7 Time, Concurrency, and Processes

So far, we have ignored three fundamental properties of real actions—they occur in time, they normally have durations, and frequently they occur together, i.e., concurrently. The situation calculus, as presented to this point, provides no way of representing these features of actions. Indeed, insofar as actions are currently represented in the situation calculus, they occur sequentially, and atemporally. This chapter is devoted to expanding the situation calculus ontology and axiomatization to accommodate a more realistic picture of action occurrences.

7.1 Concurrency and Instantaneous Actions

Modeling the possibility of concurrent action execution leads to many difficult formal and conceptual problems. For example, what can one mean by a concurrent action like $\{walk(A, B), chewGum\}$? Intuitively, both actions have durations. By this concurrent action, is it meant that both actions have the same duration? That the time segment occupied by one is entirely contained in that occupied by the other? That their time segments merely overlap? What if there are three actions and the first overlaps the second, the second overlaps the third, but the first and third do not overlap; do they all occur concurrently? A representational device in the situation calculus for overcoming these problems is to conceive of such actions as *processes*, represented by relational fluents, and to introduce durationless (instantaneous) actions that initiate and terminate these processes. For example, instead of the monolithic action representation $walk(x, y)$, we might have instantaneous actions $startWalk(x, y)$ and $endWalk(x, y)$, and the process of walking from x to y, represented by the relational fluent $walking(x, y, s)$. $startWalk(x, y)$ causes the fluent $walking$ to become true, $endWalk(x, y)$ causes it to become false. Similarly, we might represent the $chewGum$ action by the pair of instantaneous actions $startChewGum$ and $endChewGum$, and the relational fluent $chewingGum(s)$. It is straightforward to represent these fluents and instantaneous actions in the situation calculus. For example, here are the action precondition and successor state axioms for the walking action:

$$Poss(startWalk(x, y), s) \equiv \neg(\exists u, v)walking(u, v, s) \wedge location(s) = x,$$

$$Poss(endWalk(x, y), s) \equiv walking(x, y, s),$$

$$walking(x, y, do(a, s)) \equiv a = startWalk(x, y) \vee walking(x, y, s) \wedge a \neq endWalk(x, y),$$

$$location(do(a, s)) = y \equiv (\exists x)a = endWalk(x, y) \vee location(s) = y \wedge \neg(\exists x, y')a = endWalk(x, y').$$

With this device of instantaneous $start$ and end actions in hand, arbitrarily complex concurrency can be represented. For example,

$$\{startWalk(A, B), startChewGum\}, \{endChewGum, startSing\},$$
$$\{endWalk(A, B)\}$$

is the sequence of actions beginning with simultaneously starting to walk and starting to chew, followed by simultaneously ending to chew and starting to sing, followed by ending the walk (at which time the singing process is still going on).

7.2 Concurrency via Interleaving

Before focusing on true concurrency, in which two or more actions start and end at exactly the same times, we consider what other forms of concurrency can be realized in the situation calculus as developed thus far. As we shall see, a surprisingly rich theory of *interleaved concurrency* can be expressed within the sequential situation calculus, provided we appeal to instantaneous $start$ and end actions, as in the previous section. In computer science, concurrency is most often modeled via *interleaving*. Conceptually, two actions are interleaved when one of them is the next action to occur after the other, and usually an interleaving account of such a concurrent occurrence is considered appropriate if the outcome is independent of the order in which the actions are interleaved.

We can provide interleaved concurrent representations for walking and chewing gum, such as

$$do([startWalk(A, B), startChewGum, endChewGum, endWalk(A, B)], S_0),$$

in which the gum-chewing process is initiated after the walking process, and terminated before the end of the walking process. Or, the gum-chewing can start before the walking, and terminate before the walking ends:

$$do([startChewGum, startWalk(A, B), endChewGum, endWalk(A, B)], S_0).$$

In other words, we can represent any overlapping occurrences of walking and chewing gum, *except for exact co-occurrences of any of the instantaneous initiating and terminating actions*. For many applications, this is sufficient. The great advantage is that interleaved concurrency can be represented in the sequential situation calculus, and no new extensions of the theory are necessary.

It is important to have a clear picture of exactly what, conceptually, is being modeled by interleaved concurrency with instantaneous actions like $startWalk(x, y)$ and $endChewGum$. Since as yet, we have no explicit representation for time, the situation calculus

axioms capture a purely qualitative notion of time. Sequential action occurrence is the only temporal concept captured by the axioms; an action occurs *before* or *after* another. It may occur one millisecond or one year before its successor; the axioms are neutral on this question. So a situation $do([A_1, A_2, \ldots, A_n], S_0)$ must be understood as a world history in which, after a nondeterministic period of time, A_1 occurs, then, after a nondeterministic period of time, A_2 occurs, etc. If, for example, this action sequence was the result of a Golog robot program execution, then the robot's action execution system would make the decision about the exact times at which these actions would be performed sequentially in the physical world, but the axioms, being neutral on action occurrence times, contribute nothing to this decision. Later, we shall show how to incorporate time into the situation calculus, after which one can axiomatically specify the times at which actions are to occur.

7.2.1 Examples of Interleaved Concurrency

Imagine a door with a spring handle. The door can be unlocked by turning the handle, but the agent must hold the handle down, for if not, the spring loaded mechanism returns the handle to its locked position. To open the door, an agent must turn the handle, and hold it down while she pushes on the door. The concurrent handle turning and door pushing causes the door to open. Neither action by itself will open the door. This is easy to do in the situation calculus if we view the action of turning and holding the handle down, which intuitively has a duration, as a composite of two instantaneous actions, $turnHandle$ and $releaseHandle$, whose effects are to make the fluent $locked(s)$ false and true respectively. In the same spirit, we treat the action of pushing on a door, which also intuitively has a duration, as a composite of two instantaneous actions $startPush$ and $endPush$, whose effects are to make the fluent $pushing(s)$ true and false respectively. The appropriate successor state axiom for *open* is:

$$open(do(a, s)) \equiv pushing(s) \wedge a = turnHandle \vee \\ \neg locked(s) \wedge a = startPush \vee open(s).$$

Those for $pushing$ and $locked$ are:

$$pushing(do(a, s)) \equiv a = startPush \vee pushing(s) \wedge a \neq endPush,$$

$$locked(do(a, s)) \equiv a = releaseHandle \vee locked(s) \wedge a \neq turnHandle.$$

Another interesting example is the following. Turning on the hot water faucet causes hot water to run (denoted by the fluent $hot(s)$); similarly for turning on the cold. Both the hot and cold water faucets share a common spout, so if only the hot water is running, you will burn your hand.

$$Poss(turnonHot, s) \equiv \neg hot(s),$$

$$Poss(turnonCold, s) \equiv \neg cold(s),$$

$$Poss(turnoffHot, s) \equiv hot(s),$$

$$Poss(turnoffCold, s) \equiv cold(s),$$

$$hot(do(a, s)) \equiv a = turnonHot \vee hot(s) \wedge a \neq turnoffHot,$$

$$cold(do(a, s)) \equiv a = turnonCold \vee cold(s) \wedge a \neq turnoffCold.$$

The following successor state axiom captures the conditions for burning oneself:

$$burn(do(a, s)) \equiv hot(s) \wedge a = turnoffCold \vee \neg cold(s) \wedge a = turnonHot \vee burn(s).$$

7.2.2 Limitations of Interleaved Concurrency

Despite the wide applicability of interleaved concurrency with instantaneous actions, there are examples for which true concurrency appears to be more appropriate and convenient. The standard example that seems not to be representable by an interleaving account—but see Exercise 4 below—is the scenario of a duel, in which an instantaneous shoot action by one duelist causes the death of the other. Being alive is a precondition for shooting. If both duelists shoot simultaneously, both die, whereas, in the absence of true concurrency, only one death can result from a duel. A more interesting setting where true concurrency seems appropriate is in the modeling of physical systems, such as a number of objects tracing out trajectories under Newtonian equations of motion. These equations may predict the simultaneous collisions of several objects, and an interleaving axiomatization for predicting and simulating these occurrences is clumsy and unnatural. Section 7.9 below shows how to axiomatize such physical systems using true concurrency.

7.3 The Sequential, Temporal Situation Calculus

In this section, we add an explicit representation for time to the sequential situation calculus. This will allow us to specify the exact times, or a range of times, at which actions in a world history must occur. For simplicity, and because we remain interested in interleaved concurrency, we continue to consider instantaneous actions. We want to represent the fact that a given such action occurs at a particular time. Recall that in the situation calculus, actions are denoted by first-order terms, like $startMeeting(Susan)$ or $bounce(ball, wall)$. Our proposal for adding a time dimension to the situation calculus is to add a new temporal argument to all instantaneous actions, denoting the actual time at which that action occurs. Thus, $startMeeting(Susan, t)$ might be the instantaneous action of $Susan$ starting a meeting at time t, and $bounce(ball, wall, 7.3)$ might be the instantaneous action of $ball$ bouncing against $wall$ at time 7.3.

We now investigate how to extend the foundational axioms for the sequential situation calculus (Section 4.2.2) to accommodate time. These four foundational axioms remain exactly the same as before; it will be necessary only to add one new axiom to them, and to introduce two new function symbols into $\mathcal{L}_{sitcalc}$.

First, introduce a new function symbol $time : action \rightarrow reals$. $time(a)$ denotes the time of occurrence of action a. This means that in any application involving a particular action $A(\vec{x}, t)$, we shall need an axiom specifying the occurrence time of the action A:

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

for example, $time(startMeeting(person, t)) = t$. Next, it will be convenient to have a new function symbol $start : situation \rightarrow reals$. $start(s)$ denotes the start time of situation s. This requires the new foundational axiom:

$$start(do(a, s)) = time(a). \tag{7.1}$$

Notice that we do not define the start time of S_0; this is arbitrary, and may (or may not) be specified to be any real number, depending on the application. Notice also that we are imagining temporal variables to range over the reals, although nothing prevents them from ranging over the integers, rationals, or anything else on which a binary relation $<$ is defined. In this connection, we are not providing axioms for the reals (or integers), but rely instead on the standard interpretation of the reals and their operations (addition, multiplication, etc.) and relations ($<$, $\leq$, etc.).

