions that must occur at their predetermined times, provided no earlier actions occur to prevent them. Specifically, in this paper, we axiomatize the deadline driven scheduler of Liu and Layland [5] in the situation calculus. Because this scheduler is sequential, we must

*Fellow of the Canadian Institute for Advanced Research

first suitably modify Reiter's concurrent approach to natural actions so it applies to the sequential case. Having done this, we show how the situation calculus axiomatization of this scheduler yields a very simple simulator in GOLOG.

2 An Informal Introduction to the Situation Calculus[1]

2.1 Intuitive Ontology for the Situation Calculus

The situation calculus (McCarthy [7]) is a first order language (with, as we shall see later, some second order features) specifically designed for representing dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the *initial situation*, namely that situation in which no actions have yet occurred. There is a distinguished binary function symbol do; $do(\alpha, s)$ denotes the successor situation to s resulting from performing the action α. Actions may be parameterized. For example, $put(x, y)$ might stand for the action of putting object x on object y, in which case $do(put(A, B), s)$ denotes that situation resulting from placing A on B when the world is in situation s. Notice that in the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, $do(putdown(A), do(walk(L), do(pickup(A), S_0)))$ is a situation denoting the world history consisting of the sequence of actions [pickup(A), walk(L), putdown(A)]. Notice that the sequence of actions in a history, in the order in which they occur, is obtained from a situation term by reading off the actions from right to left.

Relations whose truth values vary from situation to situation, called *relational fluents*, are denoted by predicate symbols taking a situation term as their last argument. For example, $is\_carrying(robot, p, s)$, meaning that a robot is carrying package p in situation s, is a relational fluent. Functions whose denotations vary from situation to situation are called *functional fluents*. They are denoted by function symbols with an extra argument taking a situation term, as in $pos(robot, s)$, i.e., the robot's position in situation s.

2.2 Axiomatizing Actions and their Effects in the Situation Calculus

Actions have *preconditions* – necessary and sufficient conditions that characterize when the action is physically possible. For example, in a blocks world, we might have:[2]

$$Poss(pickup(x), s) \equiv \\ [(\forall z)\neg holding(z, s)] \wedge nexto(x, s) \wedge \neg heavy(x).$$

World dynamics are specified by *effect axioms*. These describe the effects of a given action on the fluents – the causal laws of the domain. For example, a robot dropping a

[1]This section is borrowed from Levesque et al [3].

[2]In formulas, free variables are considered to be universally prenex quantified. This convention will be followed throughout the paper.

fragile object causes it to be broken:

$$Poss(drop(r,x),s) \wedge fragile(x,s) \supset broken(x, do(drop(r,x),s)). \tag{1}$$

Exploding a bomb next to an object causes it to be broken:

$$Poss(explode(b),s) \wedge nexto(b,x,s) \supset broken(x, do(explode(b),s)). \tag{2}$$

A robot repairing an object causes it to be not broken:

$$Poss(repair(r,x),s) \supset \neg broken(x, do(repair(r,x),s)). \tag{3}$$

2.3 The Frame Problem

As first observed by McCarthy and Hayes [8], axiomatizing a dynamic world requires more than just action precondition and effect axioms. So-called *frame axioms* are also necessary. These specify the action *invariants* of the domain, namely, those fluents which remain unaffected by a given action. For example, a robot dropping things does not affect an object's color:

$$Poss(drop(r,x),s) \wedge color(y,c,s) \supset color(y,c,do(drop(r,x),s)).$$

A frame axiom describing how the fluent *broken* remains unaffected:

$$\begin{aligned} &Poss(drop(r,x),s) \wedge \neg broken(y,s) \wedge [y \neq x \vee \neg fragile(y,s)] \\ &\qquad \supset \neg broken(y, do(drop(r,x),s)). \end{aligned}$$

The problem introduced by the need for such frame axioms is that we can expect a vast number of them. Only relatively few actions will affect a given fluent's value; all other actions leave the fluent invariant. For example, an object's color is not changed by picking things up, opening a door, going for a walk, electing a new prime minister of Canada, etc. This is problematic for the axiomatizer – she must think of all these axioms – and it is problematic for the theorem proving system – it must reason efficiently in the presence of so many frame axioms.

2.3.1 What Counts as a Solution to the Frame Problem?

Suppose the person responsible for axiomatizing an application domain has specified all of the causal laws for the world being axiomatized. More precisely, he has succeeded in writing down *all* the effect axioms, i.e. for each fluent F and each action A which can cause F's truth value to change, axioms of the form

$$Poss(A,s) \wedge R(\vec{x},s) \supset (\neg)F(\vec{x}, do(A,s)).$$

Here, R is a first order formula specifying the contextual conditions under which the action A will have its specified effect on F.

A solution to the frame problem is a systematic procedure for generating, from these effect axioms, all the frame axioms. If possible, we also want a *parsimonious* representation for these frame axioms (because in their simplest form, there are too many of them).

2.4 A Simple Solution to the Frame Problem

By appealing to earlier ideas of Haas [1], Schubert [14] and Pednault [9], Reiter [11] proposes a simple solution to the frame problem, which we illustrate with an example. Suppose that (1), (2), and (3) are all the effect axioms for the fluent *broken*, i.e. they describe all the ways that an action can change the truth value of *broken*. We can rewrite (1) and (2) in the logically equivalent form:

$$\begin{array}{c} Poss(a,s) \wedge [(\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \\ \vee(\exists b)\{a = explode(b) \wedge nexto(b,x,s)\}] \\ \supset broken(x, do(a,s)). \end{array} \tag{4}$$

Similarly, consider the negative effect axiom (3) for *broken*; this can be rewritten as:

$$Poss(a,s) \wedge (\exists r) a = repair(r,x) \supset \neg broken(x, do(a,s)). \tag{5}$$

In general, we can assume that the effect axioms for a fluent F have been written in the forms:

$$Poss(a,s) \ \wedge \ \gamma_F^+(\vec{x},a,s) \ \supset \ F(\vec{x}, do(a,s)), \tag{6}$$

$$Poss(a,s) \ \wedge \ \gamma_F^-(\vec{x},a,s) \ \supset \ \neg F(\vec{x}, do(a,s)). \tag{7}$$

Here $\gamma_F^+(\vec{x},a,s)$ is a formula describing under what conditions doing the action a in situation s leads the fluent F to become true in the successor situation $do(a,s)$; similarly $\gamma_F^-(\vec{x},a,s)$ describes the conditions under which performing a in s results in F becoming false in the next situation. The solution to the frame problem of [11] rests on a *completeness assumption*, which is that the causal axioms (6) and (7) characterize all the conditions under which action a can lead to a fluent $F(\vec{x})$ becoming true (respectively, false) in the successor situation. In other words, axioms (6) and (7) describe all the causal laws affecting the truth values of the fluent F. Therefore, if action a is possible and $F(\vec{x})$'s truth value changes from *false* to *true* as a result of doing a, then $\gamma_F^+(\vec{x},a,s)$ must be *true* and similarly for a change from *true* to *false*. Reiter [11] shows how to derive a *successor state axiom* of the following form from the causal axioms (6) and (7) and the completeness assumption.

