9

STEP : a Scripting Language for Embodied Agents

9.1 MOTIVATION

Embodied agents are autonomous agents which have bodies by which the agents can perceive their world directly through sensors and act on the world directly through effectors. Embodied agents whose experienced worlds are located in real environments, are usually called *cognitive robots*. *Web agents* are embodied agents whose experienced worlds are the Web; typically, they act and collaborate in networked virtual environments. In addition, *3D web agents* are embodied agents whose 3D avatars can interact with each other or with users via Web browsers[Huang et al., 2000].

Embodied agents usually interact with users or each other via multimodal communicative acts, which can be non-verbal or verbal. Gestures, postures and facial expressions are typical non-verbal communicative acts. In general, specifying communicative acts for embodied agents is not easy; they often require a lot of geometrical data and detailed movement equations, say, for the specification of gestures.

In [Huang et al., 2002b] we propose the scripting language STEP (Scripting Technology for Embodied Persona), in particular for communicative acts of embodied agents. At present, we focus on aspects of the specification and modeling of gestures and postures for 3D web agents. However, STEP can be extended for other communicative acts, like facial expressions, speech, and other types of embodied agents, like cognitive robots. Scripting languages are to a certain extent simplified languages which ease the task of computation and reasoning. One of the main advantages of using scripting languages is that

the specification of communicative acts can be separated from the programs which specify the agent architecture and mental state reasoning. Thus, changing the specification of communicative acts doesn't require to re-program the agent.

The avatars of 3D web agents are built in the Virtual Reality Modeling Language (VRML). These avatars are usually humanoid-like ones. We have implemented the proposed scripting language for H-anim based humanoids in the distributed logic programming language DLP.

In this chapter, we discuss how STEP can be used for embodied agents. STEP introduces a Prolog-like syntax, which makes it compatible with most standard logic programming languages, whereas the formal semantics of STEP is based on dynamic logic [Harel, 1984]. Thus, STEP has a solid semantic foundation, in spite of a rich number of variants of the compositional operators and interaction facilities on worlds.

9.2 PRINCIPLES

We design the scripting language primarily for the specification of communicative acts for embodied agents. Namely, we separate external-oriented communicative acts from internal changes of the mental states of embodied agents because the former involves only geometrical changes of the body objects and the natural transition of the actions, whereas the latter involves more complicated computation and reasoning. Of course, a question is: why not use the same scripting language for both external gestures and internal agent specification? Our answer is: the scripting language is designed to be a simplified, user-friendly specification language for embodied agents, whereas the formalization of intelligent agents requires a powerful specification and programming language. It's not our intention to design a scripting language with fully-functional computation facilities, like other programming languages as Java, Prolog or DLP. A scripting language should be interoperable with a fully powered agent implementation language, but offer a rather easy way for authoring. Although communicative acts are the result of the internal reasoning of embodied agents, they do not need the same expressiveness of a general programming language. However, we do require that a scripting language should be able to interact with mental states of embodied agents in some ways, which will be discussed in more detail later.

We consider the following design principles for a scripting language.

Principle 1: Convenience As mentioned, the specification of communicative acts, like gestures and facial expressions usually involve a lot of geometrical data, like using ROUTE statements in VRML, or movement equations, like those in computer graphics. A scripting language should hide those geometrical difficulties, so that non-professional authors can use it in a natural way.

For example, suppose that authors want to specify that an agent turns his left arm forward slowly. It can be specified like this:

```
turn(Agent, left_arm, front, slow)
```

It should not be necessary to specify it as follows, which requires knowledge of a coordination system, rotation axis, etc.

```
turn(Agent, left_arm, rotation(1,0,0,1.57), 3)
```

One of the implications of this principle is that embodied agents should be aware of their context. Namely, they should be able to understand what certain indications mean, like the directions 'left' and 'right', or the body parts 'left arm', etc.

Principle 2: Compositional Semantics Specification of composite actions, based on existing components. For example, an action of an agent which turns his arms forward slowly, can be defined in terms of two primitive actions: turn-left-arm and turn-right-arm, like:

```
par([turn(Agent, left_arm, front, slow),
    turn(Agent, right_arm, front, slow)])
```

Typical composite operators for actions are sequence action *seq* , parallel action *par*, repeat action *repeat*, which are used in dynamic logics [Harel, 1984].