Next, we need to reconsider the relation $executable(s)$ on situations. Recall that this was defined to be an abbreviation for a formula that intuitively says that all the actions occurring in the action sequence s can be executed one after the other. Consider the situation

$$do(bounce(B, W, 4), do(startMeeting(Susan, 6), S_0)),$$

in which the time of the second action precedes that of the first. Intuitively, such an action sequence should not be considered possible, and we suitably amend the abbreviation $executable(s)$ of (4.5):

$$\begin{aligned} executable(s) \stackrel{def}{=} (\forall a, s^*).do(a, s^*) \sqsubseteq s \supset \\ Poss(a, s^*) \wedge start(s^*) \leq time(a). \end{aligned} \tag{7.2}$$

Now, $executable(s)$ means that all the actions in s are possible, and moreover, the times of those action occurrences are nondecreasing.

Finally, notice that the constraint $start(s^*) \leq time(a)$ in abbreviation (7.2) permits action sequences in which the time of an action may be the same as the time of a preceding action. For example,

$$do(endLunch(Bill, 4), do(startMeeting(Susan, 4), S_0)),$$

might be a perfectly good executable situation. This situation is defined by a sequence of

two actions, each of which has the same occurrence time. We allow for this possibility because often an action occurrence serves as an enabling condition for the simultaneous occurrence of another action. For example, cutting a weighted string at time t enables the action $startFalling(t)$. Both actions occur at the same time, but conceptually, the falling event happens "immediately after" the cutting. Accordingly, we want to allow the situation $do(startFalling(t), do(cutString(t), S_0))$.

The four axioms of Σ, namely the old foundational axioms for the situation calculus, together with (7.1) are the foundational axioms for the *sequential, temporal situation calculus*.

7.3.1 Concurrent Temporal Processes

With the capacity to explicitly represent time, we can now give an interleaving account for processes that overlap temporally in arbitrarily complex ways. Figure 7.1 illustrates a possible time line for instances of three overlapping processes, $walking(x, y, s)$, $singing(s)$, and $chewingGum(s)$.

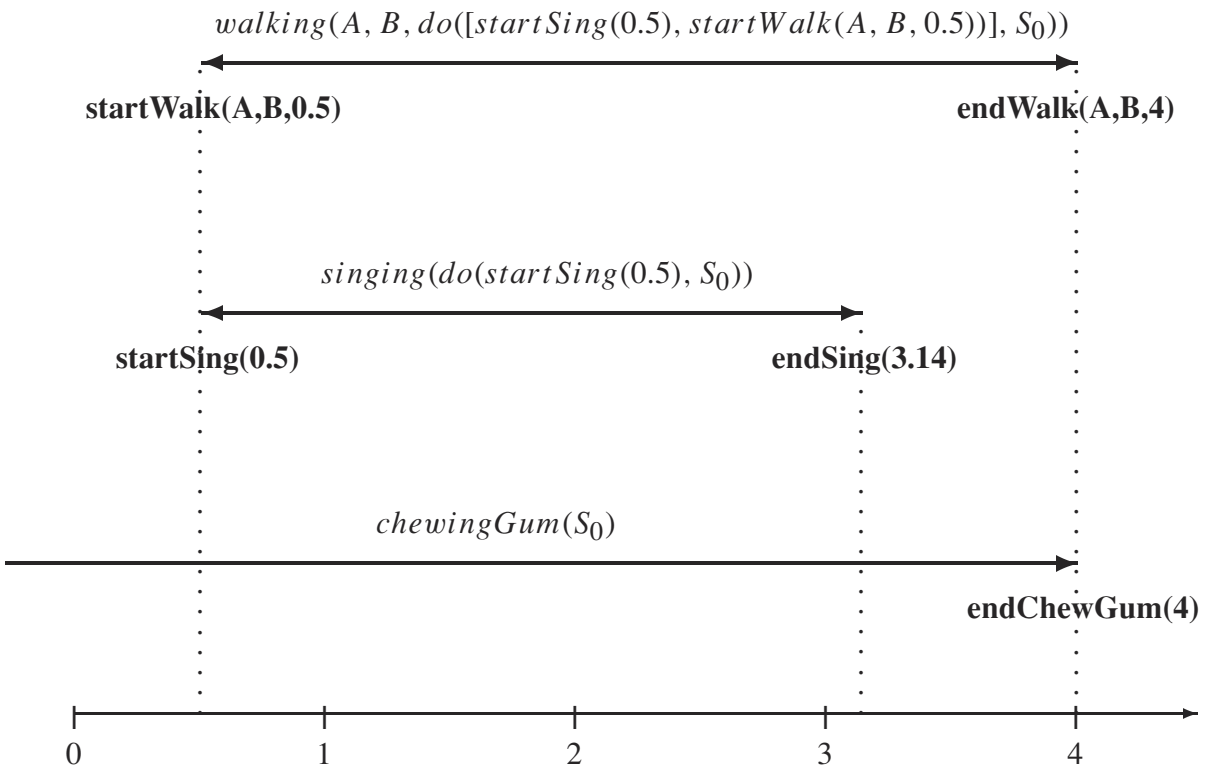

Figure 7.1: Temporal Processes in the Situation Calculus.

The indicated instances of the *singing* and *walking* processes begin together (but the first initiates "before" the second), and they terminate at different times. The instance of the *chewingGum* process was in progress initially (and therefore does not have an initiating action) and it terminates at the same time as the *walking* process.

7.4 Sequential, Temporal Golog

With the above axioms and abbreviations for the sequential, temporal situation calculus in hand, it is easy to modify the Golog semantics and interpreter to accommodate time. Semantically, we need only change the definition of the Do macro for primitive actions (Section 6.1) to:

$$Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge start(s) \leq time(a[s]) \wedge s' = do(a[s], s). \qquad (7.3)$$

Everything else about the definition of Do remains the same. To suitably modify the Golog interpreter of Section 6.3.1, replace the clause

```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).
```

by

```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S),
                   start(S,T1), time(E,T2), T1 =< T2.
```

Finally, because of the new predicate, *start*, taking a situation argument, the earlier Golog interpreter must be augmented by the clauses:

```
restoreSitArg(start(T),S,start(S,T)).
start(do(A,S),T) :- time(A,T).
```

We can now write sequential, temporal Golog programs. However, to execute such programs, the Golog interpreter must have a temporal reasoning component. It must, for example, be able to infer that $T_1 = T_2$ when given that $T_1 \leq T_2 \wedge T_2 \leq T_1$. While such a special purpose temporal theorem-prover could be written and included in the Golog interpreter, this would involve a great deal of effort. To illustrate the use of temporal Golog, we shall instead rely on a logic programming language with a built-in constraint solving capability. Specifically, we shall appeal to the ECRC Common Logic Programming System ECLIPSE 3.5.2, which provides a built-in Simplex algorithm for solving linear equations and inequalities over the reals. So we shall assume that our Golog program makes use of linear temporal relations like $2 * T_1 + T_2 = 5$ and $3 * T_2 - 5 \leq 2 * T_3$, and rely on ECLIPSE to perform the reasoning for us in the temporal domain. ECLIPSE provides a special syntax for those relations over the reals recognized by its built-in theorem-prover. These relations are: $=$, $\neq$, $\geq$, $>$, $\leq$, $<$, which are represented in ECLIPSE by the infix

$ =, $ <>, $ >=, $ >, $ <=, $ <, respectively. So, in ECLIPSE, the above modification of the Golog interpreter to include time is:

```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S),
                   start(S,T1), time(E,T2), T1 $<= T2.
```

All other clauses of the earlier interpreter of Section 6.3.1 will work correctly under ECLIPSE.

7.4.1 Example: A Coffee Delivery Robot

Here, we describe a robot whose task is to deliver coffee in an office environment. The robot is given a schedule of the preferred coffee periods of every employee, as well as information about the times it takes to travel between various locations in the office environment. The robot can carry just one cup of coffee at a time, and there is a central coffee machine from which it gets coffee. The robot's task is to schedule coffee deliveries in such a way that, if possible, everyone gets coffee during his/her preferred time periods.

Primitive actions:

- $pickupCoffee(t)$. The robot picks up a cup of coffee from the coffee machine at time t.
- $giveCoffee(person, t)$. The robot gives a cup of coffee to $person$ at time t.
- $startGo(loc_1, loc_2, t)$. The robot starts to go from location loc_1 to loc_2 at time t.
- $endGo(loc_1, loc_2, t)$. The robot ends its process of going from location loc_1 to loc_2 at time t.

Fluents:

- $robotLocation(s)$. A functional fluent denoting the robot's location in situation s.
- $hasCoffee(person, s)$. $person$ has a cup of coffee in situation s.
- $going(loc_1, loc_2, s)$. In situation s, the robot is going from loc_1 to loc_2.
- $holdingCoffee(s)$. In situation s, the robot is holding a cup of coffee.

Situation-Independent Predicates and Functions:

- $wantsCoffee(person, t_1, t_2)$. $person$ wants to receive coffee at some point in the time period $[t_1, t_2]$.
- $office(person)$. Denotes the office inhabited by $person$.
- $travelTime(loc_1, loc_2)$. Denotes the amount of time that the robot takes to travel between loc_1 and loc_2.
- CM. A constant denoting the location of the coffee machine.
- Sue, $Mary$, $Bill$, Joe. Constants denoting office people.