Successor State Axiom

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

This single axiom embodies a solution to the frame problem. Notice that this axiom universally quantifies over actions a. In fact, this is one way in which a parsimonious solution to the frame problem is obtained.

Applying this to our example about breaking things, we obtain the following successor state axiom:

$$\begin{array}{l} Poss(a,s) \supset [broken(x, do(a,s)) \equiv \\ \qquad (\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \vee \\ \qquad (\exists b)\{a = explode(b) \wedge nexto(b,x,s)\} \vee \\ \qquad broken(x,s) \wedge \neg(\exists r)a = repair(r,x)]. \end{array}$$

It is important to note that the above solution to the frame problem presupposes that there are no *state constraints*, as for example in the blocks world constraint: $(\forall s).on(x,y,s) \supset$

$\neg on(y, x, s)$. Such constraints sometimes implicitly contain effect axioms (so-called indirect effects), in which case the above completeness assumption will not be true. The assumption that there are no state constraints in the axiomatization of the domain will be made throughout this paper. In [4], the approach discussed in this section is extended to deal with state constraints, by compiling their effects into the successor state axioms.

3 Formal Preliminaries

3.1 The Language of the Situation Calculus

The language $\mathcal{L}$ of the situation calculus is many-sorted, second-order, with equality. We assume the following sorts: *situation* for situations, *action* for actions, and *object* for everything else. We also assume the following domain independent predicates and functions:

- A constant S_0 of sort *situation* denoting the initial situation.
- A binary function do - $do(a, s)$ denotes the situation resulting from performing action a in situation s.
- A binary predicate $Poss$ - $Poss(a, s)$ means that action a is possible (executable) in situation s.
- A binary predicate $<$ over situations. We shall follow convention, and write $<$ in infix form. By $s < s'$ we mean that s' can be obtained from s by a sequence of executable actions. As usual, $s \leq s'$ will be a shorthand for $s < s' \vee s = s'$.

We assume a finite number of relational *fluents*, which are predicate symbols of arity $object^n \times situation$, $n \geq 0$, and are domain dependent. Similarly, we assume a finite number of functional *fluents*, which are function symbols of arity $object^n \times situation \rightarrow object$. Finally, we need a finite number of function symbols of arity $object^n \rightarrow action$, $n \geq 0$, for actions, and a finite number of function symbols of arity $object^n \rightarrow object$, $n \geq 0$.

3.2 Axiomatizing the Situation Calculus

We shall need the following foundational axioms (Lin and Reiter [4], Reiter [12]) for the situation calculus:

$$S_0 \neq do(a, s), \tag{8}$$

$$do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2), \tag{9}$$

$$(\forall P)[P(S_0) \wedge (\forall a, s)(P(s) \supset P(do(a, s))) \supset (\forall s)P(s)], \tag{10}$$

$$\neg s < S_0, \tag{11}$$

$$s < do(a, s') \equiv (Poss(a, s') \wedge s \leq s'). \tag{12}$$

Intuitively, the first two axioms are unique names assumptions. They stipulate that two situations are identical iff they consist of identical sequences of actions. The third axiom is second order induction. It amounts to the domain closure axiom that every situation is obtained from the initial one by repeatedly applying the function do.[3] The last two axioms define $<$ inductively.

Notice the similarity between these axioms and the Peano foundational axioms for number theory. However, unlike Peano arithmetic which has a unique successor function, we have a class of successor functions here represented by the function do.

3.3 Extending the Situation Calculus to Include Time

Time plays no role in the situation calculus as described above in Section 2. Actions occur in sequence, but there is no way to express that a particular action occurs at a given time in such a sequence, nor are any properties of the time line captured in the foundational axioms of Section 3.2. Accordingly, we begin by expanding the situation calculus ontology beyond that of Section 3.1, to include the following additional sorts, function and predicate symbols for the purposes of adding a temporal component to the situation calculus:

1. As before, there is a sort *action* of actions. We now view all actions as instantaneous, and actions will take a parameter (in the last argument position) denoting the time of the action's occurrence. So, $start\_meeting(person, t)$ might be the instantaneous action of *person* starting a meeting at time t. In this paper, we do not need to treat actions with durations. For a description of how to do this using instantaneous actions, see Pinto [10].

2. A sort *time* ranging over the reals.

3. We need a function symbol $time$: $time(a)$ denotes the time of the action a. So, we will have axioms like $time(start\_meeting(person, t)) = t$.

4. We need a function symbol $start$: $start(s)$ denotes the start time of the situation s.

5. Finally, there are predicate symbols $natural$, $legal$ and $lntp$, to be described later.

3.4 Foundational Axioms for the Sequential, Temporal Situation Calculus

We now augment the foundational axioms (8) - (12) for the atemporal situation calculus by three new axioms for accommodating time. The time of an action occurrence is the value of that action's temporal argument. So, for each action function $A(\vec{x}, t)$ of our situation calculus language, we need an axiom:

$$time(A(\vec{x}, t)) = t. \tag{13}$$

The start time of a situation is determined by:

$$start(do(a, s)) = time(a). \tag{14}$$

[3]For a detailed discussion of the use of induction in the situation calculus, see (Reiter [12]).

We require the following global constraint:

$$Poss(a, s) \supset start(s) \leq start(do(a, s)). \tag{15}$$

Axioms (8) - (15) are the foundational axioms for the *sequential temporal situation calculus.*

3.5 Action Precondition Axioms

As in Section 2.2, we shall require action precondition axioms, but we must take into account that the global constraint (15) is a qualification constraint that "compiles" (see Lin and Reiter [4]) into assertions about $Poss$ of the form:

$$Poss(A(\vec{x}, t), s) \supset start(s) \leq t,$$

for each action A. In view of the results in Lin and Reiter [4] on the qualification problem, this means that in the temporal situation calculus, action precondition axioms will all have the form:

$$Poss(A(\vec{x}, t), s) \equiv start(s) \leq t \wedge \Phi(\vec{x}, t, s). \tag{16}$$

Here, $\Phi(\vec{x}, t, s)$ is any first order formula with free variables among $\vec{x}, t$ and s whose only term of sort *situation* is s.

3.6 Axiomatizing an Application Domain in the Situation Calculus

In general, a particular domain of application will be specified by the union of the following sets of axioms:

1. Action precondition axioms, one for each primitive action.

2. Successor state axioms, one for each fluent.

3. Unique names axioms for the primitive actions. These specify that distinct names for actions denote distinct actions. See Section 5.2 below for an example.

