Pers Ubiquit Comput (2003) 7: 1–21
DOI 10.1007/s00779-002-209-4

ORIGINAL ARTICLE

Claudio S. Pinhanez · Aaron F. Bobick

Interval scripts: a programming paradigm for interactive environments and agents

Received: 1 May 2002 / Accepted: 5 October 2002

Abstract In this paper we present *interval scripts*, a new paradigm for the programming of interactive environments and computer characters. In this paradigm, actions and states of the users and the system computational agents are associated with temporal intervals. Programming is accomplished by establishing temporal relationships as constraints between the intervals. Unlike previous temporal constraint-based programming languages, we employ a strong temporal algebra based in Allen's interval algebra with the ability to express mutually exclusive intervals and to define complex temporal structures. To avoid the typical computational complexity of strong temporal algebras we propose a method, *PNF propagation*, that projects the network implicit in the program into a simpler, 3-valued (*past*, *now*, *future*) network where constraint propagation can be conservatively approximated in linear time. The interval scripts paradigm is the basis of *ISL*, or *Interval Scripts Language*, that was used to build three large-scale, computer-vision-based interactive installations with complex interactive dramatic structures. The success in implementing these projects provides evidence that the interval scripts paradigm is a powerful and expressive programming method for interactive environments.

Keywords Interactive spaces · Programming paradigms · Programming with constraints · System architecture · Temporal reasoning · Ubiquitous computing

Claudio S. Pinhanez (✉)
IBM Research, T.J. Watson,
19 Skyline Drive, Hawthorne,
NY 10532, USA
e-mail: pinhanez@us.ibm.com

Aaron F. Bobick (✉)
Georgia Institute of Technology,
GVU Center, 801 Atlantic Dr.,
Atlanta, GA 30332, USA
e-mail: bobick@cc.gatech.edu

1 Introduction

Recent years have seen an increasing interest in the development of interactive environments where users interact with devices, data, images, computational agents, and computer characters. Such environments can be virtual spaces as it is the case of Virtual Reality (VR) systems, or real physical spaces with embedded computing devices and control. Examples of the latter include location-based entertainment spaces [1–4], interactive offices [5–7], classrooms [8] and homes [9, 10].

However, little attention has been paid to the special requirements imposed by interactive environments on programming methods. In particular, in highly immersive situations both the actions and states of the environment and its computational agents, as well as the actions of the users, are not instantaneous events, but take time to be performed. This makes difficult the use of the traditional event-based programming paradigm commonly employed in desktop interaction.

Although the problem of system actions with non-zero duration has been the object of research in the multimedia community, interactive environments extend significantly the problem since they include situations where the user's actions also extend through periods of time. Unlike desktop systems where the interaction occurs mostly through mouse clicks, in interactive environments the users talk, walk, make complex gestures, and interact with objects, computational agents, computer characters, and other users. However, the programming systems for interactive environments proposed so far (e.g. [11, 12]) use simple paradigms that do not provide adequate ways to handle this kind of long-duration interactivity.

Let us examine an example to better characterise the issues involved in programming an interactive environment. Consider a simple interactive environment in a museum context, composed of three different areas A, B and C, where pre-recorded tapes have to be played

whenever a user enters the corresponding area. Let's also assume that the system is able to differentiate among users and that the tapes cannot be played simultaneously because the sound overlap makes them unintelligible. Figure 1 shows some typical difficulties to the programming of such system. In instant t_0, user 1 enters space A, triggering the playing of tape A. Notice that there is a *delay* in the start of the tape. Most important, however, if we do not want the tape to play again while user 1 is in the area A (for instance, after instant t_3), then the control system cannot be purely reactive, i.e. based on a simple event structure such as "if user is in area A, play tape A", because that event would be repeatedly activated by the presence of user 1 in area A. Here, the control program needs some way to represent and access notions of *continuous states* and *interaction history* in order to activate an action.

When user 2 enters space B in instant t_1, the system normally would start playing tape B. However, since tape A is playing, the start of tape has to be delayed, because the two actions, as described before, are *mutually exclusive*. However, since tape A lasts longer than the presence of user 2 in area B, the playing of tape B has to be, ultimately, ignored. The moving of user 2 to area C, in instant t_2, however, does not initiate the playing of the tape C either. Only after tape A ends, at t_3, it is possible to start tape C. Instant t_5 illustrates another common problem: although tape C is playing, there is no user in area C. This is a typical situation where it would be desirable to have ways to *stop an action abruptly*, in this case by triggering a fadeout or similar effect. Finally, when tape C ends on instant t_6, the system is faced with a choice between *equivalent but mutually exclusively actions*: it can either play tape A or tape B, but not both.

It is easy to see that similar problems arise also in different scenarios of interactive environments. Suppose, for instance, a home environment for a visually impaired person where two computer agents, agent 1 and agent 2, provide help with communication with other people and with dangerous situations, respectively. Looking at the same Fig. 1, agent 1 detects at instant t_0 that the person is in the environment, and decides to deliver a message left for her (tape A). At instant t_1, agent 2 sees a potentially dangerous situation where an advisory message could be helpful. Like in the previous example, there is a conflict for resources here, since two speech actions cannot happen at the same time. It is straightforward to see how the situation described in Fig. 1 develops in this scenario with a warning (a reaction to situation B at t_1) becoming obsolete in t_2, and the alert of a communication attempt (for instance, a call from a relative) being preempted at t_5 because of a warning related to a dangerous situation (tape C).

As we see in the above examples, a programming paradigm for interactive environments has to address complex temporal structures that are a result of users' and computer actions and states having duration. Although it is theoretically possible to represent such temporal structures using simple models such as state machines or event-loop systems, this approach requires the enumeration of all possible combinations of states or events, creating programs that are hard to write, debug, and maintain. Moreover, since such systems require that each combination of states to be represented as a single machine state or event, the addition of new actions, sensors, agents, or users often causes an exponential increase in the number of machine states or events. This is a direct consequence of having mutually exclusive parallel actions or states (see Pinhanez [13]). Such problems have been consistently found by the authors in the development of interactive environments [2, 4, 5, 14].

This paper proposes a method to program interactive environments based on temporal constraints. Called *interval scripts*, this paradigm encapsulates actions and states in temporal intervals constrained by the temporal primitives of Allen's algebra [15]. Our objective is to provide the programmer of an interactive environment with a tool that combines expressiveness and simplicity and, at the same time, produces control code that can be run in real time. Among the desired programming features, we include in interval scripts not only facilities for the simple handling of time durations, but also features such as the definition of mutually exclusive actions, the ability to impose restrictions on what can happen (negative scripting), easy encapsulation of actions, basic handling of interaction history, and automatic recovery from errors and unexpected situations.

To provide these programming facilities, an interval script contains descriptions of how to initiate and stop each action of the interactive environment or its agents and how to determine the occurrence of actions and states of the users, environment states, and the states of computational agents. The control of activation of an action is determined by verifying the current state of the action, comparing it with the desired state according to the constraints with other actions, and issuing commands to start and stop actions as needed. Also, as we describe in detail later, interval scripts de-couple the

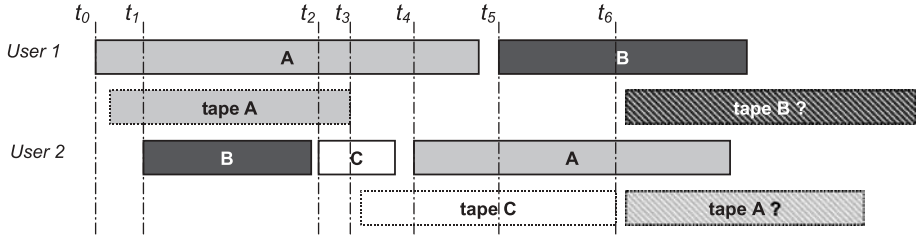


Fig. 1 Examples of difficulties when programming interactive environments

actual state of an action from its desired state, allowing for the introduction of methods for recovery from sensor errors and increasing the robustness of the system.

Traditional constraint propagation methods cannot be used to run the interval algebra networks associated to the interval scripts due to the typical size of the networks and the requirement of real-time computation [16]. To solve this problem, our system employs a novel method for constraint propagation in such networks that overcomes these limitations by projecting the network implicit in the script into a simpler, 3-valued (*past*, *now*, *future*) *PNF-network*, where constraint propagation can be approximately computed in linear time (initially proposed in Pinhanez et al. [17])

This paper starts by reviewing the current programming paradigms and identifying their shortcomings. In Sect. 3 we introduce the interval scripts paradigm through some simple examples. The core of the paper is the description of the PNF theory needed to allow the efficient computation of temporal constraint propagation algorithms, and of the run-time engine architecture that uses it, as described in Sect. 4 and 5. An implementation of interval scripts paradigm into a programming language called *Interval Scripts Language*, or *ISL,* is described in Sect. 6, together with examples of its use. Section 7 explores the three concrete cases of interactive environments that have been built using the paradigm and in Sect. 8 we discuss future directions in our research.

2 Problems with current programming techniques

The idea of interval scripts was spawned by our dissatisfaction with the tools for the integration of I/O devices and software modules in interactive environments and for the control of computer agents. As described by Cohen, systems to control interaction tend easily to become "*big messy C program(s)*" ([18], Fig. 2). Also, from our experience with *The KidsRoom* [2], it became clear that one of the major hurdles to the development of interesting and engaging interactive environments is that the complexity of the control program grows faster than the complexity of the programmed interaction.

For example, *The KidsRoom* was programmed using an event-loop paradigm [2]. However, during the development of *The KidsRoom* we encountered many situations where unanticipated and undesirable co-occurrences of events disrupted considerably the interaction. A typical case, similar to the one described in the introduction, was when multiple characters were triggered to talk or act by different events. The problem is that the characters, in many trials, ended up being triggered almost simultaneously, creating unintelligible overlapped speech or action. Since the environment used a control structure based in event-loops, it became necessary to examine each possible co-occurrence and to explicitly program how to handle every individual case. However, for each new action or character added to the environment, it was necessary to explore in the code the interaction between its actions with every action of other characters. In other words, the result was a *de facto* exponential increase in the number of different situations to be handled by the programmer.

2.1 Finite-state machines and event loops

The most common technique used to program and control interactive applications is to describe the interaction through finite-state machines. This is the case of one of the most popular languages for the developing of multimedia software, *Macromedia Director's Lingo* [19]. In *Lingo* the interaction is described through the handling of events whose context is associated with specific parts of the animation. There are no provisions to remember and reason about the history of the interaction and the management of story lines. The same situation occurs with *Max* [20], a popular language for control of music devices (see Roads [21]) and adopted in some interactive art installations [22]

Videogames are traditionally implemented through similar event-loop techniques [23]. To represent the interaction history, the only resort is to use state descriptors whose maintenance tends to become a burden as the complexity increases. Most of all, a fundamental problem with the finite state model is that it lacks appropriate ways to represent the duration and complexity of human action, computational agents, and interactive environments: hidden in the structure is an assumption that actions and occurrences are pinpoint-like events in time (coming from the typical point-and-click interfaces for which those languages are designed).