Principle 3: Re-definability Scripting actions (i.e.,composite actions), can be defined in terms of other defined actions explicitly. Namely, the scripting language should be a rule-based specification system. Scripting actions are defined with their own names. These defined actions can be re-used for other scripting actions. For example, if we have defined two scripting actions *run* and *kick*, then a new action *run_then_kick* can be defined in terms of *run* and *kick* :

```
run_then_kick(Agent)=
   seq([script(run(Agent)), script(kick(Agent))]).
```

which can be specified in a Prolog-like syntax:

```
script(run_then_kick(Agent), Action):-
   Action = seq([script(run(Agent)),script(kick(Agent))]).
```

Principle 4: Parametrization Scripting actions can be adapted to be other actions. Namely, actions can be specified in terms of how these actions cause changes over time to each individual *degree of freedom*, which is proposed by Perlin and Goldberg in [Perlin and Goldberg, 1996]. For example, suppose that we define a scripting action *run*: we know that running can be done at different paces. It can be a 'fast-run' or 'slow-run'. We should not define all

of the run actions for particular paces. We can define the action 'run' with respect to a degree of freedom 'tempo'. Changing the tempo for a generic run action should be enough to achieve a run action with different paces. Another method of parametrization is to introduce variables or parameters in the names of scripting actions, which allows for a similar action with different values. That is one of the reasons why we introduce Prolog-like syntax in STEP.

Principle 5: Interaction Scripting actions should be able to interact with, more exactly, perceive the world, including embodied agents' mental states, to decide whether or not it should continue the current action, or change to other actions, or stop the current action. This kind of interaction modes can be achieved by the introduction of high-level interaction operators, as defined in dynamic logic. The operator 'test' and the operator 'conditional' are useful for the interaction between the actions and the states.

9.3 SCRIPTING LANGUAGE STEP

In this section, we discuss the general aspects of the scripting language STEP.

9.3.1 Reference Systems

Direction Reference The reference system in STEP is based on the H-anim specification: namely, the initial humanoid position should be modeled in a standing position, facing in the +Z direction with +Y up and +X to the humanoid's left. The origin $\langle 0, 0, 0 \rangle$ is located at ground level, between the humanoid's feet. The arms should be straight and parallel to the sides of the body with the palms of the hands facing inwards towards the thighs.

Based on the standard pose of the humanoid, we can define the direction reference system as sketched in figure 9.1. The direction reference system is based on these three dimensions: front vs. back which corresponds to the Z-axis, up vs. down which corresponds to the Y-axis, and left vs. right which corresponds to the X-axis. Based on these three dimensions, we can introduce a more natural-language-like direction reference scheme, say, turning left-arm to 'front-up', is to turn the left-arm such that the front-end of the arm will point to the up front direction. Figure 9.2 shows several combinations of directions based on these three dimensions for the left-arm. The direction references for other body parts are similar. These combinations are designed for convenience and are discussed in Section 9.2. However, they are in general not sufficient for more complex applications. To solve this kind of problem, we introduce interpolations with respect to the mentioned direction references. For instance, the direction 'left_front2' is referred to as one which is located

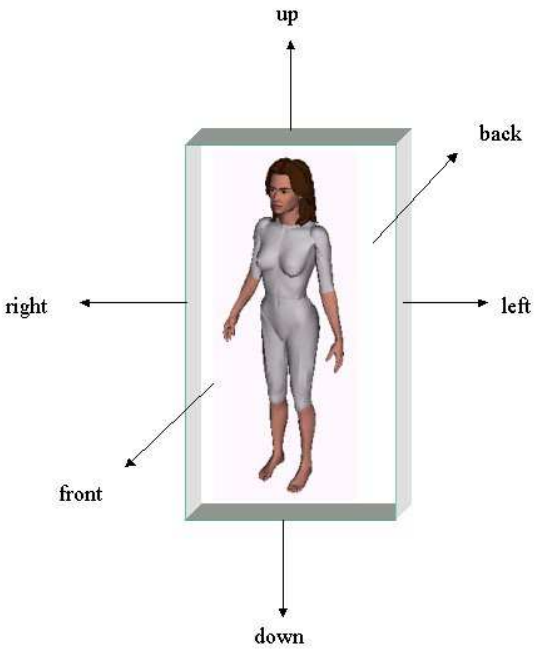

Fig. 9.1 Direction Reference for Humanoid

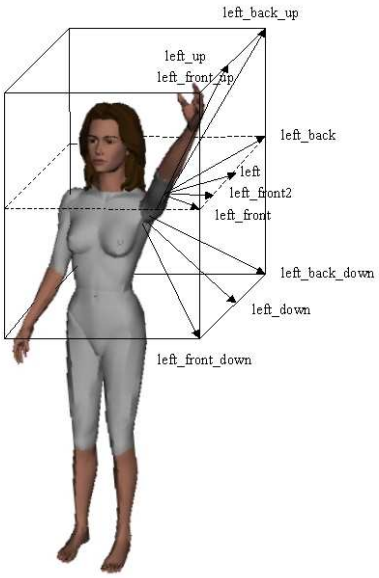

Fig. 9.2 Combination of the Directions for Left Arm

between 'left_front' and 'left', which is shown in Figure 9.2. Natural-language-like references are convenient for authors to specify scripting actions, since they do not require the author to have a detailed knowledge of reference systems in VRML. Moreover, the proposed scripting language also supports the orginal VRML reference system, which is useful for experienced authors. Directions can also be specified to be a four-place tuple $\langle X, Y, Z, R\rangle$, say, $rotation(1, 0, 0, 1.57)$.