4. Axioms describing the initial situation – what is true initially, before any actions have occurred. This is any finite set of sentences which mention only the situation term S_0, or which are situation independent.

5. The domain independent foundational axioms for the situation calculus.

4 Natural Actions[4]

Our focus in this paper is on natural exogenous actions (Pinto [10]), namely those which occur in response to known laws of physics, like a ball bouncing at times determined by Newtonian equations of motion. These laws of physics will be embodied in the action precondition axioms, in the style of Pinto's PhD thesis [10], but in a somewhat more natural form:

$$\begin{aligned} Poss(bounce(t), s) \equiv t \geq start(s) \wedge is\_falling(s) \wedge \\ height(s) + vel(s)[t - start(s)] - 1/2g[t - start(s)]^2 = 0. \end{aligned}$$

Here, $height(s)$ and $vel(s)$ are the height and velocity, respectively, of the ball at the start of situation s.

Notice that the truth of $Poss(bounce(t), s)$ *does not* mean that the bounce action must occur in situation s, or even that the bounce action must eventually occur. It simply means that the bounce is physically possible at time t in situation s; a *catch* action occurring before t should prevent the bounce action.

We introduce a predicate symbol $natural$, with which the axiomatizer can declare suitable actions to be natural, as, for example, $natural(bounce(t))$.

4.1 Natural Actions and Legal Situations

In the space of all possible situations, we want to single out the legal situations, i.e. those which respect the property of natural actions that they must occur at their predicted times, provided no earlier actions (natural or agent initiated) prevent them from occurring. We capture these legal situations with the following definition:

$$\begin{aligned} legal(s) \equiv S_0 \leq s \wedge \\ (\forall a, a', s').natural(a') \wedge Poss(a', s') \wedge do(a, s') \leq s \supset \\ time(a) \leq time(a'). \end{aligned} \tag{17}$$

In other words, the legal situations s are those with the property that every action a in the sequence of actions s is possible ($S_0 \leq s$), and no possible natural action a' can have its occurrence time precede that of a (otherwise, a' should have occurred in the sequence s instead of a, because natural actions must occur at their predicted times when possible).

The following provides a more intuitive, inductive characterization of the legal situations.

Lemma 1 *The foundational axioms imply that the definition (17) is equivalent to the conjunction of the following two sentences:*

$legal(S_0)$.

$$\begin{aligned} legal(do(a, s)) \equiv legal(s) \wedge Poss(a, s) \wedge \\ (\forall a').natural(a') \wedge Poss(a', s) \supset time(a) \leq time(a'). \end{aligned}$$

[4]This section is an adaptation, to the temporal sequential situation calculus, of the treatment of natural actions of (Reiter [13] for the concurrent situation calculus.

Proof:

$\Rightarrow$

Straightforward.

$\Leftarrow$

Use the induction axiom (10), with the definition (17) as induction hypothesis.

$\square$

4.2 Least Natural Time Points

The following definition plays a central role in theorizing about natural actions:

$$\begin{aligned} lntp(s,t) \equiv (\exists a)[natural(a) \wedge Poss(a,s) \wedge time(a) = t] \wedge \\ (\forall a')[natural(a') \wedge Poss(a',s) \supset time(a') \geq t]. \end{aligned} \tag{18}$$

Intuitively, the *least natural time point* is the earliest time during situation s at which a natural action can occur.

Remark 1 *(18) entails the following:*

$$lntp(s,t) \wedge lntp(s,t') \supset t = t'.$$

So, when it exists, the least natural time point is unique. The least natural time point need not exist, for example, when $(\forall a).natural(a) \equiv (\exists x,t)a = B(x,t)$, where x ranges over the nonzero natural numbers, and $Poss(B(x,t),s) \equiv t = start(s) + 1/x$.

The following is an easy consequence of Lemma 1 and the definition (18) of $lntp$.

Lemma 2 *Our situation calculus axioms entail the following:*

$$natural(a) \wedge legal(do(a,s)) \supset lntp(s, time(a)).$$

In the case of a domain closure assumption on natural actions, we can give an explicit formula for $lntp(s,t)$. So, suppose we have the following domain closure axiom:

$$natural(a) \equiv (\exists \vec{x},t)a = A_1(\vec{x},t) \vee \cdots \vee (\exists \vec{z},t)a = A_n(\vec{z},t), \tag{19}$$

together with the associated declarations (13):

$$\begin{aligned} time(A_1(\vec{x},t)) = t, \\ \vdots \\ time(A_n(\vec{z},t)) = t. \end{aligned} \tag{20}$$

Lemma 3 *(18), (19) and (20) entail the following:*

$$\begin{aligned} lntp(s,t) \equiv [(\exists \vec{x})Poss(A_1(\vec{x},t),s) \vee \cdots \vee (\exists \vec{z})Poss(A_n(\vec{z},t),s)] \wedge \\ (\forall \vec{x},t')[Poss(A_1(\vec{x},t'),s) \supset t' \geq t] \wedge \cdots \wedge (\forall \vec{z},t')[Poss(A_n(\vec{z},t'),s) \supset t' \geq t]. \end{aligned}$$

4.3 The Natural World Condition

This is the sentence:

$$(\forall a) natural(a). \qquad\qquad (NWC)$$

The Natural World Condition restricts the domain of discourse to natural actions only.

Theorem 1 *Our situation calculus axioms entail the following:*

$$NWC \supset [legal(do(a,s)) \equiv legal(s) \wedge Poss(a,s) \wedge lntp(s, time(a))].$$

Proof:
$\Rightarrow$
Assume NWC, and, for fixed a and s, that $legal(do(a,s))$. By Lemma 1, we conclude that $legal(s) \wedge Poss(a,s)$, so it remains to prove that $lntp(s, time(a))$. This follows from NWC and Lemma 2.
$\Leftarrow$
Assume NWC, and, for fixed a and s, that $legal(s) \wedge Poss(a,s) \wedge lntp(s, time(a))$. We must prove that $legal(do(a,s))$, which, by Lemma 1, is equivalent to proving $legal(s) \wedge Poss(a,s) \wedge (\forall a').natural(a') \wedge Poss(a',s) \supset time(a) \leq time(a')$. Since $legal(s) \wedge Poss(a,s)$ by hypothesis, we must prove $(\forall a').natural(a') \wedge Poss(a',s) \supset time(a) \leq time(a')$. This follows from the assumption $lntp(s, time(a))$ and the definition of $lntp$.

□

This theorem characterizes the legal situations in the case that all actions are natural; the next legal situations are obtained by extending the current one by any possible action whose time of occurrence is the least natural time point of the current situation. It is this theorem that provides the foundations for our formal account of a deadline driven scheduler, and of its implementation. We do so by axiomatizing the scheduler's actions in the situation calculus, and by treating all these actions as natural. In that way, the possible schedules become the same as the legal situations, for which Theorem 1 provides a characterization. We now turn to this scheduler.