2.2 Constraint-based programming languages

The difficulties created by the finite-state based model have sparked a debate in the multimedia research community concerning the applicability of constraint-based programming (starting with the works of Buchanan and Zellweger [24] and Hamakawa and Rekimoto [25]) versus procedural descriptions such as state machines (for example, see van Rossum et al. [26]). In general, it is believed that constraint-based languages are harder to learn but more robust and expressive as discussed in Vazirgiannis et al. [27].

For example, Bailey et al. [28] defined a constraint-based toolkit, *Nsync*, for constraint-based programming of multimedia interfaces that uses a run-time scheduler based on a very simple temporal algebra. The simplicity of the temporal model and, in particular, its inability to represent non-acyclic structures, is a major shortcoming of *Nsync* and similar systems such as *Madeus* [29], *CHIMP* [30], *ISIS* [31] and *TIEMPO* [32].

André and Rist [33] have built a system called *PPP* that designs multimedia presentations based on

descriptions of the pre-conditions of the media actions, their consequences and on the definition of temporal constraints. The *PPP* system uses a strong, hybrid temporal algebra that combines intervals and points as its primitives (based on Kautz and Ladkin's [34] work). The descriptions of media actions are used to plan a multimedia presentation that achieves a given goal. The multimedia presentation is represented as a constraint network. During run-time, the scheduler has to build and choose among all the possible instances of the plans. However, since Kautz and Ladkin's propagation techniques are exponential in time, and the number of possible plans can be exponentially large, the applicability of *PPP* is restricted to simple cases with limited interaction (otherwise the number of possible plans increases too quickly).

2.3 Programming interactive environments

Starting with the pioneer work of Myron Krueger [35], the interest in building interactive environments, has substantially grown in the last decades (see Bederson and Druin [36] for a review). In the case of virtual reality, it seems that the user-centred and exploratory nature of most interfaces facilitates the programming of the interface by using state machines and event loops, using languages such as *MR* [11] and *Alice* [12]. The latter was used in Disney's *Aladdin* ride (described in Pausch et al. [1]). It is a language that allows rapid prototyping but can hardly describe any non-trivial temporal structure.

Cohen [18] proposed a distributed control structure, the *Scatterbrain*. In this paradigm, used in the control of a sensor-loaded room, knowledge and behaviour is dispersed among multiple agents that negotiate among themselves. Although an appealing model, the use of multiple agents without centralised control makes authoring extremely difficult in practice. At the opposite end of the spectrum, Dey [37] proposes a framework for the prototyping of context-aware environments. Both proposals lack models of temporal control capable of efficiently handling even modestly complex temporal structures such as the ones described in the introduction section.

3 The interval scripts paradigm

In traditional desktop-based interfaces and programs, system and user actions are assumed to have no significant duration, except in some cases of multimedia programming. In the same manner, traditional programming is based on the concept that the states of all variables remain the same as long as no event occurs. When an event occurs, a piece of program is executed, normally assuming 'instantaneous' response, and then a new state is reached. In other words, states are a-temporal, i.e. they last as long as no new detected event happens.

As described in the example of the introduction, interactive environments and agents perform actions that often take a non-trivial amount of time, such as playing a sound file, showing a character animation, or closing a door. Similarly, user actions many times also have non-zero duration, such as when performing a gesture, walking around the space, or saying a sentence to a speech interface. These observations drove us to design a programming paradigm where all elements of a program, including its states and actions, are assumed to have non-zero duration.

The fundamental concept of the interval scripts paradigm is that all actions and states are associated with the temporal interval in which they occur. However, the actual beginning and end time of the intervals are not part of the script. Instead, the script contains a description of *the temporal relationships or constraints* that the intervals should satisfy during run-time. These constraints are enforced by a run-time engine that examines the current states of all intervals, and then coordinates the system's reaction to ensure that the interval's states respect the constraints.

For example, suppose we want to describe the situation where a door opens whenever the user is in front of it. To do this with interval scripts, we associate the user presence in front of the door to an interval A and the action of opening of the door to another interval B. We then put a temporal constraint stating that the interval of time of A always has the same start time as the interval B During run-time, the interval scripts engine monitors the presence of the user in front of the door. When it sees him, the engine detects that, according to the constraint, both intervals should be happening at the same time and tries to open the door in an attempt to satisfy the constraint.

Developing an interface or interactive system with interval scripts is a classical case of programming by constraints as discussed in Sect.2. However, the idea of using temporal constraints to describe interaction in interactive environments initially appeared in Pinhanez et al. [17], in a different formulation from the one described in this paper. That work drew from a technique for representing the temporal structure of human actions by Pinhanez and Bobick [38]. The formalism and theory presented here considerably improve the syntax and semantics of the paradigm, as well as introduce a much more consistent, efficient, and robust run-time engine than in those works, as we will see in the next sections.

3.1 Allen's interval algebra

To model the time relationships between two actions, a program in interval scripts employs the interval algebra proposed by Allen [15]. The representation of temporal constraints in Allen's algebra is based on the 13 possible primitive relationships between two intervals as shown in Fig. 2. Given two actions or states in the real world, all their possible time relationships can always be described by a disjunction of those primitive time relationships.

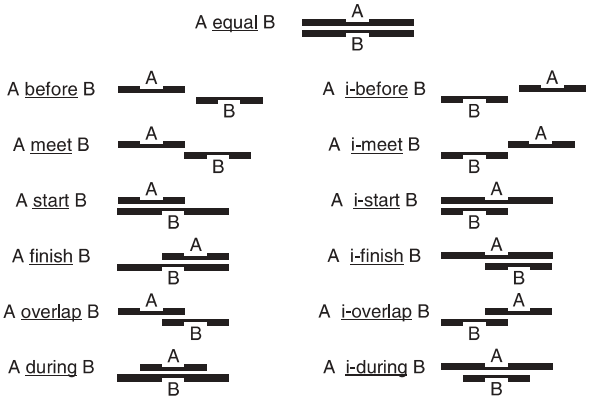


Fig. 2 Allen's 13 primitive relationships between two time intervals

For instance, in the preceding example, the relationship *A* i-start *B* correctly expresses the temporal constraint between the presence of the user (interval *A*) and the time needed to open the door (interval *B*), if we assume the user waits for the door to open. In practice, however, it is hard to determine whether the user will leave the door area before the end of the action of opening the door, so the relationship between *A* and *B* is better described by the disjunction start OR equal OR i-start, i.e. that the two intervals start at the same time. Of course, in any actual occurrence of the intervals, only one of the disjunctive elements actually happens.

To simplify the manipulation of the intervals and their constraints, a set of actions and states and their respective temporal constraints can be represented by a network in which each action or state is associated with a node and the temporal constraints between them are represented by arcs. The networks composed of such nodes and disjunctions of Allen's primitive relationships as arcs are called *interval algebra networks*, or simply, *IA-networks* [15]. It is important to notice that the nodes are, in fact, variables that can assume any temporal interval representing the actual interval of time of when an action or state occurs.

3.2 Strong interval algebras

Allen's algebra allows the expression of mutually exclusive intervals. For instance, to state that a computer agent does not perform actions *C* and *D* at the same time we simply constrain the relation between the intervals *C* and *D* to be meet OR i-meet. That is, in every occurrence of the intervals, either *C* comes before *D* or after *D*, with no overlap.

The ability of expressing mutually exclusive intervals defines different classes of temporal algebras [39]. In general, algebras without mutual exclusiveness allow fast constraint satisfaction but are not expressive [16]. However, in our approach we are able to employ Allen's *strong* algebra in a real time system because we have developed a fast method to compute approximations of the values that satisfy the constraints as described later in this paper.

André and Rits [33] have also used a strong temporal algebra to script multimedia presentations. Their work employs a more expressive temporal algebra that combines Allen's [15] and Villain's [16] proposals as suggested by Kautz and Ladkin [34]. However, constraint propagation in this temporal algebra is basically $O\left(n^2\left(e+n^3\right)\right)$, where n is the number of nodes and e is the time required to compute the minimal network of the IA-network; typically, e is exponential in the number of nodes, $O(e) = O(2^n)$, making the whole system basically $O\left(n^2 2^n + n^5\right) = O(2^n)$. Moreover, their system requires the enumeration in computer memory of all possible temporal orders of actions implied by the constrained network, which can grow exponentially, especially in the case of multiple and frequent user interaction.

3.3 Example of programming with temporal constraints

To understand how to use interval algebras in interaction programming, let us examine an example from the script of one of our interactive environments, the art installation called *It* described in Sect.7. In that installation a Computer-Graphics (CG) object that look likes a photographic camera, here referred to as *camobject*, appears on a big screen and interacts with the user in the environment.

Suppose we want to describe a situation where *camobject* moves front, that is, moves forward towards the user, produces a clicking sound (as it was taking a picture), and then moves back to its previous position. The first step is to associate each of these three actions to nodes in an IA-network, and name them, respectively, *camobject moves front*, *camobject clicks* and *camobject moves back*. To program the desired sequencing of the three actions, we define meet constraints between each consecutive pairs:

"camobject moves front" meet *"camobject clicks"*
"camobject clicks" meet *"camobject moves back"*

Figure 3 shows the desired relation between the three actions. The first constraint states that the action *camobject moves front* should meet with the action *camobject clicks*, i.e. the latter action should start immediately after the end of the former. The second constraint establishes a similar constraint between the actions *camobject clicks* and *camobject moves back*.

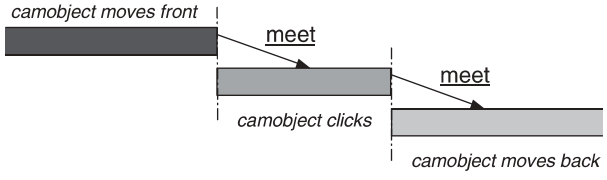


Fig. 3 Diagram of temporal constraints showing a sequence of three intervals

Now, suppose that we want to define an action *camobject takes a picture* that is composed of these three intervals. This could be accomplished by the following constraints (see also Fig. 4):

"camobject moves front" start *"camobject takes a picture"*
"camobject clicks" during *"camobject takes a picture"*
"camobject moves back" finish *"camobject takes a picture"*

Finally, if we want the whole sequence to start when the user makes a pose to the camera-like object (as required by the art installation environment), a composed disjunctive constraint can assure that whenever the user starts posing, the sequence for taking pictures is activated (see Fig. 4):

"user makes a pose" start OR equal OR i-start *"camobject takes a picture"*

In this case the disjunction start OR equal OR i-start is the way to impose the constraint that the action and the sensor have the same starting time without requesting any particular configuration of the ending time. In other words, although *"user makes a pose"* and *"camobject takes a picture"* are supposed to start together, either of them can finish first without violating this constraint.

3.4 Example of the use of mutually exclusive constraints

Suppose now that we do not want to allow the taking of pictures if the user is moving in the environment. This is an example of mutually exclusive intervals mentioned previously. First, as above, we assume that a node in the IA-network called *user is moving* has been associated with the moving state of the user as detected, for instance, by a sensing device. To program that the 'clicking' action should not happen if the user is moving, we can simply enforce the following constraint:

"camobject takes a picture" before OR i-before *"user is moving"*

Figure 5 shows four possible occurrences of the sensor associated with *user is moving*. Cases (1) and (2), show situations where it happens respectively before or after *camobject takes a picture*, and therefore are compatible with the occurrence of this sensed activity. However, cases (3) and (4) show hypothetical situations where *user is moving* overlaps with *camobject takes a picture,* and thus it is not compatible.