Body Reference An H-anim specification contains a set of *Joint nodes* that are arranged to form a hierarchy. Each Joint node can contain other Joint nodes and may also contain a *Segment node* which describes the body part associated with that joint. Each Segment can also have a number of Site nodes, which define locations relative to the segment. Sites can be used for attaching accessories, like hat, clothing and jewelry. In addition, they can be used to define eye points and viewpoint locations. Each Segment node can have a number of *Displacer nodes*, that specify which vertices within the segment correspond to a particular feature or configuration of vertices.

Figure 9.3 shows several typical joints of humanoids. Therefore, turning body parts of humanoids implies the setting of the relevant joint's rotation. Body moving means the setting of the HumanoidRoot to a new position. For instance, the action 'turning the left-arm to the front slowly' is specified as:

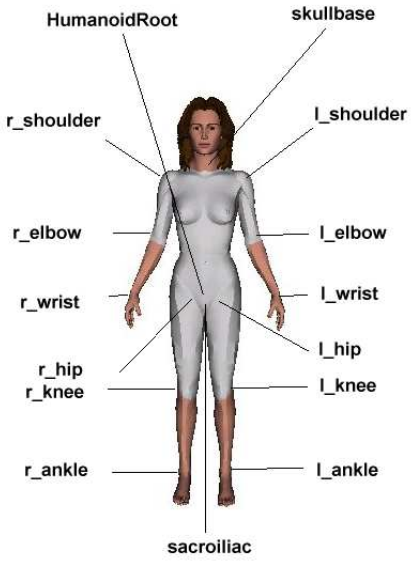

Fig. 9.3 Typical Joints for Humanoid

```
turn(Agent, l_shoulder, front, slow)
```

Time Reference STEP has the same time reference system as that in VRML. For example, the action *turning the left arm to the front in 2 seconds* can be specified as:

```
turn(Agent, l_shoulder, front, time(2, second))
```

This kind of explicit specification of duration in scripting actions does not satisfy the parametrization principle. Therefore, we introduce a more flexible time reference system based on the notions of beat and tempo. A *beat* is a time interval for body movements, whereas the *tempo* is the number of beats per minute. By default, the tempo is set to 60. Namely, a beat corresponds to a second by default. However, the tempo can be changed. Moreover, we can define different speeds for body movements, say, the speed 'fast' can be defined as one beat, whereas the speed 'slow' can be defined as three beats.

9.3.2 Primitive Actions and Composite Operators

Turn and move are the two main primitive actions for body movements. Turn actions specify the change of the rotations of the body parts or the whole body over time, whereas move actions specify the change of the positions of the body parts or the whole body over time. A turn action is defined as follows:

```
turn(Agent,BodyPart,Direction,Duration)
```

where `Direction` can be a natural-language-like direction like 'front' or a rotation value like 'rotation(1,0,0,3.14)', `Duration` can be a speed name like 'fast' or an explicit time specification, like 'time(2,second)'.

A move action is defined as:

```
move(Agent,BodyPart,Direction,Duration)
```

where `Direction` can be a natural-language-like direction, like 'front', a position value like 'position(1,0,10)', or an increment value like 'increment(1,0,0)'.

Here are typical composite operators for scripting actions:

- Sequence operator 'seq': the action `seq([`$Action_1$`, ...,`$Action_n$`])` denotes a composite action in which $Action_1$, ...,and $Action_n$ are executed sequentially, like:

  ```
  seq([turn(agent,l_shoulder,front,fast),
  turn(agent,r_shoulder,front,fast)])
  ```

- Parallel operator 'par': the action `par([`$Action_1$`, ...,`$Action_n$`])` denotes a composite action in which $Action_1$, ...,and $Action_n$ are executed simultaneously.

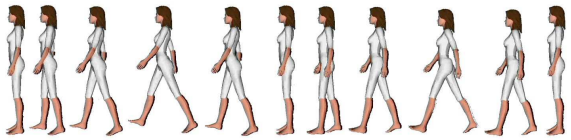

Fig. 9.4 Walk

- non-deterministic choice operator 'choice': the action `choice([`$Action_1$`, ...,`$Action_n$`])` denotes a composite action in which one of the $Action_1$, ...,and $Action_n$ is executed.
- repeat operator 'repeat': the action `repeat(Action, T)` denotes a composite action in which the *Action* is repeated T times.

9.3.3 High-level Interaction Operators

When using high-level interaction operators, scripting actions can directly interact with internal states of embodies agents or with external states of worlds. These interaction operators are based on a meta language which is used to build embodied agents, say, in the distributed logic programming language DLP. In the following, we use lower case Greek letters ϕ, ψ, χ to denote formulas in the meta language. Examples of several higher-level interaction operators:

- test: `test(`ϕ`)`, check the state ϕ. If ϕ holds then skip, otherwise fail.
- execution: `do(`ϕ`)`, make the state ϕ true, i.e. execute ϕ in the meta language.
- conditional: `if_then_else(`ϕ`,`$action_1$`,`$action_2$`)`.
- until: `until(action,`ϕ`)`: take action until ϕ holds.