5 A Deadline Driven Scheduler

5.1 Informal Description

A typical example of a scheduling algorithm for real time systems is the deadline driven scheduler of Liu and Layland [5], first proposed in 1973. It supposes that we are given a finite number of tasks, all time-sharing a single processor. Each task p_i requires a certain amount of processor time C_i for its completion, and has a deadline $T_i > C_i$ for its completion. This completion deadline is understood as follows: Assuming that everything starts out at time 0, then task p_i must occupy, in total, C_i time units of the processor, and must terminate this execution before T_i. At time T_i, it again is eligible to occupy a total of C_i time units on the processor, and it must terminate this execution before time $2 * T_i$. Etc. So each task p_i must periodically (with period T_i) consume a total of C_i time units on the processor, and must do so before time $n_i * T_i$, where n_i is the current period for p_i. While a task p is running on the processor (and is therefore consuming a portion of its running time C), it may become preempted by a highest priority task p'. When this happens, p is suspended from its execution, and p' is given a chance to run on the processor. If, at some later time (at which perhaps a third task p'' is occupying the processor), p's priority becomes highest again, then it will resume the execution from which it was preempted by p'. These priorities are dynamically determined as follows: A task, whether it is running on the processor or not, has a highest priority at time t if it still needs running time on the processor for its current period, and its completion deadline (which will be of the form $n * T$ for some positive integer n) is earlier than that of all the other tasks still needing time on the processor at time t.

To better understand the behavior of the scheduler, consider the following diagram:

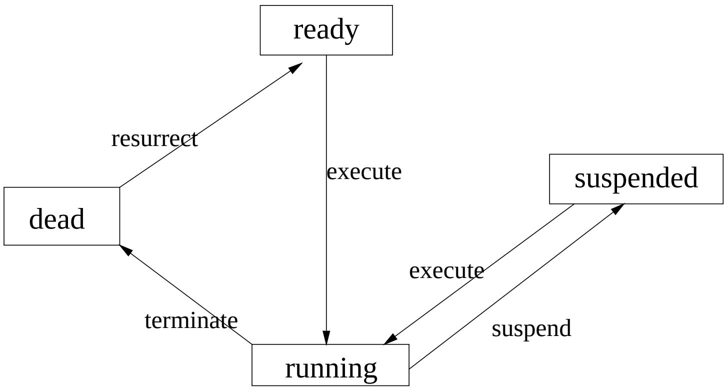

Figure 1: Scheduling Activities

Assume there are four states for each task: *dead*, *ready*, *suspended* and *running*. Initially all tasks are in their *ready* states. Among all the tasks in the *ready* or *suspended* state, the scheduler chooses the most urgent (according to the dynamic priority assignment described above) to *execute*, changing that task's state to *running*. During the execution of task p, if another task p' becomes most urgent, then p' preempts p. In other words, the scheduler *suspend*s the execution of p, and lets p' *execute* on the processor. The state of p changes from *running* to *suspended*, and that of p' from *ready* or *suspended* to *running*. When a task finishes its execution for its current period (by consuming its required processor time C for this period), it is put into a *dead* state by the action *terminate*. To put such a *dead* task into its *ready* state when its next period arrives, the scheduler performs the action *resurrect*.

The requirement for this scheduling problem is that each task should obtain enough processor time to finish its execution before its deadline. Obviously, there are situations in which the scheduler won't be able to meet this scheduling requirement for all its tasks, for example, when two tasks have the same deadline of 2, and request the same execution time of 2.

[5] established a necessary and sufficient condition for the deadline driven scheduler to meet its requirement when it serves multiple periodic tasks, i.e. when the task is to be

executed repeatedly, and the requests for execution are made at regular periodic intervals, as described above.

Example 1 Suppose there are two periodic tasks p_1 and p_2, with request periods $T_1 = 3$ time units and $T_2 = 12$, and running times $C_1 = 2$ and $C_2 = 4$.

Figure 2 is the execution diagram for these two periodic tasks according to this deadline driven strategy.

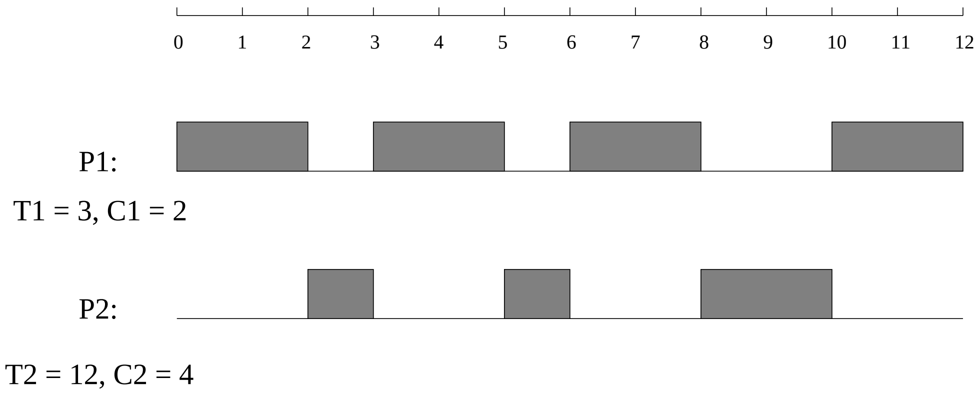

Figure 2: Execution Diagram of Example 1

1. Initially, p_1 and p_2 are both in a *ready* state.

2. At time 0, p_1 is selected to run since p_1 is currently more urgent than p_2 (p_1's current deadline is 3, and p_2's is 12). This corresponds to the following scheduler action at time 0: $execute(p_1, 0)$.

3. At time 2, p_1 is terminated, and then p_2 was chosen to run. This corresponds to the sequence of scheduler actions: $terminate(p_1, 2), execute(p_2, 2)$.

4. At time 3, p_1 gets ready again, and since p_2 hasn't yet finished its execution, and p_1 is still currently more urgent than p_2 (p_1's current deadline is 6, and p_2's is 12), p_1 preempts p_2. This corresponds to the following actions:
$resurrect(p_1, 3), suspend(p_2, 3), execute(p_1, 3)$.

5. At time 5, p_1 terminates again, so the execution of p_2 is resumed:
$terminate(p_1, 5), execute(p_2, 5)$.

6. At time 6, as at time 3, p_1 gets ready again, and preempts p_2:
$resurrect(p_1, 6), suspend(p_2, 6), execute(p_1, 6)$.

7. At time 8, similar to time 5, p_1 terminates and the execution of p_2 resumes again: $terminate(p_1, 8), execute(p_2, 8)$.