Our goal is to design a run-time engine that prevents the occurrence of case (3) by not allowing the *camobject takes a picture* action to start while *user is moving* is not finished; and by immediately (maybe prematurely) finishing *camobject takes a picture* if the user starts moving as in case (4). To understand better how this is accomplished, it is necessary to examine how the run-time engine actually works, as we will describe in Sect.4 and 5.

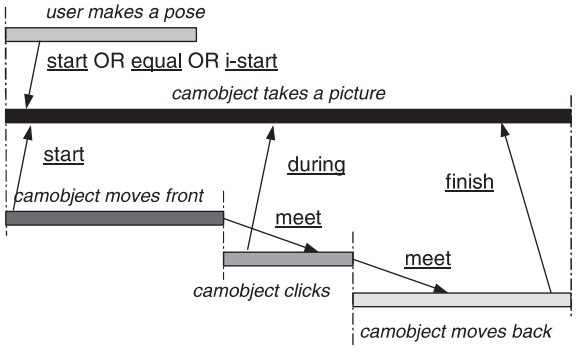


Fig. 4 Diagram of temporal constraints showing an action composed of a sequence of three others and a sensor that start at the same time as the composed action

3.5 The case for Allen's algebra

As can be seen from the above examples, there are several reasons to use Allen's algebra to describe relationships between actions as the programming paradigm for interactive environments. First, no explicit mention of the interval duration or specification of relations between the intervals' starting and endings are required. Secondly, the notion of disjunction of interval relationships can be used to declare multiple paths and interactions in a story. Third, it is possible to declare mutually exclusive intervals that describe actions and states whose co-occurrence is not desirable.

A fourth reason to use Allen's algebra, not discussed here, is that it is only necessary for the programmer to declare the relevant constraints. By applying *Allen's path consistency algorithm* [15] to the IA-network associated with the script, it is possible to generate a more constrained version of the network that contains all and only the constraints that are consistent with time. Fifth, it is possible to determine in linear time which computational agents' and characters' and environment's reactions to trigger in response to users' actions by

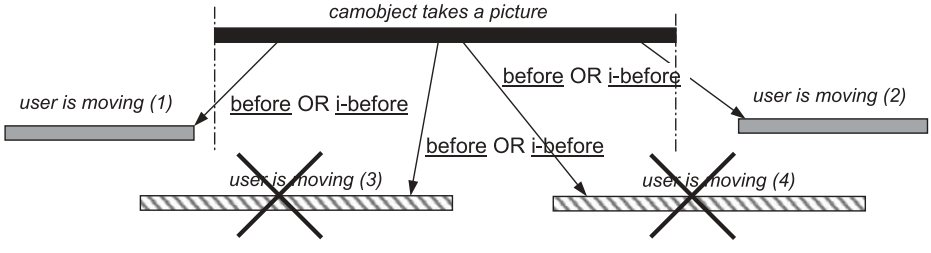


Fig. 5 Possible occurrences of *user is moving*

properly propagating occurrence information from one node of the IA-network to the others, using the PNF propagation method described next.

4 A method for fast constraint propagation in strong algebras

Programming with interval scripts involves the association of actions and world states of an interactive environment to a collection of nodes in an IA-network with temporal constraints between them. However, in order to control an interactive environment or agent based on such descriptions, it is necessary to employ a run-time engine that examines which intervals are happening, considers the constraints between them, and controls the start and stop of intervals as needed to enforce the constraints. To perform such actions, it is necessary to use a process known as *constraint propagation*. In general, performing constraint propagation in an IA-network is exponential (as proved in Vilain and Kautz [16]). We overcome this problem by using a simplification of constraint propagation in IA-networks called *PNF propagation*, as described in this section.

4.1 Minimal domains of IA-networks

As mentioned above, the collection of temporal constraints between nodes is represented by an *interval algebra network* [15], which is a network where the nodes correspond to variables on temporal intervals and the temporal constraints are links between them. Figure 6 displays the IA-network associated with the three nodes connected through meet constraints of the *camobject* example described before.

Traditional constraint satisfaction in interval algebra networks tries to determine for each network node the sets of time intervals (pairs of real numbers describing segments of the time line) that are compatible with the constraints given the known occurrence of some intervals. An assignment of time intervals that satisfies the constraints is called a *solution* of the network. Given a network node, the set of all time intervals for that node that belongs to at least one solution is called the *minimal domain* of the node. Notice that if there is at least *one* interval in the minimal domain of 2a node that contains a given instant of time then the node *can* be happening at that time. If *every* time interval of the minimal domain of a node contain that given instant of time, then the node *must* be happening at that given moment.

This observation constitutes the basic principle of our run-time engine. However, direct constraint propagation is *NP-hard* due to the combinatorial explosion involved in the manipulation of sets of intervals (see Vilain and Kautz [16]). To overcome this difficulty we devise an optimisation scheme called *PNF propagation* comprising two mechanisms, *PNF restriction* and *time expansion*.

4.2 PNF restriction

The idea of simplifying interval algebra networks was first proposed by Pinhanez and Bobick [40], in a work motivated by the problem of action recognition in a computer vision system. The key idea is that for control and recognition purposes there is little information coming from the duration of the nodes. Instead, the focus in such situations is to simply determine if an interval has happened (*past*), is happening (*now*), or may still to happen (*future*). Based on this criterion, it is possible to project an interval network into a similar network where the nodes corresponding to the intervals can assume only one of the three symbols, *past*, *now*, or *future* – a *PNF-network*.

However, in real systems it is very difficult to determine exactly in which of these three states a node is. For instance, suppose a node that is associated to a binary sensor such as a light switch which has no mechanism to store the past activation history. If the sensor is *on*, it is easy to see that we can represent its state by associating the symbol *now* to the node. However, if the sensor is *off*, it is impossible for the sensor to locally determine whether the switch was turned on in the *past*, or if it has never been switched on before, that is, that it may be *on* in the *future*. To represent such situations where information about the actual occurrence of an action or state associated to a node is not available, we can assign nodes with disjunctions of any of three basic symbols. For instance, the state of an *off* switch is easily represented by the disjunctive set $\{past, future\}$.

Let us formalise now the concepts introduced before. To simplify notation, we start by defining the set M of all subsets of $\{past, now, future\}$

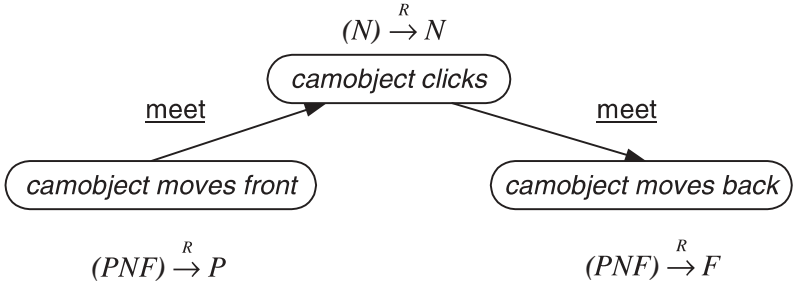


Fig. 6 Interval algebra network associated with two meet constraints and an example of PNF restriction

$$M = \{\varnothing, \{past\}, \{now\}, \{future\}, \{past, now\}, \{now, future\}, \{past, future\}, \{past, now, future\}\}$$

whose elements, called *PNF-values*, are abbreviated as

$$M = \{\varnothing, P, N, F, PN, NF, PF, PNF\}$$

Also, we call any tuple $W = (W_1, W_2, \ldots, W_n)$ of PNF-values associated to the n nodes of a PNF-network *a component domain* of the PNF-network, where each $W_i \in M$, and $W \subseteq M \times M \times \ldots M = M^n = U$. Since each element of a component domain represents possible states of its associated node, we call each W_i the *PNF-state* of the node i.

It is easy to see that, given a primitive temporal relationship r between two nodes A and B of a PNF-network, the admissible states of the two nodes must satisfy the constraints depicted in table 1. For instance, if A is happening *now* (N state), and A meet B, then B can only happen in the *future* (F state, as highlighted in Table 1). The construction of the table is formally justified in Pinhanez [13], but it should be noticed that it corresponds to our intuitive notions of *past*, *now*, and *future*.

If the temporal constraint is a disjunction of primitive temporal primitives, we simply take the union of the PNF-values corresponding to the primitives. For instance, suppose A and B are mutually exclusive, A meet OR i-meet B, and A is happening *now*. Table 1 yields that for the meet constraint, B may happen in the *future* (F), while for the i-meet constraint B must have already happened (P). By taking the union of the two constraints in the disjunction, $P \cup F = PF$, we determine that the actual state of B must belong to PF, and therefore, that B is not happening now.

On the other hand, if the node A is associated to a non-singleton PNF-value (i.e. PF, PN, NF, or PNF), it is straightforward to see that the possible values for B are simply the union of the admissible values for each element of the set associate with A. For instance, if A finish B and the PNF-state of A is PN, then the PNF state of B is the PNF-value $PN = P \cup N$.

Table 1 Admissible values for primitive temporal relations in a PNF-network

Admissible values of B when A r B			
r	A = {P}	A = {N}	A = {F}
equal	P	N	F
before	PNF	F	F
i-before	P	P	PNF
meet	PN	F	F
i-meet	P	P	NF
overlap	PN	NF	F
i-overlap	P	PN	NF
start	PN	N	F
i-start	P	PN	F
during	PN	N	NF
i-during	P	PNF	F
finish	P	N	NF
i-finish	P	NF	F

Given an initial set of PNF-values for each node represented by a component domain $W = (W_1, W_2, \ldots, W_n)$, we define a *solution* of a PNF-network under W to be any assignment of one of the basic $\{past, now, future\}$ values to each node i of the network in which each assigned value belongs to W_i and that each pair of nodes satisfies the constraints of Table 1. We call the *minimal domain of a node* under W as the set of all values of a node that belong to at least one solution under W. Notice that the minimal domain of a node is also a PNF-value. Similarly, the *minimal domain* of a PNF-network under W is the component domain composed of the minimal domain of each node of the network.

Let us then define a function, the *restriction of a PNF-network* under W, noted $R(W)$, that computes the minimal domain of the network under W. That is, the restriction of a PNF-network eliminates all the values from each component of W that are incompatible with at least one solution under W. Notice that the restriction of a component domain W is also a component domain, $R(W) \subseteq U$, and that restriction is maximal, i.e. $R(R(W)) = R(W)$. Notice that if the minimal domain of any node is empty, i.e. $W_i = \varnothing$ for any $1 \leq i \leq n$, then there are no solutions for the network.

Figure 6 also shows a simple example of PNF restriction. The initial set of values for each node is shown between parenthesis and the values after restriction appears on the right side of the arrows. In this case, we assume that it is known that the *camobject clicks* action is happening (N). Since this action must happen after *camobject moves front* because of the meet constraint, the only possible state for the later node is *past* (P), as shown in Fig 6. Similarly, *camobject moves back* cannot have happened because it must follow *camobject clicks,* and therefore its possible state is restricted to *future* (F).