Primitive Action Preconditions:

$$Poss(pickupCoffee(t), s) \equiv \neg holdingCoffee(s) \wedge robotLocation(s) = CM,$$

$$Poss(giveCoffee(person, t), s) \equiv holdingCoffee(s) \wedge robotLocation(s) = office(person),$$

$$Poss(startGo(loc_1, loc_2, t), s) \equiv \neg(\exists l, l')going(l, l', s) \wedge loc_1 \neq loc_2 \wedge robotLocation(s) = loc_1,$$

$$Poss(endGo(loc_1, loc_2, t), s) \equiv going(loc_1, loc_2, s).$$

Successor State Axioms:

$$hasCoffee(person, do(a, s)) \equiv (\exists t)a = giveCoffee(person, t) \vee hasCoffee(person, s),$$

$$robotLocation(do(a, s)) = loc \equiv (\exists t, loc')a = endGo(loc', loc, t) \vee robotLocation(s) = loc \wedge \neg(\exists t, loc', loc'')a = endGo(loc', loc'', t),$$

$$going(l, l', do(a, s)) \equiv (\exists t)a = startGo(l, l', t) \vee going(l, l', s) \wedge \neg(\exists t)a = endGo(l, l', t),$$

$$holdingCoffee(do(a, s)) \equiv (\exists t)a = pickupCoffee(t) \vee holdingCoffee(s) \wedge \neg(\exists person, t)a = giveCoffee(person, t).$$

Initial Situation:

Unique names axioms stating that the following terms are pairwise unequal:

$$Sue,\ Mary,\ Bill,\ Joe,\ CM,$$
$$office(Sue),\ office(Mary),\ office(Bill),\ office(Joe).$$

Initial Fluent values:

$$robotLocation(S_0) = CM,\ \ \neg(\exists l, l')going(l, l', S_0),$$
$$\neg holdingCoffee(S_0),\ \ start(S_0) = 0,\ \ \neg(\exists p)hasCoffee(p, S_0).$$

Coffee delivery preferences. The following expresses the fact that all, and only, the tuples satisfying the *wantsCoffee* relation are $(Sue, 140, 160), \ldots, (Joe, 90, 100)$. In other words, this relation is definitional.

$$wantsCoffee(p, t_1, t_2) \equiv p = Sue \wedge t_1 = 140 \wedge t_2 = 160 \vee p = Mary \wedge t_1 = 130 \wedge t_2 = 170 \vee p = Bill \wedge t_1 = 100 \wedge t_2 = 110 \vee p = Joe \wedge t_1 = 90 \wedge t_2 = 100.$$

Robot travel times:

$$travelTime(CM, office(Sue)) = 15, \quad travelTime(CM, office(Mary)) = 10,$$

$travelTime(CM, office(Bill)) = 8, \quad travelTime(CM, office(Joe)) = 10.$
$travelTime(l, l') = travelTime(l', l), \quad travelTime(l, l) = 0.$

Action Occurrence Times:

$time(pickupCoffee(t)) = t, \quad time(giveCoffee(person, t)) = t,$
$time(startGo(loc_1, loc_2, t)) = t, \quad time(endGo(loc_1, loc_2, t)) = t.$

Golog Procedures:

proc $deliverCoffee(t)$ % Beginning at time t the robot serves coffee to everyone,
% if possible. Else the program fails.
$now \leq t?$;
$\{[(\forall p, t', t'').wantsCoffee(p, t', t'') \supset hasCoffee(p)]?$
|
if $robotLocation = CM$ **then** $deliverOneCoffee(t)$
else $goto(CM, t)$; $deliverOneCoffee(now)$
endIf} ;
$deliverCoffee(now)$
endProc

The above procedure introduces a functional fluent $now(s)$, which is exactly the same as the fluent $start(s)$. We prefer it here to $start$ because it has a certain mnemonic value, but like $start$, it denotes the *current time*.

proc $deliverOneCoffee(t)$ % Assuming the robot is at the coffee machine,
% it delivers one cup of coffee.
$(\pi p, t_1, t_2, wait)[\{wantsCoffee(p, t_1, t_2) \wedge \neg hasCoffee(p) \wedge wait \geq 0 \wedge$
$t_1 \leq t + wait + travelTime(CM, office(p)) \leq t_2\}?$;
$pickupCoffee(t + wait)$;
$goto(office(p), now)$;
$giveCoffee(p, now)$
endProc

proc $goto(loc, t)$ % Beginning at time t the robot goes to loc.
$goBetween(robotLocation, loc, travelTime(robotLocation, loc), t)$
endProc

proc $goBetween(loc_1, loc_2, \Delta, t)$ % Beginning at time t the robot goes from loc_1
% to loc_2, taking Δ time units for the transition.
$startGo(loc_1, loc_2, t)$; $endGo(loc_1, loc_2, t + \Delta)$
endProc

The following sequential, temporal Golog program implements the above specification.

Sequential, Temporal Golog Program for a Coffee Delivery Robot

```
% Golog Procedures.

proc(deliverCoffee(T),
  ?(some(t, now(t) & t $<= T)) :
  (?(all(p,all(t1,all(t2,wantsCoffee(p,t1,t2) => hasCoffee(p)))))
   #
   pi(rloc, ?(robotLocation(rloc)) :
            if(rloc = cm, /* THEN */ deliverOneCoffee(T),
            /* ELSE */ goto(cm,T) : pi(t, ?(now(t)) :
                                       deliverOneCoffee(t))) :
            pi(t, ?(now(t)) : deliverCoffee(t)))))).

proc(deliverOneCoffee(T),
  pi(p, pi(t1, pi(t2, pi(wait, pi(travTime,
     ?(wantsCoffee(p,t1,t2) & -hasCoffee(p) & wait $>= 0 &
       travelTime(cm,office(p),travTime) &
       t1 $<= T + wait + travTime & T + wait + travTime $<= t2) :
     pi(t, ?(t $= T + wait) : pickupCoffee(t)) :
     pi(t, ?(now(t)) : goto(office(p),t)) :
     pi(t, ?(now(t)) : giveCoffee(p,t)))))))).

proc(goto(L,T),
  pi(rloc,?(robotLocation(rloc)) :
    pi(deltat,?(travelTime(rloc,L,deltat)) :
      goBetween(rloc,L,deltat,T)))).

proc(goBetween(Loc1,Loc2,Delta,T),
   startGo(Loc1,Loc2,T) :
   pi(t, ?(t $= T + Delta) : endGo(Loc1,Loc2,t))).

% Preconditions for Primitive Actions.

poss(pickupCoffee(T),S) :- not holdingCoffee(S), robotLocation(cm,S).
poss(giveCoffee(Person,T),S) :- holdingCoffee(S),
                                robotLocation(office(Person),S).
poss(startGo(Loc1,Loc2,T),S) :- not going(L,LL,S),
                                not Loc1 = Loc2, robotLocation(Loc1,S).
poss(endGo(Loc1,Loc2,T),S) :- going(Loc1,Loc2,S).

% Successor State Axioms.
```

```
hasCoffee(Person,do(A,S)) :- A = giveCoffee(Person,T) ;
                             hasCoffee(Person,S).
robotLocation(Loc,do(A,S)) :- A = endGo(Loc1,Loc,T) ;
                      robotLocation(Loc,S), not A = endGo(Loc2,Loc3,T).
going(Loc1,Loc2,do(A,S)) :- A = startGo(Loc1,Loc2,T) ;
                        going(Loc1,Loc2,S), not A = endGo(Loc1,Loc2,T).
holdingCoffee(do(A,S)) :- A = pickupCoffee(T) ;
                        holdingCoffee(S), not A = giveCoffee(Person,T).

% Initial Situation.

robotLocation(cm,s0).

start(s0,0).

wantsCoffee(sue,140,160).   wantsCoffee(bill,100,110).
wantsCoffee(joe,90,100).    wantsCoffee(mary,130,170).

travelTime0(cm,office(sue),15).  travelTime0(cm,office(mary),10).
travelTime0(cm,office(bill),8).  travelTime0(cm,office(joe),10).

travelTime(L,L,0).
travelTime(L1,L2,T) :- travelTime0(L1,L2,T) ; travelTime0(L2,L1,T).

% The time of an action occurrence is its last argument.

time(pickupCoffee(T),T).       time(giveCoffee(Person,T),T).
time(startGo(Loc1,Loc2,T),T).  time(endGo(Loc1,Loc2,T),T).

% Restore situation arguments to fluents.

restoreSitArg(robotLocation(Rloc),S,robotLocation(Rloc,S)).
restoreSitArg(hasCoffee(Person),S,hasCoffee(Person,S)).
restoreSitArg(going(Loc1,Loc2),S,going(Loc1,Loc2,S)).
restoreSitArg(holdingCoffee,S,holdingCoffee(S)).

% Primitive Action Declarations.

primitive_action(pickupCoffee(T)).
primitive_action(giveCoffee(Person,T)).
primitive_action(startGo(Loc1,Loc2,T)).
primitive_action(endGo(Loc1,Loc2,T)).

% Fix on a solution to the temporal constraints.
```

```
chooseTimes(s0).
chooseTimes(do(A,S)) :- chooseTimes(S), time(A,T), rmin(T).

% "now" is a synonym for "start".

now(S,T) :- start(S,T).
restoreSitArg(now(T),S,now(S,T)).

%  Utilities.

prettyPrintSituation(S) :- makeActionList(S,Alist), nl,
                           write(Alist), nl.

makeActionList(s0,[]).
makeActionList(do(A,S),L) :- makeActionList(S,L1), append(L1,[A],L).

coffeeDelivery(T) :- do(deliverCoffee(T),s0,S), chooseTimes(S),
                     prettyPrintSituation(S), askForMore.

askForMore :- write('More? '), read(n).
```

One problem with this constraint logic programming approach to coffee delivery is that the execution of the Golog call do(deliverCoffee(1),s0,S) will not, in general, result in a fully instantiated sequence of actions S. The actions in that sequence will not have their occurrence times uniquely determined; rather, these occurrence times will consist of all feasible solutions to the system of constraints generated by the program execution. So, to get a fixed schedule of coffee delivery, we must determine one or more of these feasible solutions. The relation chooseTimes(S) in the above program does just that. It takes a situation term as its argument. Beginning with the first action in that situation history, chooseTimes determines the time of that action (which, in general, will be a Prolog variable since the ECLIPSE constraint solver will not have determined a unique value for that action's occurrence time). It then minimizes (via rmin(T)) that time, relative to the current set of temporal constraints generated by executing the coffee delivery program. Then, having fixed the occurrence time of the first action, it repeats with the second action, etc. In this way, chooseTimes selects a particular solution to the linear temporal constraints generated by the program, thereby producing one of many possible schedules for the robot.

The following is the output obtained by running this coffee delivery program under the temporal Golog interpreter of Section 7.4. Before loading and running these programs, the ECLIPSE rational constraint solver must be loaded from its library by entering lib(r). There are two, qualitatively different solutions to this scheduling problem.