8. At time 9, p_1 gets ready again, but this time it is not currently more urgent than p_2 (p_1's current deadline is 12, the same as that of p_2). So p_2 won't be preempted; it continues running untill it is terminated. The only action that takes place is: $resurrect(p_1, 9)$.

9. At time 10, p_2 terminates, and p_1 is selected to run: $terminate(p_2, 10), execute(p_1, 10)$.

10. Finally, at time 12, p_2 gets ready again, but p_1 will continue running: $resurrect(p_2, 12)$.

11. The entire schedule will now repeat periodically.

Notice that p_2 does not continuously occupy the processor for the amount of run time it requests; it has been preempted twice by p_1 until it at last achieves 4 time units of run time before its first request period.

In this example, there is no processor idle time. In other words, the processor is always busy with executing tasks. But this is not always true; our second example does allow processor idle time.

Example 2 We have three periodic tasks p_1, p_2 and p_3, with request periods $T_1 = 4$, $T_2 = 6$, $T_3 = 12$ and running times of $C_1 = 2$, $C_2 = 1$, $C_3 = 2$.

Figure 3 shows the execution diagram for this example.

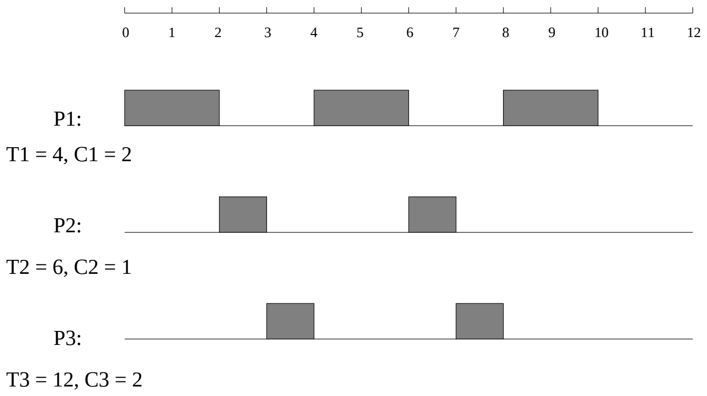

Figure 3: Execution Diagram of Example 2

Notice that in this diagram there will be processor idle time from time 10 to 12.

The following theorem provides necessary and sufficient conditions for the scheduling algorithm to meet the requirement that every task completes its running time before the end of its current period:

Theorem (Liu/Layland [5]) *For a given set of m tasks, the deadline driven scheduling algorithm is feasible if and only if ,*

$$(C_1/T_1) + (C_2/T_2) + ... + (C_m/T_m) \leq 1$$

where C_i is the run time of task i, and T_i is its request period.

T_i, C_i are reasonably assumed to be integers (say multiples of machine cycles).

5.2 Axiomatization in the Situation Calculus

Actions

- $resurrect(p,t)$: At time t, resurrect *process* p from its *dead* state to a *ready* state.
- $suspend(p,t)$: At time t, suspend *process* p from its *running* state on the processor to a *suspended* state.
- $execute(p,t)$: At time t, execute *process* p on the processor, putting it into its *running* state.
- $terminate(p,t)$: At time t, terminate the execution of *process* p on the processor, putting it into its *dead* state.

Functions and Fluents

- $T(p)$, a positive integer, denotes the period of *process* p.
- $C(p)$, a positive integer, denotes the running time of *process* p.
- $TotExTime(p,s)$ denotes the total amount of time, up to the beginning of situation s, that process p has been in a running state.
- $dead(p,s)$: In situation s, *process* p is in a *dead* state, meaning that it is not waiting to be served by the processor.
- $ready(p,s)$: In situation s, *process* p is ready and is waiting to be served by the processor.
- $suspended(p,s)$: In situation s, *process* p is suspended, and is waiting for the processor to resume its execution of p.
- $running(p,s)$: In situation s, *process* p is running on the processor.

In addition to the foundational axioms for the temporal, sequential situation calculus (Sections 3.2 and 3.4), we have the following scheduler specific axioms:

Miscellaneous Axioms

- $(\forall p)T(p) > C(p) > 0$
- There are finitely many (n) processes, denoted by $P_1, \ldots, P_n$, and these are all the processes: $(\forall p).p = P_1 \vee ... \vee p = P_n$.

Unique Names Axioms for Actions

- $resurrect(p,t) \neq suspend(p',t')$
- $resurrect(p,t) \neq execute(p',t')$
- $resurrect(p,t) \neq terminate(p',t')$
- $suspend(p,t) \neq execute(p',t')$

- $suspend(p,t) \neq terminate(p',t')$
- $execute(p,t) \neq terminate(p',t')$
- $resurrect(p,t) = resurrect(p',t') \supset p = p' \wedge t = t'$
- $suspend(p,t) = suspend(p',t') \supset p = p' \wedge t = t'$
- $execute(p,t) = execute(p',t') \supset p = p' \wedge t = t'$
- $terminate(p,t) = terminate(p',t') \supset p = p' \wedge t = t'$

The Natural World Condition

$$(\forall a) natural(a)$$

Domain Closure for Actions

$$(\forall a)(\exists p,t)[a = resurrect(p,t) \vee a = suspend(p,t) \vee a = execute(p,t) \vee a = terminate(p,t)]$$

Abbreviation 1 $NextResurrectTime(p,s) \stackrel{def}{=} (\lfloor start(s)/T(p) \rfloor + 1) * T(p)$

Abbreviation 2

$$\begin{aligned} MoreUrgent(p,p',s) \stackrel{def}{\equiv} & [ready(p,s) \vee suspended(p,s) \vee running(p,s)] \wedge \\ & NextResurrectTime(p,s) < NextResurrectTime(p',s). \end{aligned}$$

Abbreviation 3 $RequestedExTime(p,s) \stackrel{def}{=} (\lfloor start(s)/T(p) \rfloor + 1) * C(p)$

Abbreviation 4 $RemainingExTime(p,s) \stackrel{def}{=} RequestedExTime(p,s) - TotExTime(p,s)$

Abbreviation 5 $CompletionTime(p,s) \stackrel{def}{=} start(s) + RemainingExTime(p,s)$