Computing the exact minimal domain of a PNF-network is, however, still a *NP-hard* problem (see Dechter [41]). Instead, we employ in our applications a conservative approximation to the minimal domain based on the computation of the *arc-consistency* (as proposed by Mackworth [42]) of the PNF-network. The big advantage is that arc-consistency is $O(c)$ in the number of constraints c, that is, at most $O(n^2)$ in the number of nodes n.

The procedure in Fig. 7 shows an algorithm that computes the maximal arc-consistent domain of an n-node PNF-network under a component domain $W = (W_1, W_2, \ldots, W_n)$. This is a version of the arc-consistency algorithm AC-2 proposed in Mackworth [42] and adapted to the component domain notation. The algorithm uses the function ϕ which given a PNF-value and a set of primitive relations, returns a PNF-value that satisfies Table 1. In [13], Pinhanez proves that the result of this algorithm always contains all possible solutions of the network (that is, it is conservative), and it is linear in the number of constraints. Pinhanez also shows that arc-consistency produces a reasonable approximation of the minimal domain (see Pinhanez [13] for details).

Fig. 7 Algorithm to compute arc-consistency

Input: a PNF-network with nodes $w_1, w_2 \ldots w_n$ and temporal constraints P_{ij}; a component domain, $W = (W_i)_i$ representing the initial state of the nodes.

Output: $AC(W)$, the maximal arc-consistent component domain that is contained in W

initialize a queue Q with all w_i such as $W_i \neq \text{PNF}$ (1)
$\overline{W} \leftarrow W$ (2)
while $Q \neq \varnothing$ (3)
  $w_0 \leftarrow first(Q)$ (4)
  for each node w_i (5)
    $X \leftarrow \phi(\overline{W_0}, P_{i_0 i})$ (6)
    if $\overline{W_i} \neq X \cap \overline{W_i}$ (7)
      $\overline{W_i} \leftarrow X \cap \overline{W_i}$ (8)
      $queue(w_i, Q)$ (9)
return $\overline{W}$ (10)

4.3 Time expansion

PNF-restriction deals exclusively with determining feasible options of an action *at a given moment of time*. However, information from a previous time step can be used to constrain the occurrence of intervals in the next instant. For example, after an interval is determined to be in the *past*, it should be impossible for it to assume another PNF-value, since, in our semantics, the corresponding action is over. To capture this concept, we define a function that time-expands a component domain into another that contains all the possible PNF-values in the next instant of time for each node.

Formally, we define a function T_m, called the *time expansion function*, that considers a component domain W^t at time t and computes the component domain $W^{t+1} = \mathrm{T}_m(W^t)$ at time $t+1$ by considering what the possible states are in $t+1$. To define that function, we start by defining a time expansion function τ_m for each element of $\{past, now, future\}$, such as:

$$\tau_m(past) = P \quad \tau_m(now) = PN \quad \tau_m(future) = NF$$

Notice that τ_m assumes that between two consecutive instants of time there is not enough time for an interval to start and finish. That is, if a node has a value *future* at time t, it can only move to *now*, but never straight to the *past* state at time $t+1$. Based on τ_m, we define the function that expands the elements of M, $\mathrm{T}_m : M \rightarrow M$ as being the union of the results of τ_m,

$$\mathrm{T}_m(\Omega) = \bigcup_{\omega \in \Omega} \tau_m(\omega)$$

and the time expansion of a component domain W, $\mathrm{T}_m : U \rightarrow U$ (abusing the notation), as the component-wise application of the original T_m on a component domain $W = (W_i)_i$,

$$\mathrm{T}_m(W) = (\mathrm{T}_m(W_1), \mathrm{T}_m(W_2) \ldots \mathrm{T}_m(W_n)).$$

Pinhanez [13] shows that it is possible to define a similar time expansion function that allows an interval to start and finish between two consecutive instants of time. For simplicity, we limit the discussion in this paper to the function defined above.

4.4 PNF propagation

Now that we have methods both for constraint propagation and for incorporating time information, we are ready to define the temporal reasoning foundation of our run-time engine. The goal is to determine, for each instant of time, which PNF-states are compatible with the current information coming from sensors, the previous state of the system, and the constraints between the intervals.

Let us consider an initial component domain W^0 as being composed only of PNF, $W^0 = (PNF)_i$. Then, for each instant of time t, we can determine through sensor information or external sources the PNF-state of some of the nodes. With this information, we create the component domain V^t containing all these known values and assigning PNF for the other nodes. Then, given the previous component domain W^{t-1}, we can compute an upper bound of the current minimal domain of the PNF-network (see the proof in [13]) by making

$$W^t = R(\mathrm{T}_m(W^{t-1}) \cap V^t)$$

We call this process *PNF propagation*. Notice that the more information contained in V^t, the smaller is W^t. In the extreme case, if V^t is the minimal domain, then W^t is also the minimal domain. In most cases, however, we will have V^t providing new information that is filtered through the intersection with the past information (provided by $\mathrm{T}_m(W^{t-1})$). Then, PNF-states incompatible with the structure of the problem are removed by restriction, through the computation of the minimal domain. In practice, we can employ the arc-consistency algorithm as defined in Fig. 7 instead of restriction to assure that computation occurs in linear time.

Let us examine an example of PNF propagation. Suppose a simple system where an on/off motion sensor has to turn off the lights of a room whenever someone leaves the environment. We can represent this situation by an IA-network composed of two intervals, A and B, where A corresponds to the motion sensor detecting people in the environment and B the state where the lights are off. To program the automatic turn off of the lights, we simply state A meet B. The standard way to represent the *on*/*off* states of A in the PNF formalism is to assign the PNF-value N to the PNF-state of A when A is *on*, and PF when A is *off*.

Suppose that the initial state of the sensor A is *off* (PF) while B's is unknown (PNF), represented by $W^0 = (PF, PNF)$. Assume that in the next instant of time, $t = 1$, A turns *on*, represented by $V^1 = (N, PNF)$. Using PNF propagation,

$$\begin{aligned} W^1 &= R\big(\mathrm{T}_m\big(W^0\big) \cap V^1\big) = R(\mathrm{T}_m((PF, PNF)) \cap (N, PNF)) \\ &= R((PNF, PNF) \cap (N, PNF)) \\ &= R((N, PNF)) = (N, F) \end{aligned}$$

The result, $W^1 = (N, F)$, yields that since A is *on*, then B has not occurred yet. If in the next instant of time, $t = 2$, the motion sensor fails to detect a user in the environment, $V^2 = (PF, PNF)$, we obtain that the lights must be *on*. Indeed,

$$\begin{aligned} W^2 &= R\big(\mathrm{T}_m\big(W^1\big) \cap V^2\big) = R(\mathrm{T}_m((N, F)) \cap (PF, PNF)) \\ &= R((PN, NF) \cap (PF, PNF)) \\ &= R((P, NF)) = (P, N) \end{aligned}$$

5 The interval scripts run-time engine

Programming in interval scripts is achieved by defining temporal constraints between nodes associated to sensors and actuators of the interactive environment, agent, or computer characters. The constraints describe the desired temporal structure of the occurrences of the actions and states associated with the sensors and actuators. However, an interval script only declares how the actions and sensed states should and should not happen when the interactive environment or agent is running. To actually control the environment or agent, it is necessary to use a run-time system that collects the information from the sensors and, based on the interval script, actually triggers and stops the actions of actuators — the *interval scripts run-time engine*. Although there may be different ways to implement a run-time engine for intervals scripts (for instance, using André and Rist's constraint propagation method [33]), we describe here an implementation based on the concept of PNF propagation introduced in the previous section.

The basic cycle of our run-time engine is composed of four stages. First the engine computes the states of all sensors. Second, it applies PNF propagation to determine how the system can act so in the next cycle the states of all sensors and intervals are compatible with the constraints. Third, the engine selects a particular combination of goal states for all actuator nodes. And fourth, the engine tries to start or stop actions calling appropriate functions, aiming to achieve node states in which all constraints are satisfied.

One way to understand our run-time engine is to regard the engine as one player in a two-player game, and assuming the other player to be the environment and its users. Every play is composed of two "moves", the first performed by the environment and/or its users (sensed/computed by the first stage of the engine). The second "move" is performed by the actions triggered by the fourth stage of the run-time engine. The second and third stages can then be seen as a planning process in which the system explores the effects of different combinations of actions, trying to find a set of actions that results in states of the nodes that satisfy the constraints. This planning process, whose computation is based on PNF propagation, assumes that the environment remains constant except for the system's actions. We will later discuss how to overcome this limitation within this formalism.

5.1 Start, stop, and state functions

To connect an interval script to a real environment or agent, it is necessary to associate nodes of the interval script to actual sensors and actuators. In an interval script, this is accomplished by associating three types of functions to a node:

- **START:** a function that determines what must be done to start the action associated with a node;
- **STOP:** a function defining what must be done to stop the action associated with a node;
- **STATE:** a function that returns the actual current state of the interval (in our implementation as a disjunction of the three values, *past*, *now*, or *future*) corresponding, respectively, to the situation where the node has already happened, is happening in that moment of the time, or it has not happened yet.

Given the difficulty of controlling actuators in real environments, START or STOP do no need to guarantee that starting/stopping is achieved immediately, or at all. As we shall see in the rest of this section, the interval

scripts run-time engine has ways to cope with actuators' delays and failures. To do this, the STATE functions should be oblivious to START and STOP functions and should be defined so they provide a real assessment of the actual situation of the interactive environment and the on-going actions of the system and its users. STATE functions are used both to define nodes attached to sensors and to provide the actual state of execution of an action. As we see, in interval scripts we de-couple the goal of starting or stopping an action (represented by the START and STOP functions) from the actual state of its associated node.

5.2 Choosing globally consistent states for actions

Before we describe the details of the run-time engine, it is necessary to understand an additional concept. To do so, let us examine a very simple case of a network of two nodes, A and B, constrained to be temporally equal. For instance, the two nodes could correspond to the state of two devices that are required to be always turned on at the same time, such as the lights and the ventilation in a bathroom control system trying to minimise energy consumption by turning on/off the devices only when people are inside the bathroom. Let us also assume that different sensor devices separately control the lights or the ventilation, but the overall goal is to have the two actuators always in the same activation state.

Figure 8 shows a situation where the PNF-state of node A is PF and B is N, and therefore their values do not satisfy the constraint between them. In that situation the system should take one of the two actions: try to start the action corresponding to node A (so its PNF-state becomes N) or to stop node B (so its PNF-state becomes P). However, as shown in Fig. 8, there are four options of sets of actions to be taken, considering that the no action can be done, one of the two, or both. In particular, notice that if the system decides both to start A and to stop B, the constraint remains violated.

This example shows that when choosing a set of actions it is better to look for ones whose outcome lead to the global consistency of the network. If we look only locally in each node for a way to make the network consistent, it might happen the nodes change their state but keep the network inconsistent as, in the example of Fig. 8, it is the case when both A is started and B is stopped. However, this example also shows that there are multiple, often contradictory, ways to satisfy the constraints of a network.