Running the Coffee delivery Program

```
[eclipse 2]: coffeeDelivery(1).

[pickupCoffee(80), startGo(cm,office(joe),80),
 endGo(cm,office(joe),90), giveCoffee(joe,90),
 startGo(office(joe),cm,90), endGo(office(joe),cm,100),
 pickupCoffee(100), startGo(cm,office(bill),100),
 endGo(cm,office(bill),108), giveCoffee(bill, 108),
 startGo(office(bill),cm,108), endGo(office(bill),cm,116),
 pickupCoffee(125), startGo(cm,office(sue),125),
 endGo(cm,office(sue),140), giveCoffee(sue, 140),
 startGo(office(sue),cm,140), endGo(office(sue),cm,155),
 pickupCoffee(155), startGo(cm,office(mary),155),
 endGo(cm,office(mary),165), giveCoffee(mary,165)]
More?  y.

[pickupCoffee(80), startGo(cm,office(joe),80),
 endGo(cm,office(joe),90), giveCoffee(joe,90),
 startGo(office(joe),cm,90), endGo(office(joe),cm,100),
 pickupCoffee(100),startGo(cm,office(bill),100),
 endGo(cm,office(bill),108), giveCoffee(bill,108),
 startGo(office(bill),cm,108), endGo(office(bill),cm,116),
 pickupCoffee(120), startGo(cm,office(mary),120),
 endGo(cm,office(mary),130), giveCoffee(mary,130),
 startGo(office(mary),cm,130), endGo(office(mary),cm,140),
 pickupCoffee(140), startGo(cm,office(sue),140),
 endGo(cm,office(sue),155), giveCoffee(sue,155)]
More?  y.

no (more) solution.
```

7.4.2 A Singing Robot

To simplify the exposition, we did not endow the above program with any interleaving execution of processes, as described in Section 7.3.1. This would, however, be easy to do. Suppose we wanted the robot to sing a song, but only while it is in transit between locations. Introduce two instantaneous actions $startSing(t)$ and $endSing(t)$, and a process fluent $singing(s)$, with action precondition and successor state axioms:

$$Poss(startSing(t), s) \equiv \neg singing(s).$$

$$Poss(endSing(t), s) \equiv singing(s),$$

$$singing(do(a, s)) \equiv (\exists t)a = startSing(t) \vee singing(s) \wedge \neg(\exists t)a = endSing(t).$$

Then the following version of the Golog procedure $goBetween$ turns the robot into a singing waiter:

proc $goBetween(loc1, loc2, \Delta, t)$
  $startGo(loc1, loc2, t)$;
  $startSing(t)$; $endSing(t + \Delta)$;
  $endGo(loc1, loc2, t + \Delta)$
endProc

This provides a temporal, interleaving account of the concurrent execution of two processes: singing and moving between locations.

7.4.3 Plan-Execution Monitoring

While we now know how to specify and compute robot schedules with temporal Golog, it would be a serious mistake to believe that the problem of controlling a robot over time has therefore been solved. It would be unrealistic to expect a robot to execute a schedule like that returned by the coffee delivery program. Frequently, it will be impossible to meet the exact times in such a schedule, for example, if the robot is unexpectedly delayed in traveling to the coffee machine. Moreover, travel times cannot be precisely predicted, and errors necessarily arise due to mechanical factors like the robot's wheels slipping on a smooth floor. So a coffee delivery schedule like that computed above must be viewed as an idealized artifact; its physical realization by a real robot is unlikely to succeed at the exact times called for by the schedule. How then can one make use of such a schedule in controlling an imperfect robot in an imperfect world? This is an instance of the general problem of specifying how a robot can monitor, and correct, its own execution when it is following a predetermined plan, like the coffee delivery schedule. Insofar as is possible, the robot should follow its plan, but it is permitted, at plan-execution time, to make suitable modifications to that plan in order to accommodate unexpected situations. How does the robot determine that its current plan has failed? How does it repair a failed plan? When does it give up completely on a plan, and what does it do instead?

Plan execution monitoring is a difficult and largely unsolved problem, and we are not about to tackle it here. Nevertheless, it is instructive to look at the problem a bit more closely. To begin, the robot can detect plan failure only by sensing its environment. It sees that it is not in John's office; it reads its internal clock to determine that it is now too late to serve Mary her coffee. So it seems that if we are to take plan-execution monitoring seriously, *we need an account of sense actions*. Notice that such actions are fundamentally different from the "ordinary" actions (picking up a block, moving an elevator) we have considered thus far. Ordinary actions change the physical world; sense actions do not. Except in quantum mechanics, reading the time does not cause any physical changes to the

world. What then are the effects of sense actions? Rather than changing the state of the world, *sense actions change the mental state of the sensing agent*. After performing such an action, the agent may come to know something he did not know before—it's now 3PM; John is in his office. In fact, sense actions are often called *knowledge-producing actions*. So it seems that, in order to model sense actions, we shall need a situation calculus account that distinguishes between the state of the physical world, and the mental state of an agent, and that, moreover, formalizes what it means for an agent to know something. With such a formal story in hand, we can then axiomatically express causal laws for sense actions, such as: sensing a clock causes the sensing agent to know what time it is. Furthermore, we can extend our approach to the frame problem for ordinary actions to include sense actions. All of which is an extended advertisement and motivation for Chapter 11, which gives just such a formal treatment of sense actions.

Returning to the original problem, we can imagine a coffee delivery robot that monitors its own execution, recomputing what remains of the schedule, after it has determined (by sensing its internal clock) the actual occurrence times of its actions. We do not instantiate a schedule's action occurrence times (as we did using chooseTimes(S)), but leave these free, subject to the constraints generated by the Golog program. Whenever the robot physically performs an action, it senses the action's actual occurrence time, adds this to the constraints, then computes a remaining schedule, or fails if no continuing schedule can be found.

7.5 The Concurrent, Non-Temporal Situation Calculus

In this section, we focus on true concurrency for primitive actions, ignoring time for the moment. So the picture will be the same as for the sequential situation calculus that has been developed so far, except that many actions may occur together. This means that there will be no explicit representation for the time of an action occurrence, and situations will be sequences of concurrent action occurrences instead of sequences of single action occurrences as in our earlier development. To represent concurrent actions, we shall use sets of simple actions. To avoid the conceptual problems described in Section 7.1, we shall restrict ourselves to actions all of which have equal, but unspecified durations. These durations could be zero, in which case concurrent actions will be represented by sets of instantaneous actions.

We now consider how to represent such concurrent actions in the situation calculus, which we do by treating concurrent actions as sets, possibly infinite, of simple actions. As we shall see later, the possibility of infinitely many actions occurring concurrently must be taken seriously, so that the obvious notation $a_1 \| a_2 \| \cdots \| a_n$ cannot accommodate this

possibility. Because concurrent actions are sets of simple actions, we can use the notation $a \in c$ to mean that simple action a is one of the actions of the concurrent action c. We do not axiomatize sets, but instead rely on the standard interpretation of sets and their operations (union, intersection, etc.) and relations (membership, subset, etc.). This is in the same spirit as our treatment of time; we did not axiomatize the reals for this purpose, but instead relied on the standard interpretation of the reals and their operations (addition, multiplication etc.) and relations ($<$, $\leq$, etc.). To distinguish the sorts *action* of simple actions and *concurrent*, we use variables $a, a', \ldots,$ and $c, c', \ldots,$ respectively.

Next, we consider how to generalize the foundational axioms of the sequential situation calculus (Section 4.2.2) to provide for concurrency. As we do so, keep in mind that, conceptually, all simple actions have identical, but unspecified durations and the co-occurrence of two or more such actions means that they all begin and end together. To begin, we need to view situations as sequences of concurrent actions, so we extend the function symbol do to take concurrent actions as an argument. Then we have situation terms like $do(\{startMeeting(Sue), collide(A, B)\}, S_0)$.

After extending do in this way, the rest is easy; simply replace each *action* variable in the foundational axioms by a variable of sort *concurrent action*:

$$(\forall P).P(S_0) \wedge (\forall c, s)[P(s) \supset P(do(c, s))] \supset (\forall s)P(s), \tag{7.4}$$

$$do(c, s) = do(c', s') \supset c = c' \wedge s = s', \tag{7.5}$$

$$\neg s \sqsubset S_0, \tag{7.6}$$

$$s \sqsubset do(c, s') \equiv s \sqsubseteq s'. \tag{7.7}$$

The abbreviations for $executable(s)$ becomes:

$$executable(s) \stackrel{def}{=} (\forall c, s^*).do(c, s^*) \sqsubseteq s \supset Poss(c, s^*).$$

Notice that now the predicate $Poss$ is permitted to take a concurrent action as its first argument, so in axiomatizing a particular application, we shall need to specify the conditions under which certain concurrently occurring actions are possible. What can one say in general about the preconditions of concurrent actions? At the very least, we need:

$$Poss(a, s) \supset Poss(\{a\}, s), \tag{7.8}$$

$$Poss(c, s) \supset (\exists a)a \in c \wedge (\forall a)[a \in c \supset Poss(a, s)]. \tag{7.9}$$

This last axiom tells us that if a concurrent action is possible, then it contains at least one action, and all its simple actions are possible. As we shall see later, the converse need not hold.

The six axioms (7.4) - (7.9) are the foundational axioms for the *concurrent, non-temporal situation calculus*. Notice that, except for axioms (7.8) and (7.9), these are identical to those for the sequential situation calculus, with the exception that they refer

to concurrent actions instead of simple actions.

7.6 Axiomatizing Concurrent Worlds

7.6.1 Successor State Axioms

In the sequential situation calculus, we provided a solution to the frame problem by appealing to a systematic way of obtaining successor state axioms from the effect axioms. For the concurrent setting, we have to generalize these successor state axioms slightly; this turns out to be quite straightforward, as the following example shows:

$$pickingUp(x, do(c, s)) \equiv startPickup(x) \in c \vee \\ pickingUp(x, s) \wedge endPickup(x) \notin c.$$

The next example axiomatizes the two duelists scenario that, intuitively, cannot be captured by an interleaving account of concurrency. Suppose that $shoot(x, y)$ is the instantaneous action of person x shooting person y.

$$dead(x, do(c, s)) \equiv (\exists y) shoot(y, x) \in c \vee dead(x, s).$$

Then it is easy to prove that

$$dead(Tom, do(\{shoot(Tom, Harry), shoot(Harry, Tom)\}, S_0))$$

and

$$dead(Harry, do(\{shoot(Tom, Harry), shoot(Harry, Tom)\}, S_0)),$$

but that

$$dead(Harry, do(\{shoot(Tom, Harry)\}, S_0))$$

and

$$\neg dead(Tom, do(\{shoot(Tom, Harry)\}, S_0)) \equiv \neg dead(Tom, S_0).$$

7.6.2 Action Precondition Axioms

The earlier approach to axiomatizing dynamic worlds in the situation calculus relied on a collection of *action precondition axioms*, one for each simple action, and we also rely on such axioms here. However, concurrency introduces certain complications, which we now describe.