We have implemented the scripting language STEP in the distributed logic programming language DLP.

9.4 EXAMPLES

9.4.1 Walk and its Variants

A walking posture can be simply expressed as a movement which exchanges the following two main poses: a pose in which the left-arm/right-leg move forward while the right-arm/left-leg move backward, and a pose in which the right-arm/left-leg move forward while the left-arm/right-leg move backward.

The main poses and their linear interpolations are shown in Figure 9.4. The walk action can be described in the scripting language as follows:

```
script(walk_pose(Agent), Action):-
   Action = seq([par([
       turn(Agent,r_shoulder,back_down2,fast),
       turn(Agent,r_hip,front_down2,fast),
       turn(Agent,l_shoulder,front_down2,fast),
       turn(Agent,l_hip,back_down2,fast)]),
    par([turn(Agent,l_shoulder,back_down2,fast),
       turn(Agent,l_hip,front_down2,fast),
       turn(Agent,r_shoulder,front_down2,fast),
       turn(Agent,r_hip,back_down2,fast)])]).
```

Thus, a walk step can be described to be as a parallel action which consists of the walking posture and the moving action (i.e., changing position) as follows:

```
script(walk_forward_step(Agent),Action):-
  Action= par([script_action(walk_pose(Agent)),
           move(Agent,front,fast)]).
```

The step length can be a concrete value. For example, for the step length with 0.7 meter, it can be defined as follows:

```
script(walk_forward_step07(Agent),Action):-
   Action= par([script_action(walk_pose(Agent)),
          move(Agent,increment(0.0,0.0,0.7),fast)]).
```

Alternatively, the step length can also be a variable like:

```
script(walk_forward_step0(Agent,StepLength),Action):-
  Action = par([script_action(walk_pose(Agent)),
      move(Agent,increment(0.0,0.0,StepLength),fast)]).
```

Therefore, the walking forward N steps with the $StepLength$ can be defined as follows:

```
script(walk_forward(Agent,StepLength,N),Action):-
   Action = repeat(script_action(
      walk_forward_step0(Agent,StepLength)),N).
```

As mentioned above, the animations of the walk based on these definitions are just simplified and approximated ones. As analysed in [Faure, 1997], a realistic animation of the walk motions of human figure involves a lot of computations which rely on a robust simulator where forward and inverse kinematics are combined with automatic collision detection and response. We do not want to use the scripting language to achieve a fully realistic animation of

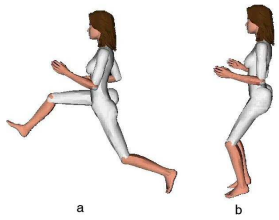

Fig. 9.5 Poses of Run

the walk action, because they are seldom necessary for most web applications. However, we would like to point out that there does exist the possibility to accommodate some inverse kinematics to improve the realism by using the scripting language.

9.4.2 Run and its Deformation

The action 'run' is similar to 'walk', however, with a bigger wave of the lower-arms and the lower-legs, which is shown in Figure 9.5a. As we can see from the figure, the left lower-arm points to the direction 'front-up' when the left upper-arm points to the direction 'front_down2' during the run. Consider the hierarchies of the body parts, we should not use the primitive action $turn(Agent, l\_elbow, front\_up, fast)$ but the primitive action $turn(Agent, l\_elbow, front, fast)$, for the direction of the left lower-arm should be defined with respect to the default direction of its parent body part, i.e., the left arm (more exactly, the joint l_shoulder). That kind of re-direction would not cause big difficulties for authoring, for the correct direction can be obtained by reducing the directions of its parent body parts to be the default ones. As we can see in Figure 9.5b, the lower-arm actually points to the direction 'front'.

The action 'run_pose' can be simply defined as an action which starts with a basic run pose as shown in Figure 9.5b and then repeat the action 'walk_pose' for N times as follows:

```
script(basic_run_pose(Agent), Action):-
 Action=par([turn(Agent,r_elbow,front,fast),
      turn(Agent, l_elbow, front, fast),
      turn(Agent, l_hip, front_down2, fast),
      turn(Agent, r_hip, front_down2, fast),
      turn(Agent, l_knee, back_down, fast),
      turn(Agent, r_knee, back_down, fast)]).

script(run_pose(Agent,N),Action):-
    Action = seq([script_action(basic_run_pose(Agent)),
      repeat(script_action(walk_pose(Agent)),N)]).
```

Therefore, the action running forward N steps with the *StepLength* can be defined in the scripting language as follows:

```
script(run(Agent, StepLength,N),Action):-
 Action=seq([script_action(basic_run_pose(Agent)),
   script_action(walk_forward(Agent,StepLength,N))]).
```

