Scripting Interactive Environments with Interval Scripts

Claudio Pinhanez
MIT Media Laboratory
Current Address:
IBM TJ Watson Research Center
30 Saw Mill River Road
Hawthorne – NY 10532 – USA
pinhanez@us.ibm.com

Aaron Bobick
Georgia Institute of Technology

College of Computing
801 Atlantic Dr.
Atlanta - GA 30332 – USA
bobick@cc.gatech.edu

ABSTRACT

In this paper we present a new paradigm for high-level scripting of computer characters and stories in interactive environments called *interval scripts*. In this paradigm, the actions of characters and users are associated with temporal intervals and scripting is accomplished by establishing temporal constraints between the intervals. Unlike previous constraint-based interaction 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 slowness of strong temporal algebras, we propose a method — *PNF propagation* — that projects the network implicit in the script into a simpler, 3-valued (past, now, future) network where constraint propagation can be approximately computed in linear time. The paper introduces the interval script paradigm by referring to its current implementation as a text-based language that was used to build three large-scale, computer-vision-based interactive installations. The success on implementing these three very complex projects is presented as evidence that interval scripts are a simpler and more expressive scripting method than any currently used system for scripting interaction in interactive environments.

RESUMO

Este artigo apresenta um novo paradigma — *roteiros temporais* — para a roteirização de alto nível de personagens e estórias computacionais em ambientes interativos. Neste paradigma as ações de personagens e usuários são associadas a intervalos temporais, enquanto que a progressão temporal do roteiro é estabelecida através de restrições temporais entre os intervalos. Ao contrário de outras linguages de interação baseadas em restrições, este paradigma emprega uma álgebra temporal forte (baseada na álgebra de intervalos de Allen), capaz de expressar ações mutuamente exclusivas e de definir complexas estruturas temporais. A fim de evitar a lentidão normalmente associada a álgebras temporais fortes, emprega-se um método— *propagação PNF* — que projeta a rede definida implicitamente pelo roteiro em uma rede mais simples, onde os nós assumem somente três valores (past, now, future). Nesta rede é possível fazer a propagação de restrições em tempo linear. Este artigo apresenta o paradigma de roteiros temporais através de sua implementação atual como uma linguagem de programação que foi extensivamente usada na construção de três instalações computadorizadas de larga escala (baseadas em visão computacional). O sucesso obtido na implementação destes três projetos é apresentado como evidência de que roteiros temporais são uma maneira mais simples e expressiva de descrever interação em ambientes interativos do que qualquer outro método atualmente em uso.

1. INTRODUCTION

The research described in this paper proposes a method to script computer characters and interactive stories based on temporal constraints. Called *interval scripts*, this paradigm is based on the encapsulation of actions and states in temporal intervals constrained by the temporal primitives of Allen's algebra [2]. The objective is to provide the designer of an interactive system or complex computer graphics scenes with a tool that combines expressiveness and simplicity. Among the desired features, we included in interval scripts facilities for easy abstraction of actions; the ability to impose restrictions on what can happen (negative scripting); mechanisms to infer indirectly the occurrence of users' and characters' actions (see also [30]); and the capacity of recovery from errors and unexpected situations.

To accomplish this, we are developing a language based on temporal constraints that also includes some elements of traditional procedural methods. An interval script contains descriptions of how to activate and stop the actions of the characters, but the activation of the intervals is determined by verifying the current state of the system, comparing it with the desired state according to the constraints, and issuing commands to start and stop action as needed. Another characteristic of interval scripts is the de-coupling of the actual state from the desired state which allows for the introduction of methods for recovery from errors and increases the robustness of the system.

The idea of interval scripts has been implemented in an *interval script language* that we describe with some detail in this paper. Although the concept of interval scripts can have different implementations (as a graphics-user interface, for instance), in this paper we present all the concepts of the interval scripts using the language.

Interval scripts proved to be essential to manage the complexity of some large-scale projects of interactive, story-based spaces we have developed. In this paper we briefly describe three of these projects in order to provide the reader with a sense of the difficulties involved. Particularly, we do not believe that it would be possible to implement those systems in the allotted time in any other language or currently used paradigm. In fact that it was possible to build them at all is good evidence of the usefulness and appropriateness of our language.

This paper starts by reviewing the current scripting languages and identifying their shortcomings. In sections 3 and 4 we introduce the interval script language through some simple examples. The core of the paper is the description of the run-time engine architecture and the theory needed to allow efficient run of temporal constraint propagation algorithms, contained in section 5. We then explore in section 6 more complex constructions allowed by the interval script language. Section 6 contains description of the interactive environments that have been built using the paradigm.

2. PROBLEMS WITH CURRENT SCRIPTING TECHNIQUES

The idea of interval scripts was fueled by our dissatisfaction with the lack of methods for the integration of characters, stories, and I/O devices in interactive environments. As described by Cohen, systems to control interaction tend easily to become *"big messy C program(s)"* ([10], fig. 2). From the experience of *"The KidsRoom"* [7], 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 system grows faster than the complexity of the system.

2.1 State Machines and Event Loops

The most common technique for the scripting and control of interactive applications is to describe the interaction through state machines. This is the case of the most popular language for the developing of multimedia software, *Macromedia Director's Lingo* [1]. In *Lingo*, the interaction is described through the handling of events whose context is associated to specific parts of the animation with no provisions to handle the history of the interaction nor the management of story lines.