The Precondition Interaction Problem

In the case of action preconditions for concurrent actions, the converse of (7.9) need not hold. Two simple actions may each be possible, their action preconditions may be jointly

consistent, yet intuitively they should not be concurrently possible. We call this the *precondition interaction problem*. Here is a simple example:

$$Poss(startMoveLeft, s) \equiv \neg movingLeft(s),$$
$$Poss(startMoveRight, s) \equiv \neg movingRight(s).$$

Intuitively, $Poss(\{startMoveLeft, startMoveRight\}, s)$ should be false. Such impossible combinations of actions are the reason that the foundational axiom (7.9) was not a biconditional; the converse of axiom (7.9) can be false.

Notice that the precondition interaction problem arises only when modeling true concurrency. This is one reason why, whenever possible, one should appeal to an interleaving account for concurrency.

Infinitely Many Actions Can Co-Occur

Nothing prevents one from writing:

$$Poss(A(x), s) \equiv true,$$

in which case $A(x)$ can co-occur, for all x. So if x ranges over the natural numbers (or the reals, or ...) we get lots of possible co-occurrences. This is the principal reason that we chose to represent concurrent actions by sets, rather than a function symbol $\|$, because one cannot always denote a concurrent collection of actions by $a_1 \| a_2 \| \cdots \| a_n$. Unfortunately, there does not seem to be a natural way, in the foundational axioms, to rule out the possibility of infinitely many co-occurring actions.

Notice that this problem cannot arise for interleaved concurrency, which is yet another reason for modeling concurrency with interleaving, whenever possible.

7.7 The Concurrent, Temporal Situation Calculus

Section 7.3 extended the sequential, non-temporal situation calculus to represent time by providing an explicit time argument to actions. We now investigate how to modify and extend the foundational axioms for the concurrent, non-temporal situation calculus (Section 7.5) to accommodate time. As before, actions will be viewed as instantaneous, and will have an explicit temporal argument denoting the time at which the action occurs. We remain committed to the induction axiom (7.4), and unique names axiom (7.5) for situations, as in Section 7.5, as well as the axioms for $\sqsubset$. Recall that for the sequential, temporal situation calculus, we introduced a function symbol $time$, where $time(a)$ denotes the time of occurrence of action a, and we shall also need this function here. As before, this means that in any application involving a particular action $A(\vec{x}, t)$, there must be an axiom telling us the time of the action A:

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

A concurrent action makes no intuitive sense if it is empty, or if it contains two or more simple actions whose occurrence times are different, for example

$$\{startMeeting(Sue, 3), bounce(B, W, 4)\}.$$

Accordingly, define the notion of a coherent concurrent action to be one for which there is at least one action in the collection, and for which all of the (instantaneous) actions in the collection occur at the same time. This can be done with an abbreviation:

$$coherent(c) \stackrel{def}{=} (\exists a) a \in c \wedge (\exists t)(\forall a')[a' \in c \supset time(a') = t].$$

Now, extend the function $time$ from simple actions to concurrent ones:

$$coherent(c) \supset [time(c) = t \equiv (\exists a)(a \in c \wedge time(a) = t)]. \tag{7.10}$$

Next, it will be convenient to have a function $start$: $start(s)$ denotes the start time of situation s. This requires the axiom:

$$start(do(c, s)) = time(c). \tag{7.11}$$

Notice that we do not define the start time of S_0; this is arbitrary, and may (or may not) be specified to be any real number, depending on the application.

We also need to slightly revise the abbreviation (7.2) for $executable(s)$ to accommodate concurrent actions:

$$executable(s) \stackrel{def}{=} (\forall c, s^*).do(c, s^*) \sqsubseteq s \supset Poss(c, s^*) \wedge start(s^*) \leq time(c).$$

Now, $executable(s)$ means that all the concurrent actions in s are possible, and moreover, the times of those action occurrences are nondecreasing.

Finally, as for the non-temporal concurrent case,

$$Poss(a, s) \supset Poss(\{a\}, s), \tag{7.12}$$

and we need to generalize axiom (7.9) of the concurrent, non-temporal situation calculus to the following:

$$Poss(c, s) \supset coherent(c) \wedge (\forall a)[a \in c \supset Poss(a, s)]. \tag{7.13}$$

This tells us that if a concurrent action is possible, then it is coherent and all its simple actions are possible. Because of the precondition interaction problem, we do not adopt the converse of (7.13) as an axiom.

The sentences (7.10)–(7.13)—together with the axioms (7.4)–(7.7) of Section 7.5 for induction, unique names for situations, and $\sqsubset$—are the foundational axioms for the *concurrent, temporal situation calculus*.

7.8 Concurrent, Temporal Golog

With an axiomatization for the concurrent, temporal situation calculus, it is easy to modify the Golog semantics and interpreter to accommodate these features. Semantically, we need only change the definition (7.3) of the Do macro for sequential, temporal Golog to apply to concurrent actions instead of simple actions:

$$Do(c, s, s') \stackrel{def}{=} Poss(c[s], s) \wedge start(s) \leq time(c[s]) \wedge s' = do(c[s], s).$$

Everything else about the definition of Do remains the same. To suitably modify the sequential, temporal Golog interpreter of Section 7.4, replace the clause

```
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S),
                   start(S,T1), time(E,T2), T1 =< T2.
```

by

```
do(C,S,do(C,S)) :- concurrent_action(C), poss(C,S),
                   start(S,T1), time(C,T2), T1 =< T2.
```

Suitable clauses must also be included for concurrent_action and time. The following is the resulting interpreter.

A Prolog Interpreter for Concurrent, Temporal Golog

```
do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,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 # ?(-P) : E2,S,S1).
do(star(E),S,S1) :- S1 = S ; do(E : star(E),S,S1).
do(while(P,E),S,S1):- do(star(?(P) : E) : ?(-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(C,S,do(C,S)) :- concurrent_action(C), poss(C,S), start(S,T1),
                   time(C,T2), T1 =< T2.

/* The time of a concurrent action is the time of its first simple
   action. This assumes that the concurrent action is coherent: it is
   non-empty, and all its simple actions occur at the same time.  */

time([A | C],T) :- time(A,T).

% The start time of situation do(C,S) is the time of C.

start(do(C,S),T) :- concurrent_action(C), time(C,T).
```

```
%  Restore suppressed situation arguments.

restoreSitArg(start(T),S,start(S,T)).
restoreSitArg(poss(A),S,poss(A,S)).

%  Concurrent actions are Prolog lists.

concurrent_action([]).    concurrent_action([A | R ]).

%  coherent actions defined.

coherent([A | R]) :- time(A,T), sameTime(R,T).
sameTime([],T).
sameTime([A | R],T) :- time(A,T), sameTime(R,T).

/*  The operator declarations, and clauses for holds and sub are
    the same as for the earlier Golog interpreter.  */
```

The next section illustrates the use of this interpreter for simulating physical systems.

7.9 Natural Actions

A natural application of the concurrent, temporal situation calculus is in modeling dynamic physical systems, where the system evolution is due to *natural actions*, namely, actions like a falling ball bouncing when it reaches the floor. Such actions obey natural laws, for example the Newtonian equations of motion. The fundamental property of such natural actions, that needs to be formally captured, is that they must occur at their predicted times, provided no earlier actions (natural or agent initiated) prevent them from occurring. Because several such actions may occur simultaneously, a theory of concurrency is needed. Because such actions may be modeled by equations of motion, continuous time must be represented. Since the concurrent, temporal situation calculus has these properties, it will provide the foundations of our approach to natural actions. This section is devoted to spelling out the details of this approach.

7.9.1 Representing Physical Laws

Our focus will be on natural actions, namely those that 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's precondition axioms, for example:

$$Poss(bounce(t), s) \equiv isFalling(s) \wedge \\ \{height(s) + vel(s)[t - start(s)] - 1/2G[t - start(s)]^2 = 0\}.$$

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.

7.9.2 Permissiveness of the Situation Calculus

Before continuing with our treatment of natural actions, it is important to think a little more carefully about the nature of agents and the actions that they can perform, as they have been modeled in the situation calculus developed thus far.

Recall that a situation is nothing but a history—a sequence of primitive actions. Among all such situations, we distinguished those that are executable, in the sense that the actions making up the sequence are all physically possible; their action preconditions are true in the situations in which they are to occur. These executable situations s were captured by the abbreviation $executable(s)$ (Section 4.6.1). But the truth of an action's preconditions does not mean that the action *must* be mentioned in every executable situation. For example, suppose A and B are two actions that are always possible: $Poss(A, s) \equiv true$, $Poss(B, s) \equiv true$. Then $do([A, \ldots, A], S_0)$ is an executable situation for any finite sequence of A's; B need not occur in every executable situation, despite the fact that its action preconditions are always possible. In this sense, the situation calculus, as presented so far, is *permissive* with respect to action occurrences; there are perfectly good executable world futures that do not include certain actions, even when it is possible to perform them. One useful way to think about this aspect of the situation calculus is that the executable situations describe all the possible ways that the world can evolve, assuming that the agent capable of performing the primitive actions has the "free will" to perform, or withhold, her actions. Thus, any particular situation represents one world history in which the agent has chosen to perform the actions in that sequence, and has chosen to withhold other actions not in the sequence.

In augmenting the situation calculus with natural actions, we can no longer allow such permissiveness in defining the executable situations. Nature does not have the free will to withhold her actions; if the time and circumstances are right for a falling ball to bounce against the floor, it must bounce. This means that in modeling physical laws in the situation calculus, we can no longer rely on our earlier concept of permissive executable situations. The executable situations must be extended so they remain permissive with respect to the free will agents in the world, but are *coercive* with respect to natural actions. The next section formally defines this new concept of an executable situation.