Actually, the action 'run' may have a lot of variants. For instances, the lower-arm may point to different directions. They would not necessarily point to the direction 'front'. Therefore, we may define the action 'run' with respect to certain degrees of freedom. Here is an example to define a degree of freedom with respect to the angle of the lower arms to achieve the deformation.

```
script(basic_run_pose_elbow(Agent,Elbow_Angle),Action):-
   Action = par([
     turn(Agent,r_elbow,rotation(1,0,0,Elbow_Angle),fast),
     turn(Agent,l_elbow,rotation(1,0,0,Elbow_Angle),fast),
     turn(Agent,l_hip,front_down2,fast),
     turn(Agent,r_hip,front_down2,fast),
     turn(Agent,l_knee,back_down,fast),
     turn(Agent,r_knee,back_down,fast)]).

script(run_e(Agent,StepLength,N,Elbow_Angle),Action):-
   Action = seq([script_action(
      basic_run_pose_elbow(Agent,Elbow_Angle)),
      script_action(walk_forward(Agent, StepLength, N))]).
```

9.4.3 Tai Chi

In this subsection, we will discuss an application example, the development of an instructional VR for Tai Chi, illustrating how our approach allows for the creation of reusable libraries of behavior patterns.

A Tai Chi exercise is a sequence of exercise stages. Each stage consists of a group of postures. These postures can be defined in terms of their body part movements. Figure 9.6 shows several typical postures of Tai Chi. For instance, the beginning-posture and the push-down-posture can be defined as follows:

```
script(taichi(Agent,beginning_posture),Action):-
    Action =seq([
              turn(Agent,l_hip,side1_down,fast),
              turn(Agent,r_hip,side1_down,fast),
            par([turn(Agent,l_shoulder,front,slow),
        turn(Agent,r_shoulder,front,slow)])]).

script(taichi(Agent,push_down_posture),Action) :-
```

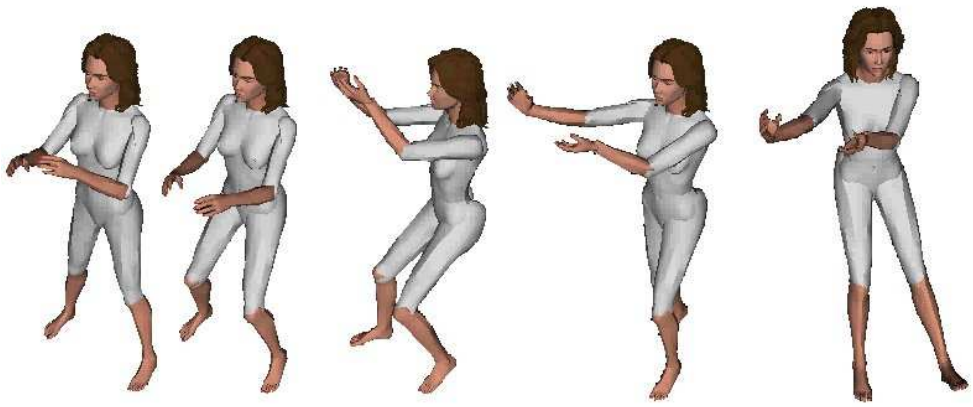

Fig. 9.6 Tai Chi

```
   Action =seq([
       par([turn(Agent,l_shoulder,front_down,slow),
            turn(Agent,r_shoulder,front_down,slow),
turn(Agent,l_elbow,front_right2,slow),
turn(Agent,r_elbow,front_left2,slow)]),
 par([turn(Agent,l_hip,left_front_down,slow),
      turn(Agent,r_hip,right_front_down,slow),
turn(Agent,l_elbow,right_front_down,slow),
turn(Agent,r_shoulder,front_down2,slow),
turn(Agent,l_knee,back2_down,slow),
turn(Agent,r_knee,back2_down,slow)])]).
```

Those defined posture can be used to define the stages like this:

```
script(taichi(Agent, stage1), Action) :-
   Action =seq([
         script(taichi(Agent,beginnin_posture),
         script(taichi(Agent,push_down_posture),
         ......
            ]).
```

Furthermore, those scripting actions can be used to define more complex Tai Chi exercise as follows:

```
script(taichi(Agent), Action) :-
    Action = seq([
        do(display('Taichi exercise ...~n')),
  script(taichi(Agent, stage1)),
```

```
        script(taichi(Agent, stage2))
        ......
               ]).
```

Combined with the interaction operators, like do-operator, with built-in predicates in mete-language, the instructional actions become more attractive. Moveover, the agent names in those scripting actions are defined as variables. These varibales can be instantiated with different agent names in different applications. Thus, the same scripting actions can be re-used for different avatars for different applications.