In our run-time engine, we choose which intervals to start or to stop by looking for global solutions for the IA-network corresponding to the achievable PNF-states. However, as shown in Fig. 9, running PNF propagation on the current state generates a state (PN, PN) which represents not only the two different possible solutions, (N, N) and (P, P), achievable by starting the A action or stopping the B action, respectively, but also the PNF-states (P, N) and (N, P) which are not compatible with the constraint equal.

To decide which action to start or stop, we could examine every possible combination of PNF-values contained in the result of the PNF propagation and discard those who are not solutions. This is clearly an exponential search. In practice we have used heuristics that 'thin' the result of PNF propagation, as described next, although always looking for a globally consistent solution.

5.3 Basic cycle of the run-time engine

We now detail how the state of the intervals is determined during run-time and the process that triggers the call of START and STOP functions. At each time step the goal is to select which START/STOP functions to call in order to make the state of the world in the next cycle, represented by the PNF-state of the nodes, consistent with the temporal constraints of the PNF-network associated to the script.

Figure 10 shows the diagram of one cycle of our run-time engine. The main component is the current state S^t at time t that contains the current state of all intervals in the associated PNF-network. Each cycle t is composed of the following four stages:

I. *Determination of the current state:* all the STATE functions are called and the engine waits until all the results are reported back and stored in the component domain S^t. For this computation, the STATE function is allowed to use the previous state of the intervals, available in the previous component domain S^{t-1}, and

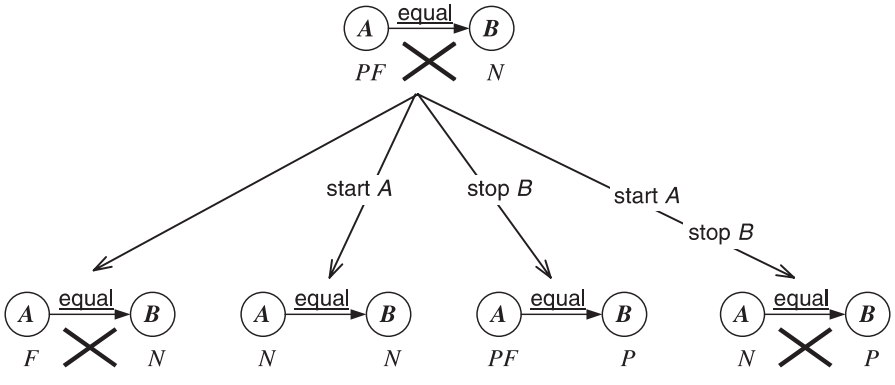


Fig. 8 Example of an inconsistent state and the effect of possible actions

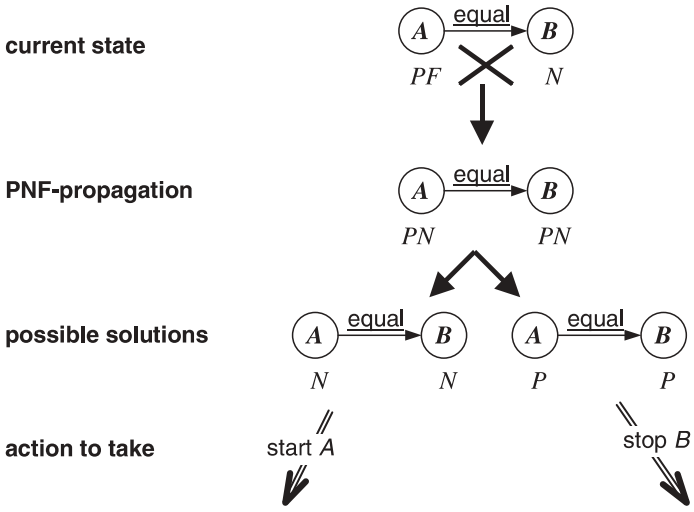


Fig. 9 Finding a globally consistent state for the situation depicted in Fig. 8

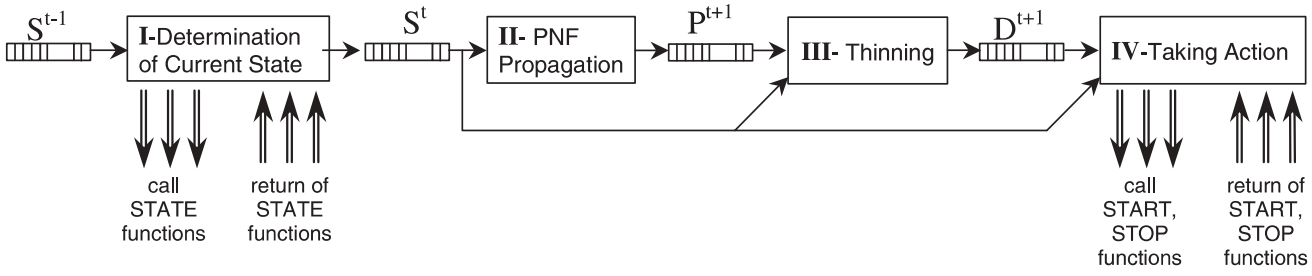


Fig. 10 A cycle of the interval script run-time engine

the current states of all the intervals that had already been computed. After this stage, the current state remains unchanged for the rest of the cycle.

II. *PNF propagation*: in this stage the engine tries to determine which actions the system can perform to satisfy the constraints in the next cycle $t+1$, noted as P^{t+1}, assuming that the state of the environment does not change. For this, we use a variation of the idea of PNF propagation where we consider for temporal expansion only those intervals that can be started or stopped,

$$V_i^{t+1} = \begin{cases} \mathrm{T}_m(S_i^t) & \text{if the interval i has START or STOP functions} \\ S_i^t & \text{otherwise} \end{cases}$$

By computing PNF propagation only on the nodes that can be started or stopped, the system only examines changes that can happen as a result of its own actions, ignoring all the possible changes in the environment that could be detected by the sensors. In practice, the run-time engine behaves as it was in a two-player game against the environment and its users. While in stage I it detects the 'play' of the environment, in this stage it explores all its own 'moves', eventually leading to a decision in stage III, and to action in stage IV.

Using the definition above, this stage then PNF-propagates the current state,

$$P^{t+1} = R\left(\mathrm{T}_m(S^t) \cap V^{t+1}\right) = R\left(V^{t+1}\right)$$

If the computation of $P^{t+1} = R(V^{t+1})$ detects a conflict, $P^{t+1} = \varnothing$, the engine tries to enlarge the space of possibilities by expanding all states regardless of the existence of START or STOP functions,

$$P^{t+1} = R(\mathrm{T}_m(S^t))$$

Although this normally handles most problems, there are situations where sensors report incorrectly and produce a state with no possible solutions. In those extreme cases, the engine simply time-expands the current state without computing the restriction,

$$P^{t+1} = \mathrm{T}_m(S^t)$$

III. *Thinning*: the result of stage II is normally too big and undetermined, although it contains all the globally consistent solutions. To have a more specific forecast of the next stage without doing search, we apply a heuristic based on the principle that minimal system action is desirable: the current state of an interval should remain the same unless it contradicts the result of stage III. This is accomplished by taking a special intersection operation between the forecast result of the actions in the next state P^{t+1} and the current state S^t. For each node the special intersection is computed by

$$S_i^t \cap P_i^{t+1} = \begin{cases} S_i^t \cap P_i^{t+1} & \text{if } S_i^t \cap P_i^{t+1} \neq \varnothing \\ P_i^{t+1} & \text{otherwise} \end{cases}$$

As seen in the formula, this special intersection operation simply computes a component domain where the PNF-value of each interval is exactly the intersection of its current value and the possible globally consistent values it can assume if action(s) is (are) taken by the system. If the intersection is empty, it means that the current state is incompatible with the result of any globally consistent action of the system, and therefore, the state of the node has to change as represented in P_i^{t+1}. If the intersection is non-empty, we have a node whose current state (or a fraction of it) is possibly compatible with the predicted results of possible actions. Following the minimal change heuristics, the thinning process makes the goal of the engine to keep the PNF-state of this node i to be the intersection of the current state S_i^t and the forecast P_i^{t+1}.

The component domain that results from the component-wise intersection is then passed through PNF restriction to assure that there are solutions and to remove impossible states, yielding the desired state D^{t+1}

$$D^{t+1} = R\left(S^t \cap P^{t+1}\right)$$

If the computation of $D^{t+1} = R\left(S^t \cap P^{t+1}\right)$ yields a state with no solutions we ignore the intersection,

$$D^{t+1} = P^{t+1}$$

In practice ignoring the intersection normally prevents any action to be taken since the states of P^{t+1} tend to be not as small as required to call start and stop functions as described next. Notice that it is not necessary, in this case, to apply the PNF restriction to P^{t+1}, since this was already been done in the previous stage and restriction is maximal.

IV. *Taking action*: the result of thinning, D^{t+1}, is compared to the current state S^t, and START and STOP functions are called if the need to change the state of an interval is detected. To choose among equally possible alternatives such as in the case of Fig. 8, we use a heuristics that makes the system more willing to start an action than to stop one. Actions are performed by the system according to the following table:

$x \subseteq S_i^t$	D_i^{t+1}	action
F	N, PN	START
N	P	STOP
F	P	STOP

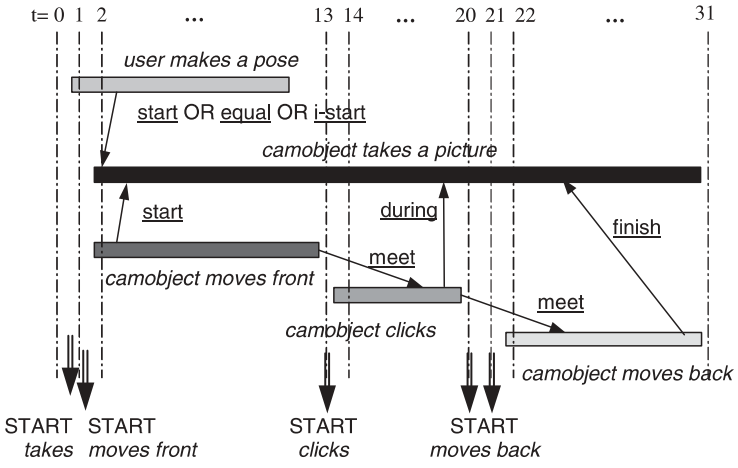
For example, if the current state of interval i can be future, $F \subseteq S_i^t$, and the desired state is *now*, $N = D_i^{t+1}$, then the START function of the interval is called. Notice that actions are started both if the desired state is N or PN, while they can be stopped only if the desired state is exactly P. In practice, this simple distinction is sufficient to make more likely the starting than the stopping of an action.
Notice also that the START function is never called if the current state of its corresponding interval does not contain the *future* state. That is, an interval is started only if there is a possibility that interval is in the *future* state. However, a STOP function is sometimes called even if the current state of the interval does not contain the *now* state. In this case we employ a mechanism by which if a STOP function is called and the state is exactly *future*, then the state of the interval is automatically set to *past*, without actually calling its STOP function.