7.9.3 Natural Actions and Executable Situations

As discussed in the previous section, in the space of all situations, we want to single out the executable situations, i.e. those that respect the property of natural actions that they must occur at their predicted times, provided no earlier actions (natural or free-will-agent initiated) prevent them from occurring. First, introduce a predicate symbol $natural$, with which the axiomatizer can declare suitable actions to be natural, as, for example, $natural(bounce(t))$. Next, capture this new concept of an executable situation with the following abbreviation:

$$executable(s) \stackrel{def}{=} (\forall c, s^*)[do(c, s^*) \sqsubseteq s \supset Poss(c, s^*) \wedge start(s^*) \leq time(c)] \wedge (\forall a, c, s')[natural(a) \wedge Poss(a, s') \wedge do(c, s') \sqsubseteq s \supset a \in c \vee time(c) < time(a)].$$

The first condition on the right-hand side is simply the old definition of an executable situation for the temporal, concurrent situation calculus (Section 7.7). It requires of the new definition for $executable(s)$ that each of the concurrent actions in s must be possible, and the occurrence times of the actions s must be nondecreasing. The second condition on the right-hand side imposes an additional constraint on situations with respect to the occurrences of natural actions. This may initially be a bit difficult to understand; the following is provable from this abbreviation, and provides a more intuitive inductive characterization of the executable situations:

$$executable(S_0).$$

$$executable(do(c, s)) \equiv executable(s) \wedge Poss(c, s) \wedge start(s) \leq time(c) \wedge (\forall a).natural(a) \wedge Poss(a, s) \wedge a \notin c \supset time(c) < time(a).$$

Here, c is a concurrent action that, in general, will include simple actions due to free-will-agents, as well as natural actions. In making sense of this definition, keep in mind that the "laws of motion" for natural actions are encoded in the actions' precondition axioms, as described above in Section 7.9.1. Intuitively, we get a next executable situation $do(c, s)$ from the current situation s iff:

1. s is executable, and,
2. c is possible and c doesn't occur before the start of s, and,
3. Whenever a is a natural action that can occur in situation s (and therefore, being natural, it must occur *next* unless something happens before it), but a is not in c (and hence doesn't occur next), then c must occur before (not after, not at the same time as) a's predicted occurrence time.

Now, the executable situations characterize all possible world evolutions, where the non-natural actions are under the control of one or more agents with the free will to perform or withhold such action occurrences, and where the natural actions must occur if the time and circumstances predict their occurrences.

7.9.4 An Example: Enabling Actions

In the discussion following the presentation of axiom (7.2), we noted the possibility of situations containing two or more concurrent actions with the same occurrence times. We now provide an example where this is a desirable feature of our axiomatization. Consider a scenario in which an agent is holding an object. At some time, chosen under her own free will, she releases the object, enabling it to start falling. The $startFalling$ action is a natural action, which is to say, it must occur immediately after the release action. For simplicity, assume that once the object starts to fall, it continues falling forever.

$$start(S_0) = 0, \quad holding(S_0), \quad \neg falling(S_0),$$

$$natural(a) \equiv (\exists t)a = startFalling(t),$$

$$Poss(release(t), s) \equiv holding(s) \wedge start(s) \leq t,$$

$$Poss(startFalling(t), s) \equiv \neg holding(s) \wedge \neg falling(s) \wedge start(s) \leq t,$$

$$falling(do(c, s)) \equiv (\exists t)startFalling(t) \in c \vee falling(s),$$

$$holding(do(c, s)) \equiv (\exists t)catch(t) \in c \vee holding(s) \wedge \neg(\exists t)release(t) \in c.$$

Then the following is an executable situation:

$$do(\{startFalling(1)\}, do(\{release(1)\}, S_0)).$$

The following are *not* executable situations:

$$do(\{startFalling(2)\}, do(\{release(1)\}, S_0)),$$

$$do(\{release(1)\}, do(\{startFalling(1)\}, S_0)).$$

7.9.5 Zeno's Paradox

Executable situations admit infinitely many distinct action occurrences over a finite time interval. Consider the natural action A:

$$Poss(A(t), s) \equiv t = (1 + start(s))/2,$$

with $start(S_0) = 0$. Then for any $n \geq 1$, the situation $do([A(1/2), \ldots, A(1 - 1/2^n)], S_0)$ is executable. This means that if B is another action, natural or not, with $Poss(B(t), s) \equiv t = 1$, then $B(1)$ never gets to be part of any executable situation; it never happens! This is arguably the right intuition, given the idealization of physical reality involved in

the axiomatization of A. There does not appear to be any simple way to prevent Zeno's paradox from arising in temporal axiomatizations like ours. Of course, this is not really a paradox, in the sense that such examples do not introduce any inconsistencies into the axiomatization.

7.9.6 The Natural-World Assumption

This is the sentence:

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

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

Lemma 7.9.1: *The following is a consequence of the foundational axioms and the definition of an executable situation:*

$$executable(do(c, s)) \wedge executable(do(c', s)) \wedge NWA \supset c = c'.$$

Intuitively, the above lemma tells us that natural worlds are *deterministic*: If there is an executable successor situation, it is unique. The following theorem extends Lemma 7.9.1 to *histories*: When there are only natural actions, the world evolves in a unique way, if it evolves at all.

Theorem 7.9.2: *The foundational axioms for the concurrent, temporal situation calculus together with the definition of an executable situation entail the following:*

$$executable(s) \wedge executable(s') \wedge NWA \supset s \sqsubseteq s' \vee s' \sqsubseteq s.$$

7.9.7 Least-Natural-Time Points

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

$$\begin{aligned} lntp(s, t) \stackrel{def}{=} & (\exists a)[natural(a) \wedge Poss(a, s) \wedge time(a) = t] \wedge \\ & (\forall a)[natural(a) \wedge Poss(a, s) \supset time(a) \geq t]. \end{aligned} \qquad (7.14)$$

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

Remark 7.9.3: (7.14) entails the following:

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

So, when it exists, the least-natural-time point is unique. Unfortunately, it 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 $Poss(B(x, t), s) \equiv t = start(s) + 1/x$. Another such example is when $(\forall a).natural(a) \equiv (\exists t)a = A(t)$, and $Poss(A(t), s) \equiv t > start(s)$.

The Least-Natural-Time Point Condition

In view of the possibility of "pathological" axiomatizations, for which the least-natural-time point may not exist (see comments following Remark 7.9.3), we introduce the following sentence:

$$(\forall s).(\exists a)[natural(a) \wedge Poss(a, s)] \supset (\exists t)lntp(s, t). \qquad (LNTPC)$$

Normally, it will be the responsibility of the axiomatizer to prove that his axioms entail $LNTPC$. Fortunately, under some very reasonable assumptions (Theorem 7.9.5 below), we can often prove that axiomatizations of natural worlds do entail $LNTPC$. The following theorem indicates why the $LNTPC$ is important for natural worlds.

Theorem 7.9.4: *The foundational axioms for the concurrent, temporal situation calculus together with the above definitions entail:*

$$\begin{aligned} &LNTPC \wedge NWA \supset \\ &\quad executable(do(c, s)) \equiv \{executable(s) \wedge Poss(c, s) \wedge start(s) \leq time(c) \wedge \\ &\qquad\qquad (\forall a)[a \in c \equiv Poss(a, s) \wedge lntp(s, time(a))]\}. \end{aligned}$$

This theorem informs us that for natural worlds satisfying $LNTPC$, we obtain the next executable situation from the current one by assembling into c all the possible actions occurring at the least-natural-time point of the current situation, provided this collection of natural actions is possible, and the least-natural-time point is greater than or equal to the start time of the current situation. Intuitively, this is as it should be for natural worlds. So, when $LNTPC$ holds, this theorem provides a complete characterization of the executable situations. What are some useful conditions guaranteeing $LNTPC$?

The normal settings where we wish to model natural actions involve a domain closure assumption specifying that there are just finitely many natural action types $A_1, \ldots, A_n$ with arguments determined by conditions $\phi_i(\vec{x}_i, t)$, where the ϕ_i are first-order formulas with free variables among $\vec{x}_i, t$:

$$\begin{aligned} natural(a) \equiv\ &(\exists \vec{x}_1, t)[\phi_1(\vec{x}_1, t) \wedge a = A_1(\vec{x}_1, t)] \vee \cdots \vee \\ &(\exists \vec{x}_n, t)[\phi_n(\vec{x}_n, t) \wedge a = A_n(\vec{x}_n, t)]. \end{aligned} \qquad (7.15)$$

For example, in the case of two balls B_1 and B_2 moving between two walls W_1 and W_2 that we shall treat below, the domain closure axiom is:

$$natural(a) \equiv$$
$$(\exists b, w, t)[(b = B_1 \vee b = B_2) \wedge (w = W_1 \vee w = W_2) \wedge a = bounce(b, w, t)] \vee$$
$$(\exists b_1, b_2, t)[b_1 = B_1 \wedge b_2 = B_2 \wedge a = collide(b_1, b_2, t)].$$

This says that

$$bounce(B_1, W_1, t), \quad bounce(B_2, W_1, t), \quad bounce(B_1, W_2, t),$$
$$bounce(B_2, W_2, t), \quad collide(B_1, B_2, t)$$

are all, and only, the natural actions.

The following is a very general sufficient condition for $LNTPC$ to hold when the natural actions satisfy a domain closure axiom. Its proof is a straightforward exercise in first-order theorem-proving.

Theorem 7.9.5: *The domain closure assumption (7.15) entails*

$$\bigwedge_{i=1}^{n}(\forall s)\left\{\begin{array}{l}(\exists \vec{x}_i, t)[\phi_i(\vec{x}_i, t) \wedge Poss(A_i(\vec{x}_i, t), s)] \supset \\ \qquad (\exists \vec{y}_i, t')[\phi_i(\vec{y}_i, t') \wedge Poss(A_i(\vec{y}_i, t'), s) \wedge \\ \qquad [(\forall \vec{z}_i, t'')[\phi_i(\vec{z}_i, t'') \wedge Poss(A_i(\vec{z}_i, t''), s) \supset t' \leq t'']]\end{array}\right\}$$
$$\supset LNTPC.$$

This theorem tells us that whenever a domain closure axiom of the form (7.15) holds for the natural actions, then in order to verify that $LNTPC$ is true, it is sufficient to verify that for each action type A_i:

$$(\forall s).(\exists \vec{x}_i, t)[\phi_i(\vec{x}_i, t) \wedge Poss(A_i(\vec{x}_i, t), s)] \supset$$
$$(\exists \vec{y}_i, t')[\phi_i(\vec{y}_i, t') \wedge Poss(A_i(\vec{y}_i, t'), s) \wedge$$
$$[(\forall \vec{z}_i, t'')[\phi_i(\vec{z}_i, t'') \wedge Poss(A_i(\vec{z}_i, t''), s) \supset t' \leq t'']]$$

This, in turn, says that if the action type A_i is possible at all in situation s, then there is a least time t' for which it is possible.