9.4.4 Interaction with Other Agents

Just consider a situation in which two agents have to move a 'heavy' table together. This scripting action 'moving-heavy-table' can be designed to be ones which consist of the following several steps (i.e., sub-actions): first walk to the table, then hold the table, and finally move the table around. Using the scripting language, it is not difficult to define those sub-actions, they can be done just like the other examples above, like walk and run. A solution to define the action 'moving-heavy-table' which involves multiple agents can be as follows [1]:

```
script(move_heavy_table(Agent1,Agent2,Table,
  NewPos), Action):-
       Action = seq([par([
            script(walk_to(Agent1,Table)),
            script(walk_to(Agent2,Table))]),
       par([script(hold(Agent1,Table,left)),
            script(hold(Agent2,Table,right))]),
       par([move(Agent1,NewPosition,slow),
            move(Agent2,NewPosition,slow),
            do(object_move(Table,NewPos,slow))
                  ])]).
```

The solution above is not a good solution if we consider this action is a co-operating action between two agents. Namely, this kind of actions should not be achieved by this kind of pre-defined actions but by certain communicative/negotiation procedures. Hence, the scripting action should be considered as an action which involves only the agent itself but not other agents. Anything the agent need from others can only be achieved via its communicative actions with others or wait until certain conditions meet. Therefore, the cooperating action 'moving-heavy-table' should be defined by the following procedure, first the agent walks to the table and holds one of the end of the

[1] We omit the details about the definitions of the actions like walk_to, hold, etc.

table[2], next, wait until the partner holds another end of the table, then moves the table. It can be defined as follows:

```
script(move_heavy_table(Agent,Partner,
    Table, NewPos), Action):-
       Action=seq([script(walk_to(Agent,Table)),
         if_then_else(not(hold(Partner,Table,left)),
                    script(hold(Agent,Table,left)),
                    script(hold(Agent,Table,right))),
       until(wait,hold(Partner,Table,_)),
       par([move(Agent,NewPos,slow),
            do(object_move(Table,NewPos,slow))]
                 )]).
```

9.5 XSTEP: THE XML-ENCODED STEP

We are also developing XSTEP, an XML encoding for STEP. We use *seq* and *par* tags as found in SMIL[3], as well as action tags with appropriate attributes for speed, direction and body parts involved. As an example, look at the XSTEP specification of the *walk* action.

```
<action name="walk(Agent)">
      <seq>
        <par>
          <turn actor="Agent" part="r_shoulder">
            <dir value="back_down2"/>
            <speed value="fast"/>
          </turn>
          <turn actor="Agent" part="r_hip">
            <dir value="front_down2"/>
            <speed value="fast"/>
          </turn>
          <turn actor="Agent" part="l_shoulder">
            <speed value="fast"/>
            <dir value="front_down2"/>
          </turn>
          <turn actor="Agent" part="l_hip">
            <dir value="back_down2"/>
            <speed value="fast"/>
          </turn>
```

[2]Here we consider only a simplified scenario. We do not consider the deadlock problem here, like the that in the *philosopher dinner problem*.
[3]http://www.w3.org/AudioVideo

```
      </par>
       ......
    </seq>
  </action>
```

Similar as with the specification of dialog phrases, such a specification is translated into the corresponding DLP code, which is loaded with the scene it belongs to. For XSTEP we have developed an XSLT stylesheet, using the Saxon[4] package, that transforms an XSTEP specification into DLP. We plan to incorporate XML-processing capabilities in DLP, so that such specifications can be loaded dynamically.

9.6 IMPLEMENTATION ISSUES

We have implemented the scripting language STEP in the distributed logic programming language DLP. In this section, we discuss several implementation and performance issues. First we will discuss the module architectures of STEP . Scripting actions are defined as a sequence or parallel set of actions. One of the main issues is how to implement parallel actions with a satisfying performance. Another issue is which interpolation method should be used to achieve smooth transitions from an initial state to a final state.

9.6.1 STEP Components

STEP is designed for multiple purpose use. It serves as an animation/action engine, which can be embodied as a component in embodied agents, or can also be located at the controlling component at XSTEP, the XML-based markup language.

STEP consists of the following components:

- **Action library**: The action library is the collections of the scripting actions, which can be of user defined or of system built-in.
- **STEP ontology**: The STEP ontology component defines the semantic meanings of the STEP reference systems. So-called *Ontology* is a description of the concepts or bodies of knowledge understood by a particular community and the relationships between those concepts. The STEP body ontological specification is based on H-anim specification. The STEP ontology component also defines the semantic meaning of the direction reference system. For instance, the semantic interpretation of the direction 'front' can be defined in the ontology component as follows:

[4]http://saxon.sourceforge.com

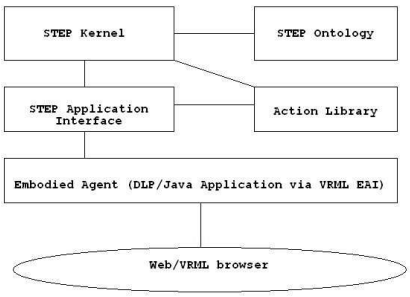

Fig. 9.7 STEP and its interface with embodied agents

```
rotationParameter(_,front,rotation(1,0,0,-1.57)).
```

Namely, turning a body part to 'front' is equal to setting its rotation to (1, 0, 0, -1.57). Separated ontology specification component would make STEP more convenient for the extension/change of its ontology. It is also a solution to the maintenance of the interoperability of the scripting langauge over the Web.