To perform the search for globally consistent solutions instead of the thinning process of stage III, this stage would be substituted by the search of all solutions and by a mechanism to arbitrary choose one solution among them (for instance, by considering the first solution found). As discussed in Pinhanez [13], it is possible to use arc-consistency to speed up considerably such search. In practice, the run-time engine with the minimal change heuristics described above has been able to run all our applications without halting, deadlocking, or flip-flopping.

Let us examine the behavior of the run-time engine in the example of Fig. 8 where intervals A and B are constrained by an equal relation and their initial states are PF and N, respectively. To simplify notation, let us assume that $t = 0$ and the current state determined at stage I is $S^0 = (PF, N)$, where PF corresponds to the state of A and N to the state of B. As shown in Fig. 9, the result of PNF propagation is $P^1 = (PN, PN)$. The first attempt of thinning the result of PNF propagation yields $S^0 \cap P^1 = (P, N)$, which is not consistent, $D^1 = R\left(S^0 \cap P^1\right) = R((P, N)) = \varnothing$. By the recovery heuristics, we define the desired state to be $D^1 = (PN, PN)$. According to the stage IV of the run-time engine, only the START function of A would be called. The choice of starting A over stopping B is, in this case, an arbitrary by-product of the action taking heuristics of stage IV that tends to favor starting actions over stopping them.

5.4 Example of a basic run without conflicts

Figure 11 shows an example of a run of the interval script of *It* described in Sect.3. In the first instant of the run $t = 0$, we assume that the state of all intervals is F. Because *user makes a pose* is a sensor, i.e. without START/STOP functions, its PNF-state F is not expanded to NF during the temporal expansion process of stage II. Although the result of time expansion would allow some of the action intervals to be also in the N state, the F state of *user makes a pose* constrains all other intervals to be in the F state. This is detected by the result of the PNF propagation process, P^1, which keeps the state of the system as it is. Given that $D^1 = P^1$, no action is taken.



interval	t=0			t=1			t=2			t=13			t=14			t=20			t=21			t=22			t=31		
	S^0	P^1	D^1	S^1	P^2	D^2	S^2	P^3	D^3	S^{13}	P^{14}	D^{14}	S^{14}	P^{15}	D^{15}	S^{20}	P^{21}	D^{21}	S^{21}	P^{22}	D^{22}	S^{22}	P^{23}	D^{23}	S^{31}	P^{32}	D^{32}
user makes a pose	F	F	F	N	N	N	N	N	N	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
camobject moves front	F	F	F	F	N	N	N	N	N	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
camobject clicks	F	F	F	F	NF	F	F	NF	F	F	N	N	N	PN	N	P	P	P	P	P	P	P	P	P	P	P	P
camobject moves back	F	F	F	F	NF	F	F	NF	F	F	NF	F	F	NF	F	F	N	N	F	N	N	N	PN	N	P	P	P
camobject takes a picture	F	F	F	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	P	P	P

Fig. 11 Example of a run of the interval script described in Sect. 3

In the next instant of time, $t = 1$, the interval *user makes a pose* is detected, as reported by the *N* value coming from its STATE function. When PNF propagation is run, the fact that *user makes a pose* and *camobject takes a picture* should start together constrains the desired PNF-state (D^2) of the latter to be *N*, generating a call for the START function of *camobject moves front*. Notice also that in this situation, the propagated PNF-state (P^2) of *the camobject clicks* interval (as well as the state of *camobject moves back*) is *NF*, meaning that they may either be occurring or may occur in the future. However, they are not triggered because the thinning heuristics prevents change in intervals whose propagated PNF-state (in this case, *NF*) contains the current state (*F*). In the next instant, $t = 2$, we assume that *camobject moves front* successfully started and therefore no further action is required. The system remains in this state up to $t = 12$.

When $t = 13$, *camobject moves front* has finished and therefore has the *P* state. Because of the meet constraint, *camobject clicks* should start immediately. Notice that the result of PNF propagation, P^{14}, shows exactly that configuration and the appropriate action is taken. In $t = 14$ the desired state is reached for the interval.

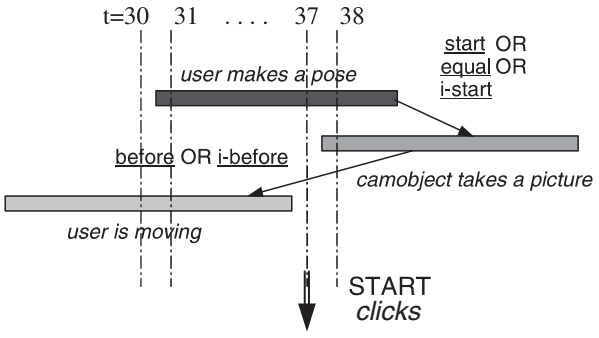
When *camobject clicks* finishes at $t = 20$ we have a similar situation where the START function of *camobject moves back* is called. However, this time we assume that, for some reason, the camera does not start moving back in the next instant of time, $t = 21$. As shown in Fig. 11, the discrepancy between the current state *F* (S^{21}) and the desired state *N* (D^{22}) persists, generating a second call for the START function of *camobject moves back*. At $t = 23$ the interval starts running and the situation is kept until the interval ends. At $t = 31$ the *P* state of *camobject moves back* is detected by the STATE function of *camobject takes a picture* (as described in the next section), and all the intervals go to the *P* state.

5.5 Run with conflicts

An example of a run with conflicts is show in Fig. 12 for the same interval script. Here, we consider the case where the intervals *camobject takes a picture* and *user is moving* are defined as mutually exclusive and a conflict situation arises when both are expected to happen.

At $t = 31$, a conflict is detected by the PNF propagation stage since there is no way to satisfy the requirements that *camobject takes a picture* should start together with *user makes a pose* and that *camobject takes a picture* can not happen while *user is moving* is occurring. In this situation the first level of recovery succeeds and finds a set of states that is compatible with the constraints as shown in Fig. 12. Basically, the system keeps the current situation as it is and does not start *camobject takes a picture*. Later, when $t = 37$, *user is moving* finishes, and since *user makes a pose* is still happening, *camobject takes a picture* is started.

As we see, the run-time engine of interval scripts was designed to avoid halts by searching for feasible states with the least amount of change. This strategy is not guaranteed to succeed but has proved to be robust enough to run quite complex structures. We are currently working on better methods to overcome contradictions such as keeping a history of previous states for backtracking purposes (such as those described in Pinjanez



interval	t=30			t=31				t=37			t=38		
	S^{30}	P^{31}	D^{32}	S^{31}	P^{32}	P^{32}	D^{32}	S^{37}	P^{38}	D^{38}	S^{38}	P^{39}	D^{39}
user makes a pose	F	F	F	N	N	**N**	N	N	N	N	N	N	N
camobject takes a picture	F	F	F	F	∅	**F**	F	F	N	N	N	PN	N
user is moving	N	N	N	N	N	**N**	N	P	P	P	P	P	P

Fig. 12 Example of a run with conflict

[13]) and devising mechanisms to detect and isolate the source of conflicts.

6 The interval scripts language (ISL)

The run-time engine described in the previous section has been implemented and used in the interactive environments described in Sect.7. For programming purposes, we have defined a text-based language called the *interval scripts language*, or *ISL,* that provides a simple way to describe an interval script. In the this section we show a programming example of ISL.

Although there are multiple ways to implement the concepts described in this paper, we chose to develop a text-based language containing the descriptions of the intervals and the temporal constraints between them. In our implementation, the file containing the language statements is run through an ISL compiler that produces a set of C++ files. The C++ files can be compiled using any standard compiler and executed at run-time using specially defined functions defined in another C++ file. It should be noticed that ISL is a particular implementation of the paradigm of interval scripts. As discussed later in Sect.8, it is possible to program with intervals and temporal constraints using, for instance, a graphical interface.

This section covers just the basic syntax and semantics of ISL, using as example the implementation of the interaction described in Sect.3. More details about the semantics of the language and its complete grammar can be found in Pinhanez [13].

6.1 Encapsulating code in intervals

We start our description of ISL by examining how the language communicates with low-level programming languages that provide basic functions and device access. In ISL, we simply allow the inclusion of C++ code directly into the script.

```
"camobject moves front" =
{
START:  execute [> camobject.MoveFront(1.0); <];
STOP:   execute [> camobject.StopMoving(); <];
STATE:  set-state pnf-expression
        [> (camobject.isMoving()?_N_:P_F) <];
}.
```

Fig. 13 Interval script describing the forward movement of the *camobject* object.

To begin our example, we show how to connect an interval in ISL to low-level C++ functions in the context of the example described in Sect.3. Figure 13 shows the definition of the interval *camobject moves front* in ISL where a camera-like CG object referred to as *camobject* moves to the front of the screen. The definition of the interval is comprised between curly brackets containing the definition of the interval's basic functions. To include C++ code we use the command execute followed by the symbols '[>' and '<]'. For instance, when the START function is called, it executes the C++ code between those symbols, that is, a C++ method called `MoveFront` for the object `camobject` with parameter value '`1.0`'. The STOP function is defined in a similar way.

The definition of the STATE function is slightly different. In this case, the function is defined to set the state of the interval to be equal to the state returned by the C++ code inside the special symbols. In the case depicted in Fig. 13, a method of the object `camobject` determines if the computer graphics camera is moving or not. If true, the state of the interval is set to *now*, referred

```
"camobject moves front" = { ... }.
"camobject clicks" = { ... }.
"camobject moves back" = { ... }.
better-if "camobject moves front" meet "camobject clicks".
better-if "camobject clicks" meet "camobject moves back".
```

Fig. 14 Interval script describing three intervals and two constraints among them

in C++ code by the special constant _N_; otherwise, the state is set to be *past OR future*, symbolised by `P_F`.

6.2 Describing constraints

Using the example of Sect.3, we examine how constraints between intervals are defined in ISL. Figure 14 shows the script corresponding to the two constraints imposed over the three intervals *camobject moves front*, *camobject clicks* and *camobject moves back*. The script first defines the intervals (here omitted for clarity), followed by two statements establishing the temporal constraints between the intervals. Notice the syntax better-if that was chosen to imply that this is a constraint that the system will try to enforce but that is not guaranteed to occur.

6.3 Defining on previous intervals

A key feature of ISL is the possibility of defining a new action or world state solely based on other intervals. With this we can create hierarchies, abstract concepts, and develop complex, high-level scripts. Continuing with the example of Sect.3, we define the interaction between the user and the camera-like object as follows: when the user makes a pose (as detected by a computer vision system), the camera takes a picture, an action that is composed of moving the camera forward, closer to the user, clicking, and returning to its original position. Figure 15 shows the program corresponding to this interaction.