Theorems 7.9.4 and 7.9.5 provide the theoretical foundations for a situation calculus-based simulator for physical systems that we now describe.

7.9.8 Simulating Natural Worlds

With the above account for natural actions in hand, we can write concurrent, temporal Golog programs that simulate natural worlds. We illustrate how to do this with an example. Two perfectly elastic point balls, B_1 and B_2, of equal mass, are rolling along the x-axis on a frictionless floor, between two walls, W_1 and W_2, that are parallel to the y-axis. We expect them to bounce indefinitely between the two walls, occasionally colliding with each other. Such bounces and collisions will cause the balls' velocities to change discontinuously. Let $wallLocation(w)$ denote the distance from the y-axis of wall w.

Primitive Actions and their Precondition Axioms:

- $collide(b_1, b_2, t)$. Balls b_1 and b_2 collide at time t.
- $bounce(b, w, t)$. Ball b bounces against wall w at time t.

$$Poss(collide(b_1, b_2, t), s) \equiv vel(b_1, s) \neq vel(b_2, s) \wedge t > start(s) \wedge t = start(s) - \frac{pos(b_1,s)-pos(b_2,s)}{vel(b_1,s)-vel(b_2,s)}$$

$$Poss(bounce(b, w, t), s) \equiv vel(b, s) \neq 0 \wedge t > start(s) \wedge t = start(s) + \frac{wallLocation(w)-pos(b,s)}{vel(b,s)}$$

Fluents and Their Successor State Axioms:

- $vel(b, s)$. A functional fluent denoting the velocity of ball b in situation s.
- $pos(b, s)$. A functional fluent denoting the position (x-coordinate) of ball b in situation s.

$$pos(b, do(c, s)) = pos(b, s) + vel(b, s) * (time(c) - start(s)).$$

On hitting a wall, a ball's new velocity becomes the opposite of its old velocity. When two equal mass, perfectly elastic balls collide along a straight line, they exchange velocities according to the conservation laws of energy and momentum of Newtonian physics. However, when such a collision occurs at the same time as the balls reach a wall, we make the idealized assumption that each ball moves away from the wall with a velocity opposite to its old velocity.

$$\begin{aligned} vel(b, do(c, s)) = v \equiv\ & (\exists w, t) bounce(b, w, t) \in c \wedge v = -vel(b, s) \vee \\ & \neg(\exists w, t) bounce(b, w, t) \in c \wedge (\exists b', t)[v = vel(b', s) \wedge (collide(b, b', t) \in c \vee \\ & \qquad collide(b', b, t) \in c)] \vee \\ & v = vel(b, s) \wedge \neg(\exists b', t)[collide(b, b', t) \in c \vee collide(b', b, t) \in c] \wedge \\ & \qquad \neg(\exists w, t) bounce(b, w, t) \in c. \end{aligned}$$

Initial Situation:

$$B_1 \neq B_2, \quad W_1 \neq W_2, \quad pos(B_1, S_0) = 0, \quad pos(B_2, S_0) = 120, \quad vel(B_1, S_0) = 10, \quad vel(B_2, S_0) = -5, \quad wallLocation(W_1) = 0, \quad wallLocation(W_2) = 120.$$

A domain closure axiom for natural actions:

$$\begin{aligned} natural(a) \equiv\ & (\exists b_1, b_2, t)[b_1 = B_1 \wedge b_2 = B_2 \wedge a = collide(b_1, b_2, t)] \vee \\ & (\exists b, w, t)[(b = B_1 \vee b = B_2) \wedge (w = W_1 \vee w = W_2) \wedge a = bounce(b, w, t)]. \end{aligned}$$

The Natural-World Assumption: $(\forall a) natural(a)$.

Assume No Precondition Interaction Problem for Natural Worlds

Recall that, because of the precondition interaction problem, axiom (7.13) is not a biconditional. However, in the case of natural worlds, for which all actions obey deterministic

laws, it is reasonable to suppose that this is a biconditional:

$$Poss(c, s) \equiv coherent(c) \wedge (\forall a)[a \in c \supset Poss(a, s)].$$

This says that a collection of actions is possible iff it is coherent, and each individual action in the collection is possible. This seems to be an assumption about the accuracy with which the physics of the world has been modeled by equations of motion, in the sense that if these equations predict a co-occurrence, then this co-occurrence really happens in the physical world, so that in our situation calculus model of that world, this co-occurrence should be possible. In our bouncing balls scenario, we include this as an axiom.

Golog Procedure for Executable Situations:

Notice that the above axiomatization satisfies the antecedent conditions of Theorem 7.9.5. Hence, $LNTPC$ holds. Moreover, we are making the Natural-World Assumption that the only actions that can occur are natural actions. Hence, Theorem 7.9.4 justifies the following Golog procedure for computing the executable situation of length n. Theorem 7.9.2, assures us that if it exists, this situation is unique.

proc $executable(n)$
$n = 0? \mid$
$n > 0? \,;\ (\pi c)[\{(\forall a).a \in c \equiv Poss(a) \wedge lntp(time(a))\}? \,;\ c] \,;$
$executable(n-1)$
endProc

This completes the specification of the bouncing balls problem. The following is a concurrent, temporal Golog implementation for this axiomatization; it makes use of Prolog's setof construct for computing the set of all simple actions that are possible in a situation's least-natural-time point.

Simulation of Two Balls Bouncing between Two Walls

```
% Procedure to compute the executable situation of length N.

proc(executable(N),
      ?(N = 0) #
      pi(t, ?(N > 0 & lntp(t)) :
            pi(a, pi(c, ?(setof(a, natural(a) &
                                poss(a) & time(a,t),c)) : c))) :
      pi(n, ?(n is N - 1) : executable(n))).

%   Least-natural-time point.

lntp(S,T) :- natural(A), poss(A,S), time(A,T),
```

```
            not (natural(A1), poss(A1,S), time(A1,T1), T1 < T), !.

%  Action precondition axioms.

poss(bounce(B,W,T),S) :- vel(B,V,S), not V = 0, start(S,TS),
        pos(B,X,S), wallLocation(W,D), T is TS + (D - X)/V, T > TS.

poss(collide(B1,B2,T),S) :- vel(B1,V1,S), vel(B2,V2,S), not V1 = V2,
                            start(S,TS), pos(B1,X1,S), pos(B2,X2,S),
                            T is TS - (X1 - X2)/(V1 - V2), T > TS.

% Assume no precondition interaction problem; a concurrent action is poss-
% ible iff it is coherent, and each of its primitive actions is possible.

poss(C,S) :- coherent(C), allPoss(C,S).
allPoss([ ],S).
allPoss([A | R],S) :- poss(A,S), allPoss(R,S).

%  Successor state axioms.

pos(B,X,do(C,S)) :- pos(B,X0,S), vel(B,V0,S), start(S,TS),
                    time(C,TC), X is X0 + V0 * (TC - TS).

vel(B,V,do(C,S)) :- vel(B,V0,S),
    (  member(bounce(B,W,T),C), V is -V0 ;
       (member(collide(B,B1,T),C) ; member(collide(B1,B,T),C)),
           not member(bounce(B,W,T),C), vel(B1,V1,S), V is V1 ;
       not member(collide(B,B1,T),C),not member(collide(B1,B,T),C),
           not member(bounce(B,W,T),C), V is V0   ).

%  Initial situation.

start(s0,0.0).
vel(b1,10.0,s0).  vel(b2,-5.0,s0).  pos(b1,0.0,s0).  pos(b2,120.0,s0).
wallLocation(w1,0.0).  wallLocation(w2,120.0).

%  Natural action declarations.

natural(A) :- A = bounce(B,W,T),
              ((B = b1 ; B = b2), (W = w1 ; W = w2)) ;
              A = collide(B1,B2,T), B1 = b1, B2 = b2.

%  Restore suppressed situation arguments.
```

```
restoreSitArg(pos(B,X),S,pos(B,X,S)).
restoreSitArg(vel(B,V),S,vel(B,V,S)).
restoreSitArg(lntp(T),S,lntp(S,T)).
restoreSitArg(setof(X,Generator,Set),S,setof(X,holds(Generator,S),Set)).

%  Action occurrence times.

time(bounce(B,W,T),T).   time(collide(B1,B2,T),T).

%  Utilities.

prettyPrintSituation(S) :- makeActionList(S,Alist), nl, write(Alist), nl.

makeActionList(s0,[]).
makeActionList(do(A,S),L) :- makeActionList(S,L1), append(L1,[A],L).

simulate(T) :- do(executable(T),s0,S),
               prettyPrintSituation(S), askForMore.
askForMore :- write('More? '), read(n).
```

The following is the output received under ECLIPSE Prolog, for a run of this program for simulating the first 10 concurrent actions for this natural world. The program was run using the interpreter for concurrent, temporal Golog of Section 7.8.

```
[eclipse 2]: simulate(10).

[[collide(b1,b2,8.0)], [bounce(b2,w2,12.0)], [bounce(b1,w1,24.0),
  bounce(b2,w1,24.0), collide(b1,b2,24.0)], [bounce(b2,w2,36.0)],
 [collide(b1,b2,40.0)], [bounce(b1,w1,48.0), bounce(b2,w2,48.0)],
 [collide(b1,b2,56.0)], [bounce(b2,w2,60.0)], [bounce(b1,w1,72.0),
  bounce(b2,w1,72.0), collide(b1,b2,72.0)], [bounce(b2,w2,84.0)]]
More?  y.

no (more) solution.
```

As expected, the balls first collide at time 8.0; then B_2, reversing its direction, bounces against wall W_2 at time 12.0; then B_2 reverses its direction and the two balls meet at wall W_1, causing the concurrent occurrence, at time 24.0, of two *bounce* actions with a *collide* action; then the two balls move off to the right together, etc.