Action Precondition Axioms

$$\begin{aligned} Poss(resurrect(p,t),s) \equiv & \; dead(p,s) \wedge \\ & [t = NextResurrectTime(p,s) \vee t = start(s) \wedge \lfloor \tfrac{t}{T(p)} \rfloor = \tfrac{t}{T(p)}] \end{aligned}$$

$$Poss(suspend(p,t),s) \equiv running(p,s) \wedge (\exists p') MoreUrgent(p',p,s) \wedge t = start(s)$$

$$\begin{aligned} Poss(execute(p,t),s) \equiv & \; (ready(p,s) \vee suspended(p,s)) \\ & \wedge \neg(\exists p') MoreUrgent(p',p,s) \wedge \neg(\exists p') running(p',s) \wedge t = start(s) \end{aligned}$$

$$Poss(terminate(p,t),s) \equiv running(p,s) \wedge t = CompletionTime(p,s)$$

Successor State Axioms

$$\begin{aligned} Poss(a,s) \supset [& TotExTime(p,do(a,s)) = \\ & \textbf{if } running(p,s) \textbf{ then } TotExTime(p,s) + time(a) - start(s) \\ & \textbf{else } TotExTime(p,s)] \end{aligned}$$

$$Poss(a,s) \supset [ready(p, do(a,s)) \equiv (\exists t) a = resurrect(p,t) \vee ready(p,s) \wedge \neg(\exists t) a = execute(p,t)]$$

$$Poss(a,s) \supset [suspended(p, do(a,s)) \equiv (\exists t) a = suspend(p,t) \vee suspended(p,s) \wedge \neg(\exists t) a = execute(p,t)]$$

$$Poss(a,s) \supset [running(p, do(a,s)) \equiv (\exists t) a = execute(p,t) \vee running(p,s) \wedge \neg(\exists t)[a = terminate(p,t) \vee a = suspend(p,t)]]$$

$$Poss(a,s) \supset [dead(p, do(a,s)) \equiv (\exists t) a = terminate(p,t) \vee dead(p,s) \wedge \neg(\exists t) a = resurrect(p,t)]$$

Initial Situation

$$ready(p, S_0), \quad \neg dead(p, S_0), \quad \neg suspended(p, S_0),$$
$$\neg running(p, S_0), \quad TotExTime(p, S_0) = 0.$$

Because we have domain closure on actions, Lemma 3 gives us the following characterization of the least natural time points:

$$lntp(s,t) \equiv (\exists p) \begin{pmatrix} Poss(resurrect(p,t),s) \vee Poss(suspend(p,t),s) \\ \vee Poss(execute(p,t),s) \vee Poss(terminate(p,t),s) \end{pmatrix}$$
$$\wedge (\forall p, t') \begin{pmatrix} (Poss(resurrect(p,t'),s) \supset t' \geq t) \wedge \\ (Poss(suspend(p,t'),s) \supset t' \geq t) \wedge \\ (Poss(execute(p,t'),s) \supset t' \geq t) \wedge \\ (Poss(terminate(p,t'),s) \supset t' \geq t) \end{pmatrix}$$

5.3 What is a Schedule?

Intuitively, a schedule of length n is a sequence of n ground actions chosen from the scheduler's action repertoire $resurrect(p,t), suspend(p,t), \ldots$, with the property that each action of the sequence is among the earliest actions which can occur at their predetermined times, according to their precondition axioms, i.e. each action in the schedule occurs at the least natural time point of its situation. In other words, the finite schedules are precisely the legal situations, as defined in Section 4.1.[5]

6 The Scheduler: An Implementation in GOLOG

6.1 GOLOG

GOLOG (Levesque et al [3]) is a situation calculus-based logic programming language for defining complex actions using a repertoire of user specified primitive actions. GOLOG provides the usual kinds of programming language control structures (sequence, iteration,

[5]Strictly speaking, the question of what counts as a schedule is more subtle than we have let on here. It is by no means obvious that a given legal situation, which is a finite sequence of actions, can always be extended to a longer schedule. This property is necessary, because the deadline driven scheduler computes forever. In fact, this can be done, and we have a proof of this, but the proof is not entirely trivial.

conditionals, recursive procedures) as well as three flavours of nondeterministic choice. Here we briefly describe the GOLOG control structures. For a precise description of their semantics, see [3].

1. *Sequence:* $\alpha\,;\,\beta$. Do action α, followed by action β.
2. *Test actions:* $p?$ Test the truth value of expression p in the current situation.
3. *While loops:* **while** p **do** α **endWhile.**
4. *Conditionals:* **if** p **then** α **else** β.
5. *Nondeterministic choice of actions:* $\alpha \mid \beta$. Do α or do β.
6. *Nondeterministic choice of arguments:* $(\pi\ x)\alpha$. Nondeterministically pick a value for x, and for that value of x, do the action α.
7. *Nondeterministic repetition:* α^*. Do α a nondeterministic number of times.
8. *Procedures, including recursion.*

The semantics of a GOLOG program is defined (see [3]) by macro-expansion, using a ternary relation *Do*. $Do(program, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive meaning is that s' is one of the situations reached by evaluating the GOLOG *program*, beginning in situation s. This means that to execute *program*, one must *prove*, using the situation calculus axiomatization of some background domain, the situation calculus formula $(\exists s)Do(program, S_0, s)$. Any binding for s obtained by a constructive proof of this sentence is an execution trace, in terms of the primitive actions, of the *program*.

6.2 GOLOG Description of the Scheduler

The primitive actions of a GOLOG program must be axiomatized in the situation calculus using successor state and action precondition axioms, just as we have done for the deadline driven scheduler. Accordingly, it is natural to implement the scheduler in GOLOG; the result is a simulator for the scheduler, as we now describe. Recall that we have identified the finite schedules with the legal situations. The following GOLOG program computes exactly the legal situations of length n:

$$
\begin{array}{l}
\textbf{proc}\ schedule(n) \\
\quad n = 0? \mid (\pi\ t)[lntp(t)?\,;\,(\pi\ p)[resurrect(p,t) \mid terminate(p,t) \mid execute(p,t) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad suspend(p,t)]]\,;\,schedule(n-1) \\
\textbf{endProc}
\end{array}
$$

schedule is a recursive procedure which simulates the first n actions of the deadline driven scheduler. It would be "called", for some given n, say $n = 100$, by constructing a proof of the sentence $(\exists s)Do(schedule(100), S_0, s)$, using as premises the axioms for the situation calculus, and the axioms of Section 5.2 above. Any binding for s obtained from this proof will be a legal situation, and hence a schedule.

In defining the procedure *schedule*, we are appealing to Theorem 1, which characterizes the legal situations when all actions are natural (as they are for the scheduler). Specifically, when $n > 0$, *schedule* first determines the least natural time point of the current situation: $(\pi\ t)[lntp(t)?\,; \cdots$. It then nondeterministically determines a primitive action whose action preconditions allow it to execute at this least natural time point:
$(\pi\ p)[resurrect(p,t) \mid terminate(p,t) \mid execute(p,t) \mid suspend(p,t)]$.
Then it calls itself recursively to determine the next primitive action to perform, etc.

Notice that test conditions (e.g. $lntp(t)?$ in the above) in GOLOG programs suppress their situation arguments. These situation arguments are restored during evaluation of the program; the restored value of this argument is the current situation, i.e. the situation that the simulated system would be in, at the point at which the test condition is being evaluated.[6] This is the standard account of test conditions in conventional programming languages; test conditions are always evaluated relative to the machine state at the time of the evaluation.