Initially the interval *user makes a pose* is defined as before, by a reference to C++ code that communicates with the vision system. Notice that this interval corresponds to a sensor and therefore has no START or STOP functions. To accomplish the clicking of the camera and the taking of the picture, the interval

Fig. 15 Scripting based on previously defined intervals

```
"user makes a pose" =
{
STATE:    set-state pnf-expression
      [> (user.isMoving()?_N_:P_F) <];
}.
"camobject takes a picture" =
{
STOP:     if current-state is NOW
          tryto start "camobject moves back".
STATE:    if "camobject moves front" is NOW
             OR "camobject clicks" is NOW
             OR "camobject moves back" is NOW
          set-state NOW
          else if "camobject moves back" is PAST
             set-state PAST
             endif
          endif.
}.
better-if "user makes a pose" start OR equal OR i-start "camobject takes a
picture".
better-if "camobject moves front" start "camobject takes a picture".
better-if "camobject clicks" during "camobject takes a picture".
better-if "camobject moves back" finish "camobject takes a picture".
"user is moving" = { ... }.
better-if "user is moving" before OR i-before "camobject takes a picture".
```

camobject takes a picture is defined, based on its three component intervals. First, a STOP function is defined so if the interval is happening and it is made to stop, it first makes the camera-like object move back to its original position before actually stopping the main interval. Notice that, as showed in this example, stopping an interval may require actions to be executed and, therefore, some time is required before an interval actually stops. If the STOP function was not explicitly defined, the interval still could be stopped but in that case the interval *camobject moves back* would not necessarily be executed.

This is also the case of the STATE function that defines the computation of the current state of the interval *camobject takes a picture*. As shown in Fig. 15, if any of the component actions is happening, the state of the interval is set to *now*. If the action of moving back the camera has occurred, that is, its current state is *past*, then the interval *camobject takes a picture* is determined to have already happened and its own state is set to *past*. Notice that, according to this definition, when the STOP function of the interval is called, the interval is not considered finished until the object completely moves back to its original position.

Finally, following that example, we want that whenever *user makes a pose* starts happening, the camera takes the picture. This is established by the constraint start OR equal OR i-start that forces the two intervals to start together. Also, we impose constraints that relate the interval *camobject takes a picture* to its three constituents. Notice that, it is not necessary to explicitly define a START function for *camobject takes a picture*, since the constraint start between the intervals *camobject moves front* and *camobject takes a picture* establishes that the triggering of each interval immediately starts the other. And, completing the example of Sect.3, the interval *user is moving* is defined and constraints are imposed that make it mutually exclusive to the interval *camobject takes a picture*.

The Interval Scripts Language includes many other features that have shown to be extremely helpful in the programming of the interactive environments described in Sect.7. See Pinhanez [13] for a detailed description of those features.

7 Working with interval scripts

Evaluating a programming method is always difficult. Although we have not performed formal studies on how easy it is for programmers to understand and use the paradigm and the ISL language, we describe three systems here built using interval scripts as evidence towards our belief that the language is an improvement over current paradigms and programming methods for interactive environments and agents. We do not see how these systems (especially the last two) could have been programmed using event-loops or state-machines in a reasonable amount of time and without major problems in debugging and maintenance. In fact, all the control structures of the systems described here were developed in very short periods of time. Based on these experiences, we believe that our main objectives when designing intervals scripts were achieved, that is, that the paradigm provides expressiveness and simplicity beyond current programming methods.

The first two cases of interactive environments developed with interval scripts, *SingSong* and *It/I,* are interactive theatrical performances where human and autonomous computer-graphics actors interact following a story. They constitute what is called *computer theater*, a term referring to live theatrical performances involving the active use of computers in the artistic process (see Pinhanez [43]). The third system, *It*, is an interactive art installation where interval scripts were used not only to program the environment but also its inhabitant computer characters and agents. Although all three interactive environments correspond to entertainment/art applications, it is reasonable to expect that similar development and performance results would be obtained in other scenarios of interactive environments such as computer-controlled offices or homes. In fact, given the complexity of everyday life, we expect that a powerful programming paradigm such as the interval scripts is even more necessary in practical cases to achieve similar behaviour complexity. For instance, one of the authors of this paper, Claudio Pinhanez, has participated in the design and development of a prototype of a computer-augmented office environment (see Lai et al. [5] for a description of the system). In that project, it was observed that the lack of a more complex programming model constrained the diversity and adaptability of the behaviours of the space to simple reactive mechanisms.

7.1 *SingSong:* an interactive performance environment

SingSong was our first experience with interval scripts [17]. *SingSong* is composed of a large video screen that displays four computer graphics-animated characters that can "sing" musical notes (as produced by a MIDI synthesizer). A camera watches the user or performer determining the position of his/her head, hands, and feet. The body position is recovered by the software *pfinder* developed at the MIT Media Laboratory [44].

All the interaction is physical, non-verbal: the user or performer gestures and the CG-characters sing notes and move. *SingSong* was produced and performed in the summer of 1996. Although it is a short play of about four minutes, the interval script involves around 200 intervals, including a great deal of low-level control of I/O devices. It took about two days to write and debug the controlling program. Figure 16 shows a sequence of scenes from *SingSong*.

SingSong used the first version of the concept of interval scripts, described in Pinhanez et al. [17]. The

Fig. 16 Scenes from *SingSong*

SingSong version of interval scripts already included the important idea of separating the desired action from the actual action. However, in that version, the idea was implemented using two implicitly defined intervals instead the simpler use of START, STOP, and STATE functions (see Pinhanez et al. [17]). The use of two intervals required a more complicated model of the situation, making scripting more difficult than in the subsequent versions.

The main problem with the *SingSong* version of the interval scripts idea was that the script had to be written explicitly into C++ code. Although the interval scripts paradigm helped to avoid extremely complex and undebuggable control structures, after a while it became quite difficult to read the C++ file. This realisation prompted us for the development of the Interval Scripts Language introduced in the previous section.

7.2 *It/I:* a computer theatre play

Following *SingSong* we decided that it was necessary to develop a more comprehensive test for interval scripts. Particularly, we were interested in testing the robustness of the interval scripts paradigm in conditions where

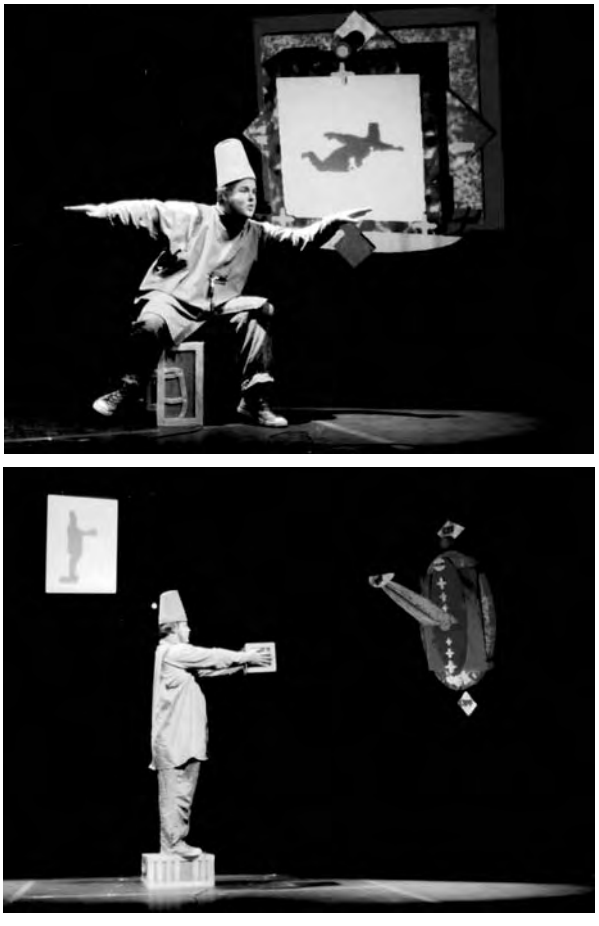


Fig. 17 Scenes from *It/I*. The computer graphics objects on the screen are autonomously controlled by the computer character ***It***

difficulties in scripting or run-time failures would be critical.

With these ideas in mind, one of the authors of this paper, Claudio Pinhanez, wrote the computer theater play *It/I*. The play is a pantomime where one of the characters, *It*, has a non-human body composed of CG-objects projected on screens (see Fig. 17). The objects are used to play with the human character, *I*. *It* "speaks" through images and videos projected on the screens, through sound played on stage speakers, and through the stage lights.

The play is composed of four scenes, each being a variation of a basic cycle where *I* is lured by *It*, is played with, gets frustrated, quits, and is punished for quitting (see Pinhanez and Bobick [14] for a detailed script of the play). It is important to note that any individual scene of *It/I* has a complexity level beyond previous full-body interactive systems [35, 45]. The play was produced in the summer/fall of 1997 and performed six times for a total audience of about 500 people. Each performance, lasting 40 minutes, was followed by an explanation of the workings of the computer-actor. After that members of the audience were invited to go up on stage and play the second scene from the play (see Pinhanez and Bobick [14]).

For performance safety, we implemented each scene of *It/I* in a separate script, generating four different executable files that were loaded in the beginning of the corresponding scene. Table 2 summarizes the composition of the interval scripts of each scene of *It/I*, displaying the total number of intervals (i.e. the number of nodes of the PNF-network) and the number of constraints. As can be seen in the table, we have more than 100 nodes per scene with an average of 80 constraints between them.

Although simple, the design of the run-time engine of the *It/I* version of interval scripts was not able to take full advantage of the prediction powers of constraint propagation of the current version of the run-time engine, where the idea of computing the desired state by PNF propagation and thinning (as described in Sect.5) considerably improved the robustness of the system.

7.3 *It:* an Interactive interactive installation

After *It/I,* we developed an interactive art installation called *It* where users can experience the story of the play without having previously watched it. In *It,* the user enters the area, is approached by the character *It,* starts playing with the CG-objects as *I* did in the original play, and like *I,* is not allowed to leave the space or stop playing. The installation was finished in the spring of 1999 and demonstrated successfully during two open houses held at the MIT Media Laboratory.

In *It* we employed the version of ISL introduced in Sect.6, and the run-time engine discussed in Sect.5. The ISL was used both to program the environment and its story and the computer agents that control the characters and agents that inhabit it. In fact, the control structure of *It* is distributed among four agents: the *story* agent, that controls the basic development of the environment; the *It* character and the *I* character agents, corresponding to the two main characters of the interactive story; and the *light designer* agent, that controls all the lighting effects of the environment.

In Table 3 we summarise the composition of the interval scripts used for the different agents of *It*. Table 3 shows that interval scripts can be used to construct and control large and complex collections of actions. For example, the *It* character agent employs 224 nodes for a total of more than 350 constraints. During run-time, we apply PNF propagation on this network at the rate of 20 Hz without any visible impact on the performance of the *SGI R10000* machine used in the installation. In fact, all four agents run simultaneously in one single machine, consuming at most 15% of the CPU time, including the communication overhead.

8 Future directions

Although the interval scripts language has been fully implemented and intensively tested in three different projects, there is still work to be done in many directions. First, it would be interesting to get the system to a state where it could be released to other researchers and programmers interested in using interval scripts to build interactive systems. In this context it becomes possible to evaluate the difficulty in learning the language and which features different types of programmers would like to have added to the language.

In particular, an area we want to investigate is the best interface for the interval scripts programming paradigm: text-based or graphical. In the case of a graphical interface, a possible approach is to allow the user to

Table 2 The composition of the interval scripts for the different scenes of *It/I*

	Total number of nodes	Total number of constraints*
scene I	120	88
scene II	80	70
scene III	92	66
scene IV	115	97

(*) including implicit constraints (see [13]).