7.9.9 Animating Natural Worlds

With a minimum of additional effort, from a simulation like that above, data can be obtained for generating computer animations of natural worlds. We illustrate how to do this

using the earlier bouncing balls program. Add to this program a new natural action:

- $plot(x_1, x_2, t)$. Plot the values, at time t, of the x-coordinates of balls B_1 and B_2.

$$Poss(plot(x_1, x_2, t), s) \equiv t = clock(s) \wedge \\ x_1 = pos(B_1, s) + vel(B_1, s) \wedge x_2 = pos(B_2, s) + vel(B_2, s).$$

Here, $clock(s)$ is a new functional fluent, denoting the clock time in situation s. The clock is initialized to 0, and is advanced, by one time unit, only by a $plot$ action.

$$clock(do(c, s)) = t \equiv (\exists x_1, x_2, t')[plot(x_1, x_2, t') \in c \wedge t = t' + 1] \vee \\ clock(s) = t \wedge \neg(\exists x_1, x_2, t) plot(x_1, x_2, t) \in c.$$

Finally, the initial situation must be augmented by $clock(S_0) = 0$, $plot$ must be included among the natural actions, and we must augment the domain closure axiom for natural actions of the previous example by a disjunct $(\exists x_1, x_2, t)a = plot(x_1, x_2, t)$.

It is straightforward to verify that the antecedent conditions of Theorem 7.9.5 hold, and therefore, the $LNTPC$ is true, so by Theorem 7.9.4, we can continue using the earlier Golog program for computing the executable situations. To incorporate the $plot$ action into the Golog program for simulating the bouncing balls, we need only add to the above program the following clauses:

```
poss(plot(X1,X2,T),S) :- clock(T,S), pos(b1,P1,S), vel(b1,V1,S),
                  start(S,TS), X1 is P1 + V1*(T - TS), pos(b2,P2,S),
                  vel(b2,V2,S), X2 is P2 + V2*(T - TS).

clock(T,do(C,S)) :- member(plot(X1,X2,T1),C), T is T1 + 1 ;
                    clock(T,S), not member(plot(X1,X2,T1),C).

clock(0.0,s0).
natural(plot(X1,X2,T)).
restoreSitArg(clock(T),S,clock(T,S)).
time(plot(X1,X2,T),T).
```

The following is the result of executing this plot program for 40 steps. As before, it was run using the interpreter for concurrent, temporal Golog of Section 7.8.

```
[eclipse 2]: simulate(40).

[[plot(0.0,120.0,0.0)], [plot(10.0,115.0,1.0)], [plot(20.0,110.0,2.0)],
 [plot(30.0,105.0,3.0)], [plot(40.0,100.0,4.0)], [plot(50.0,95.0,5.0)],
 [plot(60.0,90.0,6.0)], [plot(70.0,85.0,7.0)], [collide(b1,b2,8.0),
  plot(80.0,80.0,8.0)], [plot(75.0,90.0,9.0)], [plot(70.0,100.0,10.0)],
 [plot(65.0,110.0,11.0)], [bounce(b2,w2,12.0), plot(60.0,120.0,12.0)],
```

```
 [plot(55.0,110.0,13.0)], [plot(50.0,100.0,14.0)], [plot(45.0,90.0,15.0)],
 [plot(40.0,80.0,16.0)], [plot(35.0,70.0,17.0)], [plot(30.0,60.0,18.0)],
 [plot(25.0,50.0,19.0)], [plot(20.0,40.0,20.0)], [plot(15.0,30.0,21.0)],
 [plot(10.0,20.0,22.0)], [plot(5.0,10.0,23.0)], [bounce(b1,w1,24.0),
  bounce(b2,w1,24.0), collide(b1,b2,24.0), plot(0.0,0.0,24.0)],
 [plot(5.0,10.0,25.0)], [plot(10.0,20.0,9.0)], [plot(30.0,60.0,30.0)],
 [plot(35.0,70.0,31.0)], [plot(40.0,80.0,32.0)], [plot(45.0,90.0,33.0)],
 [plot(50.0,100.0,34.0)], [plot(55.0,110.0,35.0)], [bounce(b2,w2,36.0),
  plot(60.0,120.0,36.0)], [plot(65.0,110.0,37.0)],[plot(70.0,100.0,38.0)],
 [plot(75.0, 90.0, 39.0)]]
More?  y.

no (more) solution.
```

It is easy to imagine how this kind of output could be passed to a graphics routine to generate a computer animation of the bouncing balls scenario.

7.10 Exercises

1. (a) Show that the foundational axioms for the concurrent situation calculus (both temporal and non-temporal versions) entail that

   $$(\forall s)\neg Poss(\{\}, s).$$

   In other words, the empty concurrent action is never possible.

   (b) Show that the foundational axioms for the concurrent temporal situation calculus entail that

   $$(\forall c, s).Poss(c, s) \supset (\forall a, a').a \in c \wedge a' \in c \supset time(a) = time(a').$$

2. Using the concurrent, non-temporal situation calculus, axiomatize the following table lifting problem. A table on which there is a cup of coffee is resting on the floor with its top horizontal. There are two agents; one can lift the left side of the table, the other the right side. The coffee will not spill iff no lift action occurs, or both agents start to lift their ends of the table together, and subsequently terminate their lifting actions together. Use the actions $startLiftL$, $startLiftR$, $endLiftL$, $endLiftR$ and the fluents $liftingL$, $liftingR$, $spilled$, and give action precondition axioms for the actions and successor state axioms for the fluents. Using these, and assuming initially the coffee is not spilled, prove that after executing the action sequence

   $$[\{startLiftL,\ startLiftR\}, \{endLiftL,\ endLiftR\}]$$

   the coffee will not be spilled, but after the sequence

$$[\{startLiftL,\ startLiftR\}, \{endLiftL\}],$$

it will be spilled. Why would an interleaving account of concurrency not be appropriate here?

3. For the concurrent, temporal situation calculus, prove:

$$executable(do(c, s)) \equiv executable(s) \wedge Poss(c, s') \wedge start(s') \leq time(c).$$

4. Give an interleaving axiomatization for the two duelists example by introducing a process fluent $bulletDirectedAt(p)$, initiated by action $shoot(p', p)$ and terminated by $strikes(p)$.
5. Consider the sequential situation calculus in which actions have situation-dependent durations, possibly zero, as well as initiation times.
   (a) Propose suitable foundational axioms for this setting, as well as an appropriate notion of executable situation.
   (b) Using these, implement a Golog interpreter for this setting.
   (c) Modify the coffee delivery program of Section 7.4.1 for this version of Golog and run it. Assume that the $pickupCoffee$ and $giveCoffee$ actions continue to have zero duration.
6. When the time line consists of the integers, prove that the *LNTPC* is always true.
7. Consider the setting of two balls falling vertically onto a horizontal floor under the influence of gravity. The balls lie in different vertical planes, so they cannot collide. On hitting the floor, a ball bounces, with a rebound coefficient $r > 0$, meaning that it's upward velocity becomes r times its previous downward velocity. Each ball has its own value of r.
   (a) Implement a simulation for this setting.
   (b) With the help of this simulation, illustrate Zeno's paradox, by choosing suitable initial heights, velocities and rebound coefficients for the two balls.
   (c) Finally, modify your simulation to provide a graphical plot of the falling balls.

7.11 Bibliographic Remarks

An early, and very influential account of time and actions was due to James Allen [2]. The door latch problem of Section 7.2.1 is taken from one of his papers [3]; the other papers in the collection containing the latter give an excellent overview of the recent status of Allen's theory. Another early, and quite sophisticated, fully axiomatic account of time and actions was due to McDermott [142]. Here, McDermott addresses the frame problem, continuous

time and change, and branching futures quite similar to those of the situation calculus. Chapter 5 of Davis's book [36] describes many of the details of this approach.

Perhaps the earliest treatment, in the situation calculus, of concurrency and actions with durations was by Gelfond, Lifschitz, and Rabinov [61]. By basing it on the language $\mathcal{A}$ of Gelfond and Lifschitz [59], Baral and Gelfond [13] provide a semantic account of concurrency that, although not formulated in the situation calculus, has many similarities with ours. The principal difference is that Baral and Gelfond focus exclusively on concurrency, so their ontology does not include time or natural actions. More recent treatments for concurrency in the situation calculus, again, ignoring time, are by Schubert [192] and Lin and Shoham [127]. See also Pinto [161]. Pelavin [156] addresses the formalization of concurrent actions by extending the ontology of Allen's linear time logic [3] to include histories to represent branching futures, and suitable modal operators semantically characterized with respect to these histories. This gives a rich representation for time and concurrency, somewhat like that of the situation calculus, but at the expense of a rather complicated logic.

The idea of decomposing actions with durations into two instantaneous start and end actions, together with a fluent representing a process, was proposed for the situation calculus by Pinto [159] and Ternovskaia [210]. The precondition interaction problem for concurrent actions is discussed by Pelavin [156] and by Pinto [159]. For an extensive discussion of Zeno's paradox, see Davis [37].

The material on sequential, temporal Golog and the coffee delivery program is taken from Reiter [177], and that on concurrency and natural actions from Reiter [176], which in turn relies heavily on earlier work by Pinto [159]. See also Pinto [162]. For the use of time in the situation calculus for the simulation of physical and mechanical systems, see Kelley [95]. There are several other approaches to modeling natural actions, mostly based upon linear temporal logics. Examples are the work by Miller and Shanahan [194, 144, 145], and Van Belleghem, Denecker, and De Schreye [17], both using extended versions of the event calculus of Kowalski and Sergot [98]. Closely related ideas about representing physical processes were proposed earlier by Sandewall [186]. See also Herrmann and Thielscher [84]. Section 7.9.9 indicated how Golog might be used for computer graphics applications. For a much more interesting exploration of this idea in computer animation that involves sharks, a merman, Velociraptors and a Tyrannosaurus rex, see Funge [55, 56, 57, 54].

A general framework for execution monitor for an agent executing a Golog program on-line is described in (De Giacomo, Reiter, and Soutchanski [68]). This monitor is used by Soutchanski [205], who presents an implementation for the coffee-delivery program of Section 7.4.1 in which the robot monitors its execution of its delivery schedule by sensing and recording the current time in ECLIPSE's temporal constraint store, along the lines discussed in Section 7.4.3.