- **STEP Kernel**: The STEP kernel is the central controlling component of STEP . It translates scripting actions into executable VRML/X3D EAI commands based on the semantics of the action operators and the ontological definitions of the reference terms. The STEP kernel and the STEP ontology component are application-independent. The two components together is called the *STEP engine*.

- **STEP application interface**: The STEP application interface component offers the interface operators for users/applications. The scripting actions can be called by using these application interface operators. They can be java-applet-based, or java-script-based, or XML-based.

The avatars of VRML/X3D-based embodied agents are displayed in a VRML/X3D browser. These embodied agents are usually designed as java applets which are embodied in the browser. They interact with virtual environments via VRML/X3D External Application Interface (EAI). STEP is also designed as the part of the java applets, which can be called by embodied agents via the step interface component. STEP module architecture and its interface with embodied agents is shown in Figure 9.7.

9.6.2 Parallelism and Synchronization

How to implement parallel actions with a satisfying performance is an important issue for a scripting language. A naive solution is to create a new thread

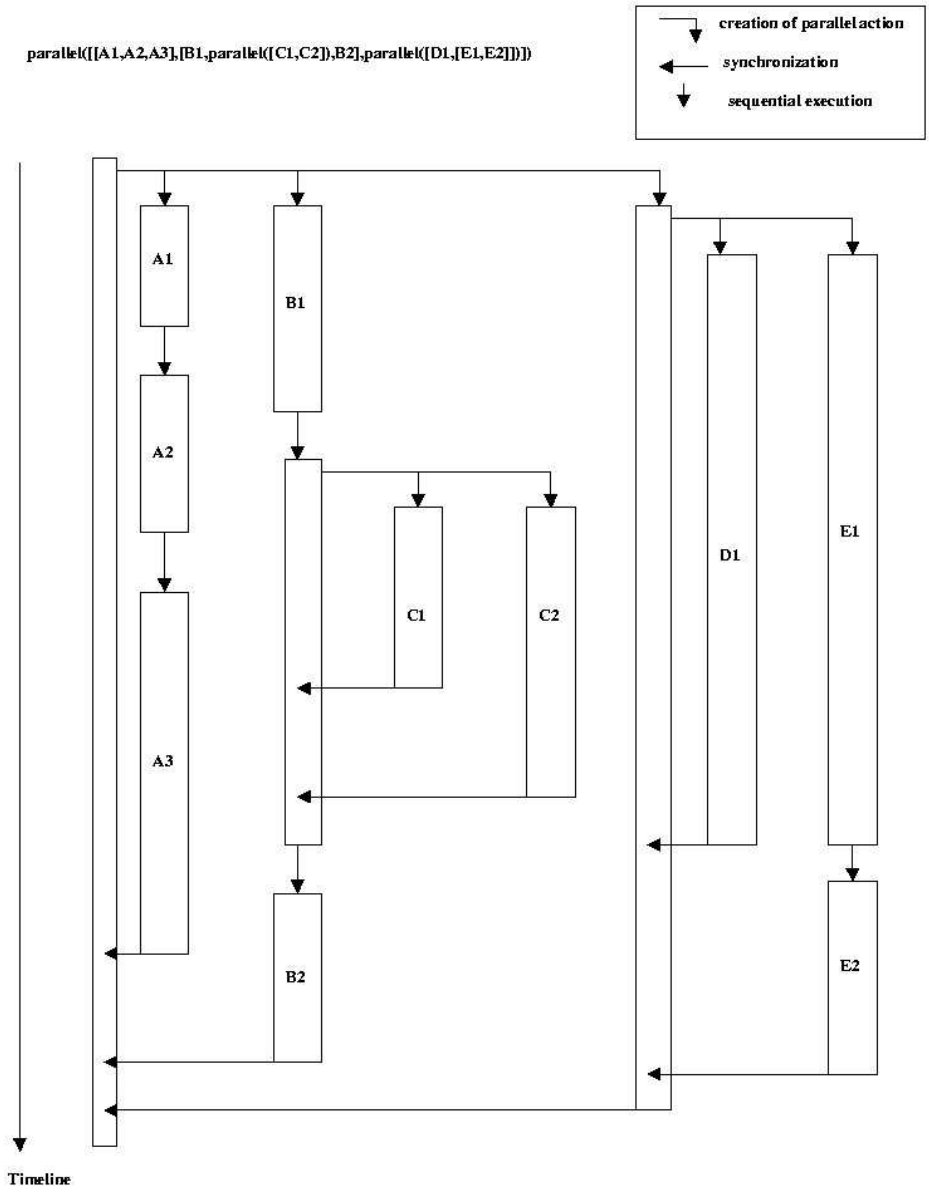

Fig. 9.8 Processing Parallel Actions

for each component action of parallel actions. However, this naive solution will result in a serious performance problem, because a lot of threads will be created and killed frequently. Our solution to handle this problem is to create only a limited number of threads, which are called *parallel threads*. The system assigns component actions of parallel actions to one of the parallel threads by a certain scheduling procedure. We have to consider the following issues with respect to the scheduling procedure: the correctness of the scheduling procedure and its performance. The former implies that the resulting action should be semantically correct. It should be at least order-preserving. Namely, an intended late action should not be executed before an intended early action. In general, all actions should be executed in due time, with the sense that they are never executed too early or too late.