Table 3 The composition of the interval scripts for the different agents in *It*

	No. of nodes	Total no. of constraints
story	59	101
It character	224	355
I character	8	8
light designer	94	133

construct temporal diagrams similar to Fig. 4. However, since we allow disjunction of temporal constraints, there are always multiple possibilities for the visual arrangement of the intervals and therefore the visualisation may create more confusion than to provide help. Nevertheless, we have always found very useful to draw temporal diagrams when programming in ISL.

We would also like to integrate into the interval scripts paradigm mechanisms that represent actions in forms that the machine can reason about. Currently, the run-time engine can only control the execution of scripts but has no mechanisms to create action plans beyond the simple 1-step look-ahead performed in stage II. Comparing, for instance, to the *Hap* system developed by Loyall and Bates [46], the current interval scripts control system is basically reactive. Although it can run and control interactive plans with very complicated temporal structures, it has no way to construct plans from goals beyond the immediate one-step future. This limitation is partially due to the ideas employed to reduce the complexity of temporal constraint propagation as described in Sect.4.

We have considered different mechanisms to overcome this limitation. A basic idea is to perform multi-step look-ahead, by considering all possible different combinations of sensor input. In Pinhanez [13] we show that it is possible to maintain multiple threads of sensor values, under some simplifying assumptions, because a majority of the generated threads become inconsistent and can be discarded after some cycles, due to contradiction with real sensor values. Since that work was focused on action recognition, this was done in the context of keeping multiple possibilities for the events in the past. Similarly, we could use multiple threads of sensor values to search the future for an activation pattern that achieves a determined goal expressed as a combination of PNF-states for the different intervals of a script. To accomplish this, it is necessary to incorporate mechanisms that forecast the likely duration of actions and the likely delays in starting and stopping them, so the systematic PNF-style exploration of the future becomes feasible.

9 Conclusion

In this paper, we propose the interval scripts paradigm for programming interactive environments that is based on declarative programming of temporal constraints. Unlike previous constraint-based systems, we allow the use a strong temporal algebra with mutually exclusive intervals. Among other things, we have created a paradigm that adequately handles the representation and control of some of the main issues in the programming of actual interactive environments: delays, continuity, and history of actions and states; mutually exclusively actions; stopping of long actions; and recovery from conflicts and contradictions. Through examples we have shown that our paradigm allows good expressiveness and facilitates the management of context in an interactive environment.

This paper significantly extends the initial proposal of interval scripts outlined in Pinhanez et al. [17]. Among other innovations, the run-time engine described in this paper is able to actively forecast future states that satisfy the temporal constraints, allowing the search for a consistent set of action triggerings. Also, the definition of the Interval Scripts Language greatly simplifies the use of temporal constraints in the programming process.

From the experience acquired in the use of the paradigm for the implementation of three different projects in the entertainment/art domain, we believe that programming systems incorporating the interval scripts paradigm can significantly ease the design and building of interactive systems in the future.

Acknowledgements *SingSong*'' was written, directed, and performed by Claudio Pinhanez. *It*/*I* was written and directed by Claudio Pinhanez, and produced by Aaron Bobick; the crew was composed by John Liu, Chris Bentzel, Raquel Coelho, Leslie Bondaryk, Freedom Baird, Richard Marcus, Monica Pinhanez, Nathalie van Bockstaele, Maria Redin, Alicia Volpicelli, Nick Feamster, and the actor Joshua Pritchard. *It* was designed and implemented by Claudio Pinhanez based on the elements produced for *It*/*I*.

Claudio Pinhanez was supported in different stages of this research by the scholarship from CNPq, process number 20.3117/89.1; by the DARPA contract DAAL01-97-K-0103; by the MIT Media Laboratory; and by the MIT Japan Program through the Starr Foundation. *SingSong* was sponsored by the Media Integration and Communication (MIC) laboratory of the Advanced Technology Research (ATR) laboratories in Kyoto, Japan. *It*/*I* and *It* were sponsored by the Digital Life Consortium of the MIT Media Laboratory.

References

1. Pausch R, Snoddy J, Taylor R, Watson S, Haseltine E. Disney's Alladin: first steps toward storytelling in virtual reality. In: Proceedings of SIGGRAPH'96. 1996, pp 193–203
2. Bobick A, Intille S, Davis J, et al. The KidsRoom: a perceptually-based interactive immersive story environment. Presence: Teleoperators and Virtual Environments 1999; 8(4): 367–391
3. Tosa N. Expression of emotion, unconsciouness with art and technology. In: Hatano G, Okada N, Tanabe H (eds) Affective minds, Elsevier, 2000, pp 183–205
4. Pinhanez C, Davis J, Intille S, et al. Physically interactive story environments. IBM Systems Journal 2000; 39(3&4): 438–455
5. Lai J, Levas A, Chou P, Pinhanez C, Viveros M. BlueSpace: Personalizing workspace through awareness and adaptability. International Journal of Human Computer Studies 2002; 57(5): 415–428
6. Crowley JL, Coutaz J, Berard F. Things that see. Communications of the ACM 2000; 43(3): 54–64
7. Raskar R, Welch G, Cutts M, Lake A, Stesin L. The office of the future: a unified approach to image-based modeling and spatially immersive displays. In: Proceedings SIGGRAPH'98. Orlando, FL, 1998, pp 179–188
8. Abowd GD. Classroom 2000: an experiment with the instrumentation of a living educational environment. IBM Systems Journal 1999; 38(4): 508–530
9. Mynatt ED, Essa I, Rogers W. Increasing the opportunities for aging in place. In: Proceedings ACM conference on universal usability (CUU'00), Arlington, VA, 2000

10. Pentland A. Smart rooms. Scientific American 1996; 274(4): 68–76
11. Shaw C, Green M, Liang J, Sun Y. Decoupled simulation in vrtual reality with the MR Toolkit. ACM Transactions on Information Systems 1993; 11(3): 287–317
12. Pausch R, Burnette T, Capeheart AC, et al. A brief architectural overview of Alice, a rapid prototyping system for virtual reality. IEEE Computer Graphics and Applications 1995
13. Pinhanez CS. Representation and recognition of action in interactive spaces. PhD Thesis, Media Arts and Sciences Program, MIT, Cambridge, MA, 1999
14. Pinhanez CS, Bobick AF. It/I: A theater play featuring an autonomous computer character. Presence: Teleoperators and Virtual Environments 2002; 11(5): 536–548
15. Allen JF. Maintaining knowledge about temporal intervals. Communications of the ACM 1983; 26(11): 832–843
16. Vilain M, Kautz H. Constraint propagation algorithms for temporal reasoning. In: Proceedings of AAAI'86, Philadelphia, PA, 1986, pp 377–382
17. Pinhanez CS, Mase K, Bobick AF. Interval scripts: a design paradigm for story-based interactive systems. In: Proceedings of CHI'97, Atlanta, GA, 1997, pp 287–294
18. Coen MH. Building brains for rooms: designing distributed software agents. In: Proceedings of IAAI'97, Providence, CT, 1997, pp 971–977
19. Director's User Manual, MacroMind Inc. 1990
20. MAX Getting Started Manual, Opcode, 2002
21. Roads C. The computer music tutorial. MIT Press, Cambridge, MA, 1996, pp 969–1016
22. Rowe R. Interactive music systems. MIT Press, Cambridge, MA, 1993
23. Rollings A, Morris D. Game architecture and design. Coriolis Group, Scottsdale, AZ, 2000
24. Buchanan MC, Zellweger PT. Automatic temporal layout mechanisms. In: Proceedings of ACM Multimedia'93, Anaheim, CA, 1993, pp 341–350
25. Hamakawa R, Rekimoto J. Object composition and playback models for handling multimedia data. In: Proceedings of ACM multimedia'93,Ahaheim, CA, 1993, pp 273–281
26. van Rossum G, Jansen J, Mullender K, Bulterman D. CMIFed: a presentation environment for portable hypermedia documents. In: Proceedings of ACM multimedia'93, Anaheim, CA, 1993
27. Vazirgiannis M, Kostalas I, Sellis T. Specifying and authoring multimedia scenarios. IEEE Multimedia 1999; 6(3): 24–37
28. Bailey B, Konstan JA, Cooley R, Dejong M. Nsync – a toolkit for building interactive multimedia presentations. In: Proceedings of ACM multimedia'98 Bristol, UK, 1998, pp 257–266
29. Jourdan M, Layaida N, Roisin C, Sabry-Ismail L, Tardif L. Madeus, an authoring environment for interactive multimedia documents. In: Proceedings of ACM multimedia'98 Bristol, UK, 1998, pp 267–272
30. Selcuk KC, Prabhakaran B, Subrahmanian VS. CHIMP: a framework for supporting distributed multimedia document authoring and presentation. In: Proceedings of ACM multimedia'96, Boston, MA, 1996, pp 329–339
31. Agamapolis S, Bove Jr. VM. Multilevel scripting for responsive multimedia. IEEE Multimedia 1997; 4(4): 40–50
32. Wirag S. Modeling of adaptable multimedia documents. In: Proceedings of European workshop on interactive distributed multimedia systems and telecommunications services, Darmstadt, Germany, 1997
33. André E, Rist T. Coping with temporal constraints in multimedia presentation planning. In: Proceedings of AAAI'96, Portland, OR, 1996, pp 142–147
34. Kautz HA, Ladkin PB. Integrating metric and qualitative temporal reasoning. In: Proceedings of AAAI'91, Anaheim, CA, 1991, pp. 241–246
35. Krueger MW. Artificial reality II. Addison-Wesley 1990
36. Bederson BB, Druin A. Computer augmented environments: new places to learn, work and play. Ablex, Norwood, NJ, 1995
37. Dey AK, Salber D, Abowd GD. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction Journal 2001; 16(1)
38. Pinhanez CS, Bobick AF. Human action detection using PNF propagation of temporal constraints. In: Proceedings of CVPR'98, Santa Barbara, CA, 1998, pp 898–904
39. Meiri I. Combining qualitative and quantitative constraints in temporal reasoning. In: Proceedings of AAAI'91, Anaheim, CA, 1991, pp 260–267
40. Pinhanez CS, Bobick AF. PNF propagation and the detection of actions described by temporal intervals. In: Proceedings of DARPA image understanding workshop, New Orleans, LA, 1997, pp 227–233
41. Dechter R. From local to global consistency. Artificial Intelligence 1992; 55(1): 87–107
42. Mackworth AK. Consistency in networks of relations. Artificial Intelligence 1977; 8(1): 99–118
43. Pinhanez CS. Computer theater. In: Proceedings of eighth international symposium on electronic arts (ISEA'97), Chicago, IL, 1997
44. Wren C, Azarbayejani A, Darrell T, Pentland A. Pfinder: Real-time tracking of the human body. IEEE Transactions Pattern Analysis and Machine Intelligence 1997; 19(7): 780–785
45. Maes P, Darrell T, Blumberg B, Pentland A. The ALIVE system: full-body interaction with autonomous agents. In: Proceedings of computer animation'95, Geneva, Switzerland, 1995
46. Loyall AB, Bates J. Hap: A reactive, adaptive architecture for agents. Carnegie Mellon University, Technical Report CMU-CS-91-147, Pittsburgh, PA, 1991