6.3 A GOLOG Interpreter

We appeal to a Quintus Prolog implementation of a GOLOG interpreter, which we present in full (Figure 4) in the event that the reader wishes to experiment with it[7]. The `do` predicate here takes 3 arguments: a GOLOG action expression, and terms standing for the initial and final situations. Normally, a query will be of the form `do(`e`,s0,S)`, so that an answer will be a binding for the final situation `S`. In this implementation, a legal GOLOG action expression e is one of the following:

- `[`e_1`, ... ,`e_n`]`, sequence.
- `?(`p`)`, where p is a condition (see below).
- e_1 `#` e_2, nondeterministic choice of e_1 or e_2.
- `if(`p`,`e_1`,`e_2`)`, conditional.
- `star(`e`)`, nondeterministic repetition.
- `while(`p`,`e`)`, iteration.
- `pi(`v`,`e`)`, nondeterministic assignment, where v is an atom (standing for a GOLOG variable) and e is a GOLOG action expression that uses v.
- a, where a is the name of a user-declared primitive action or defined procedure (see below).

A condition p in the above is either a fluent or an expression of the form `and(`p_1`,`p_2`)`, `or(`p_1`,`p_2`)`, `neg(`p`)`, or `some(`v`,`p`)`, where v is an atom and p is a condition using v. In evaluating these conditions, the interpreter uses negation as failure to handle `neg`, and consults the user-supplied `holds` predicate to determine which fluents are true.

[6] See [3] for the formal semantics of test conditions.

[7] This interpreter, and indeed much of the material for this section, is taken from Levesque et al [3]. The reader who wishes a better understanding of GOLOG is advised to consult this reference.

Figure 4: A Golog interpreter in Prolog

```
:- op(950, xfy, [#]). /* Nondeterministic action choice.*/

do([],S,S).          /* This clause and the next are for sequences */
do([E|L],S,S1) :- do(E,S,S2), do(L,S2,S1).
do(?(P),S,S) :- holds(P,S).
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
do(if(P,E1,E2),S,S1) :- do([?(P),E1] # [?(neg(P)),E2],S,S1).
do(star(E),S,S1) :- do([] # [E,star(E)],S,S1).
do(while(P,E),S,S1):- do([star([?(P),E]),?(neg(P))],S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).

/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */
sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- \+ var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- \+ T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

holds(and(P1,P2),S) :- holds(P1,S), holds(P2,S).
holds(or(P1,P2),S) :- holds(P1,S); holds(P2,S).
holds(neg(P),S) :- \+ holds(P,S).   /* Negation by failure */
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).
```

In this implementation, a GOLOG application (like the scheduler, below) is expected to have the following parts:

1. a collection of clauses of the form `primitive_action(`*act*`)`, declaring each primitive action.

2. a collection of clauses of the form `proc(`*name*`,`*body*`)` declaring each defined procedure (which can be recursive). The *body* here can be any legal GOLOG action expression.

3. a collection of clauses which together define the predicate `poss(`*act*`,`*situation*`)` over every primitive action and situation.

4. a collection of clauses which together define the predicate `holds(`*fluent*`,`*situation*`)` over every fluent and situation. Normally, this is done in two parts:

(a) a collection of clauses which are situation independent, or which mention only the initial situation `s0`. These serve to describe the initial world situation.

(b) a collection of clauses defining `holds(`*fluent*`,do(`*act*`,`*situation*`))` for every fluent. For each fluent, this amounts to writing its successor state axiom.

6.4 Simulation of the Deadline Driven Scheduler in GOLOG

The following clauses provide the GOLOG description of the previously specified deadline driven scheduler:.

```
--------------------Beginning of the scheduler-----------------------------
/* Primitive control actions */

primitive_action(resurrect(p(N),T)).
primitive_action(suspend(p(N),T)).
primitive_action(execute(p(N),T)).
primitive_action(terminate(p(N),T)).

/* Preconditions for Primitive Actions */

poss(resurrect(p(N),T),S) :-holds(dead(p(N)),S),
    (nextresurrecttime(p(N),S,T);start(S,T),t(p(N),Tn), T mod Tn =:= 0).

poss(suspend(p(N),T),S) :- holds(running(p(N)),S),
    holds(moreurgent(p(J),p(N)),S), start(S,T).

poss(execute(p(N),T),S) :- holds(or(ready(p(N)),suspended(p(N))),S),
     \+ holds(some(j1,moreurgent(p(j1),p(N))),S),
     \+ holds(some(j1,running(p(j1))),S), start(S,T).

poss(terminate(p(N),T),S) :- holds(running(p(N)),S),completiontime(p(N),S,T).

/* Successor state axioms for primitive actions. */

holds(totextime(Time,p(N)), do(E,S)) :- (holds(running(p(N)),S),
      holds(totextime(T1,p(N)),S),time(E,T2),start(S,T3),
      Time is T1+T2-T3);holds(totextime(Time,p(N)),S).

holds(ready(p(N)), do(E,S)) :- E=resurrect(p(N),_);
      holds(ready(p(N)),S), \+ E=execute(p(N),_).

holds(suspended(p(N)), do(E,S)) :- E=suspend(p(N),_);
      holds(suspended(p(N)),S), \+ E=execute(p(N),_).
```

```
holds(running(p(N)), do(E,S)) :- E=execute(p(N),_);
      holds(running(p(N)),S), \+ E=terminate(p(N),_), \+ E=suspend(p(N),_).

holds(dead(p(N)), do(E,S)) :- E=terminate(p(N),_);
      holds(dead(p(N)),S), \+ E=resurrect(p(N),_).

/* Definitions */

holds(moreurgent(p(N1),p(N2)),S) :-
     holds(or(ready(p(N1)),or(running(p(N1)),suspended(p(N1)))),S),
     nextresurrecttime(p(N1),S,T1),nextresurrecttime(p(N2),S,T2),T1<T2.

holds(lntp(T),S):-
      (poss(resurrect(p(_),T),S);poss(suspend(p(_),T),S);
       poss(execute(p(_),T),S);poss(terminate(p(_),T),S)),
      \+ (poss(resurrect(p(_),T1),S), T1 < T),
      \+ (poss(suspend(p(_),T1),S), T1 < T),
      \+ (poss(execute(p(_),T1),S),T1 < T),
      \+ (poss(terminate(p(_),T1),S),T1 < T).

completiontime(p(N),S,T):- remainingextime(p(N),S,T1),start(S,T2), T is T1+T2.

remainingextime(p(N),S,T):-
               holds(totextime(Time,p(N)),S), t(p(N),Ti), c(p(N),Ci),
               start(S,T1), T is (integer(T1/Ti)+1)*Ci-Time.

nextresurrecttime(p(N),S,T):- start(S,T1), t(p(N),Tn),
                 T is (integer(T1/Tn)+1)*Tn.

holds(X=Y,S) :- X=:=Y.

start(s0,0).
start(do(E,S),T):-time(E,T).

time(resurrect(_,T),T).
time(execute(_,T),T).
time(suspend(_,T),T).
time(terminate(_,T),T).

/*  The GOLOG scheduling procedure. schedule(N) simulates the first N
    primitive actions of the scheduler.  */

proc(schedule(N),[?(N=0)#[pi(t,[?(lntp(t)),pi(i,
     resurrect(p(i),t)#terminate(p(i),t)#execute(p(i),t)#suspend(p(i),t))]),
```

```
     schedule(N-1)]]).

----------------------End of the scheduler---------------------------------
```