In general, a set of composite sequential and parallel actions can be depicted as an execution *graph*; each node in a graph represents either a sequential or a parallel activity. Execution graphs indicate exactly when a particular action is invoked or when (parallel) actions synchronize. Parallel actions can be nested, i.e. a parallel action can contain other parallel activities. Therefore, synchronization always takes place relative to the parent node in the execution graph. This way, the scheduling and synchronization scheme as imposed by the execution graph preserves the relative order of actions.

From a script point of view, a parallel action is finished when each of the individual activities are done. However, action resources are reclaimed and re-allocated incrementally. After synchronization of a particular activity in a parallel action construct its resources can be re-used immediately for other scripting purposes. Just consider a nested parallel action which consists of several sequential and parallel sub-actions, like,

$par([seq([A1, A2, A3]), seq([B1, par([C1, C2]), B2]),$
$par([D1, seq([E1, E2])])])$

The procedure of the scheduling and synchronization is shown in Figure 9.8.

9.6.3 Rotation Interpolation

We use DLP to implement the scripting language STEP . One of the issues on the implementation is to achieve the function of turn-object by introducing an appropriate interpolation between the starting rotation and the ending rotation.

Suppose that the object's current rotation is

$$R_s = \langle X_0, Y_0, Z_0, R_0 \rangle$$

and the ending rotation of the scripting action is

$$R_e = \langle X, Y, Z, R \rangle,$$

and the number of interpolations is I. In STEP , we use slerp (spherical interpolation) on unit quaternions to solve this problem. Let $Q_s = \langle w_0, x_0, y_0, z_0 \rangle$ and $Q_e = \langle w, x, y, z \rangle$ be the corresponding quaternions of the rotations R_s and R_e. The relation between a rotation $\langle X, Y, Z, R \rangle$ and a quaternion $\langle w, x, y, z \rangle$ is as follows:

$w = cos(R/2)$.
$x = X \times sin(R/2)$.
$y = Y \times sin(R/2)$.
$z = Z \times sin(R/2)$.

The function slerp does a spherical linear interpolation between two quaternions Q_s and Q_e by an amount $T \in [0, 1]$:

$slerp(T, Q_s, Q_e) = R_s \times sin((1 - T) \times \Omega)/sin(\Omega) +$
$R_e \times sin(T \times \Omega)/sin(\Omega)$

where $cos(\Omega) = Q_s \cdot Q_e = w_0 \times w + x_0 \times x + y_0 \times y + z_0 \times z$. See [Schoemake, 1985] for more details of the background knowledge on quaternions and slerp.

One of the requirement to achieve a natural transition of the two rotations is to introduce the non-linear interpolation between two rotations. STEP also allows users to introduce their own non-linear interpolation by using the enumerating type of interpolation operator. An example:

```
turnEx(Agent,l_shoulder,front,fast,
       enum([0,0.1,0.15,0.2,0.8,1]))
```

would turn the agent's left arm to the front via the interpolating points 0, 0.1, 0.15, 0.2,0.8,1.

Users can use the interaction operators *do* to calculate interpolating point lists for their own interpolation function. Therefore, the enumerating list is powerful enough to represent arbitrary discrete interpolation function.

9.7 CONCLUSIONS

In this chapter we have discussed the scripting language STEP for embodied agents, in particular for their communicative acts, like gestures and postures. Moreover, we have discussed principles of scripting language design for embodied agents and several aspects of the application of the scripting language.

STEP is close to Perlin and Goldberg's **Improv** system. In [Perlin and Goldberg, 1996], Perlin and Goldberg propose **Improv**, which is a system for scripting interactive actors in virtual worlds. STEP is different from Perlin and Goldberg's in the following aspects: First, STEP is based on the H-anim specification, thus, VRML-based, which is convenient for Web applications. Secondly, we

separate the scripting language from the agent architecture. Therefore, it's relatively easy for users to use the scripting language.

Prendinger et al. are also using a Prolog-based scripting approach for animated characters but they focus on higher-level concepts such as affect and social context[Prendiner et al., 2002]. STEP shares a number of interests with the VHML(Virtual Human Markup Language) community[5], which is developing a suite of markup language for expressing humanoid behavior, including facial animation, body animation, speech, emotional representation, and multimedia. We see this activity as complementary to ours, since our research proceeds from technical feasibility, that is how we can capture the semantics of humanoid gestures and movements within our dynamic logic, which is implemented on top of DLP.

[5]http://www.vhml.org