Now we define the periodic tasks of the first example in Section 5.1 and the initial situation.

```
-------------------------Begin example 1----------------------------------
/* Definition of the request period of each task */

t(p(1),3).
t(p(2),12).

/* Definition of the run time of each task.  */

c(p(1),2).
c(p(2),4).

/* Initial situation  */

holds(ready(p(1)),s0).
holds(ready(p(2)),s0).
holds(totextime(0,p(1)),s0).
holds(totextime(0,p(2)),s0).

--------------------------End of example 1---------------------------------
```

Next, we provide a query to the interpreter for example 1, and the result of its execution.

```
| ?- do(schedule(17),s0,S).

S = do(resurrect(p(2),12),do(execute(p(1),10),do(terminate(p(2),10),
    do(resurrect(p(1),9),do(execute(p(2),8),do(terminate(p(1),8),
    do(execute(p(1),6),do(suspend(p(2),6),do(resurrect(p(1),6),
    do(execute(p(2),5),do(terminate(p(1),5),do(execute(p(1),3),
    do(suspend(p(2),3),do(resurrect(p(1),3),do(execute(p(2),2),
    do(terminate(p(1),2),do(execute(p(1),0),s0))))))))))))))))
```

This output corresponds to the schedule of Example 1 of Section 5.1.

The periodic tasks of the second example in Section 5.1 and the initial situation are as follows:

```
--------------------------Begin example 2---------------------------------
/* Definition of the request period of each task */
```

```
t(p(1),4).
t(p(2),6).
t(p(3),12).

/* Definition of the run time of each task.  */

c(p(1),2).
c(p(2),1).
c(p(3),2).

/* Initial situation  */

holds(ready(p(1)),s0).
holds(ready(p(2)),s0).
holds(ready(p(3)),s0).
holds(totextime(0,p(1)),s0).
holds(totextime(0,p(2)),s0).
holds(totextime(0,p(3)),s0).

--------------------------End of example 2--- -----------------------------
```

We also show a query for this example and its execution result, which corresponds to the execution diagram of Section 5.1.

```
| ?- do(schedule(20),s0,S).

S = do(resurrect(p(2),12),do(resurrect(p(3),12),do(resurrect(p(1),12),
    do(terminate(p(1),10),do(execute(p(1),8),do(terminate(p(3),8),
    do(resurrect(p(1),8),do(execute(p(3),7),do(terminate(p(2),7),
    do(execute(p(2),6),do(terminate(p(1),6),do(resurrect(p(2),6),
    do(execute(p(1),4),do(suspend(p(3),4),do(resurrect(p(1),4),
    do(execute(p(3),3),do(terminate(p(2),3),do(execute(p(2),2),
    do(terminate(p(1),2),do(execute(p(1),0),s0))))))))))))))))))))
```

7 Discussion

We have found that the situation calculus, augmented with a temporal component, is well suited for representing the above complex deadline driven scheduling task. An added benefit of formalizing such tasks in the situation calculus is that one can prove properties of them. While we do not include it in this paper – it is a very long and complicated proof – we have constructed a proof of the Liu/Layland Theorem of Section 5.1 from the axioms of Section 5.2. This is not the first such axiomatic proof of this result; using the duration calculus – a propositional temporal logic – Zheng and Zhou [15] have also provided such a proof.

The advantage of the situation calculus in such real time settings, apart from its greater generality – it is first order – is that it provides the foundations for GOLOG, and hence the task specification leads simply, and naturally to a GOLOG implementation. This is a major advantage for the purposes of the Cognitive Robotics Project at the University of Toronto, whose long term goal is to use GOLOG for high level control of dynamical systems, including robotic behaviors [2]. Since real time control is an essential component of such modeling tasks, the incorporation of schedulers into the systems' GOLOG behavioral descriptions is essential, and we are encouraged in this by our experience with the above deadline driven scheduler.

Acknowledgements:

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada, the Institute for Robotics and Intelligent Systems of the Government of Canada, and the Information Technology Research Centre of the Government of Ontario.

References

[1] A. R. Haas. The case for domain-specific frame axioms. In F. M. Brown, editor, *The frame problem in artificial intelligence. Proceedings of the 1987 workshop*, pages 343–348, Los Altos, California, 1987. Morgan Kaufmann Publishers, Inc.

[2] Yves Lespérance, Hector Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter, and Richard Scherl. A logical approach to high-level robot programming – a progress report. In *Control of the Physical World by Intelligent Systems, Working Notes of the 1994 AAAI Fall Symposium*, November, 1994. New Orleans, LA.

[3] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG : A logic programming language for dynamic domains. *Journal of Logic Programming, Special Issue on Actions*, 1996. To appear.

[4] F. Lin and R. Reiter. State constraints revisited. *J. of Logic and Computation, special issue on actions and processes*, 4:655–678, 1994.

[5] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[6] D. Marcu, Y. Lespérance, H. Levesque, F. Lin, R. Reiter, and R. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J.P. Muller, and M. Tambe, editors, *Intelligent Agents Volume II – Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, pages 331–346. Springer-Verlag, Lecture Notes in Artificial Intelligence, 1996. To appear.

[7] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[8] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[9] E.P.D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R.J. Brachman, H. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332. Morgan Kaufmann Publishers, Inc., 1989.

[10] J.A. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, University of Toronto, Department of Computer Science, 1994.

[11] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[12] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[13] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Proc. Common Sense 96: 3rd Symposium on Logical Formalizations of Commonsense Reasoning*, Stanford, CA, Jan. 6-8, 1996.

[14] L.K. Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Loui, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, 1990.

[15] Zheng Yuhua and Zhou Chaochen. A formal proof of the deadline driven scheduler. In H.Langmaak, W.-P. de Roever, and J.Vytopil, editors, *Formal Techniques in Real-Time and Fault Tolerant Systems, Lecture Notes in Computer Science 863*, pages 756–775. Springer-Verlag, 1994.