Video games are traditionally implemented through similar event-loop techniques. To represent history, the only resort is to use state descriptors whose maintenance tends to become a burden as the complexity increases. Also, the state-machine model lacks appropriate ways to represent the duration and complexity of human action: hidden in the structure is an assumption that actions are pin-pointed events in time (coming from the typical point-and-click interfaces those languages are designed for) or a simple sequence of basic commands.

In the *"The KidsRoom"* [7] the interaction is controlled by a system composed of a state machine where each node has a timer and associated events. A problem with this model is that it forces the designer to break the flow of the narrative into manageable states with clear boundaries. In particular, actions and triggered events that can happen in multiple scenarios have normally to be re-scripted for each node of the state machine, making incremental developing very difficult.

2.2 Constraint-Based Scripting Languages

The difficulties involved in the use of state machines and event loops have sparkled a debate in the multimedia research community concerning the applicability of constraint-based programming (starting with the works of Buchanan & Zelllweger [9] and Hamakawa & Rekimoto [14]) versus procedural descriptions (for example, [38]). In general, it is considered that constraint-based languages are harder to learn but more robust and expressive.

Bailey et. al. [3] 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, in particular due to its inability to represent non-acyclic structures, is also the major shortcoming of *Madeus* ([16]), CHIMP ([33]), ISIS ([19]), and TIEMPO ([40]). Other examples are *TBAG* [12], and Kakizaki's work on deriving animation from text [17].

2.3 Scripting of Computer Graphics Characters

The scripting of computer graphics characters and in particular of humanoids has attracted considerable attention in the computer graphics field. A group of researchers has worked with languages for scripting movements and reactions of characters, like Perlin [25], Kalita [18], and Thalman [35]. In particular, Perlin and Goldberg have developed a language, *Improv* [25], that nicely integrates the control of low-level control of geometry with middle-level primitives such as "move", "eat", etc. The major shortcoming of *Improv* is the lack of simple methods to synchronize and to temporally relate parallel actions.

Another line of research steams directly from Rod Brooks' works with autonomous robots [8]. The objective of this school is to create characters with their own behaviors and desires. Significant examples are Blumberg's behavior control system [6], Tosa's model for emotional control [36], and Terzopoulos work on the interaction of sensory and behavior mechanisms [37]. The problem with purely behavior-based characters is that they are enable to follow stories. In [27] Pinhanez argues that there is a significant difference between computer creatures and computer actors, the later being able to accept the high-level commands needed for the precise timing and developing of stories. Integration of high-level commands into behavior-based control still defies the research in the area.

Bates et. al. [4] created an environment composed of several modules that encompass emotions, goals *(Hap)*, sensing, language analysis, and generation. However, scripting in *Hap* imposes many restrictions in the ways parallel actions can be synchronized. Also, since planning happens at every cycle, it becomes difficult to use the system in real-time, interactive applications with multiple characters.

2.4 Scripting Interactive Environments

Starting with the pioneer work of Myron Krueger [20], the interest in building interactive environments, especially for entertainment, has grown in the recent years (see [5, 7] for a good review). In the case of virtual reality, it seems that the user-centered and exploratory nature of most interfaces facilitates the scripting the interface with state machines and event loops. There are few references of scripting systems for VR (for example [34]). A recent example is *Alice* [24], a language that allows rapid prototyping but has very few possibilities in terms of temporal structures.

In many interactive environments the control of the interaction is left to the characters themselves (seen as creatures living in a world). This is the case of *"ALIVE"* [22], where the mood of the CG-dog *Silus* commands the interaction; and also in *"Swamped"* [15], where an user-controlled chicken plays cat-and-mouse with a raccoon. In both cases, however, there is no easy way to incorporate dramatic structure into the control system as discussed above. In most of these cases, as well as in Galyean's work [13], it is necessary to "hack" the behavior structure in order to make a story flow.

3. A PROPOSAL: INTERVAL SCRIPTS

Using *interval scripts* to describe interaction was first proposed in [31]. The basic goal was to overcome the limitations of traditional scripting languages as described above. The work drew from a technique originally developed for human action recognition [30] where temporal constraint propagation is used to determine the occurrence of actions described as a collection of temporarily related sub-actions.

In interval scripts, all actions and states of both the users and the computer characters are associated to the temporal interval where they occur. However, the actual beginning and ending of the intervals are not part of the script. Instead, the script contains a description of the temporal relations that the intervals must obey during run-time. For example, to describe the situation where a CG character enters the image whenever the user makes a gesture, we associate the appearance of a CG character to an interval, and the user gesture to another. To accomplish the desired sequence of events, we put a temporal constraint stating that the end of the gesture interval should be immediately followed by the entrance of the character. During run-time, the interval script engine monitors the occurrence of the gesture interval. When it finishes, the engine solicits the entrance of the CG character in an attempt to satisfy the temporal constraint. Developing a system or scene with interval scripts is a classical case of programming by constraints. Notice that programming by constraints always requires a run-time engine that examines the current information and assigns new values for the different variables of a problem to satisfy the constraints.

To model the time relationships between two intervals we employ the interval algebra proposed by Allen [2]. The interval algebra is based on the 13 possible primitive relationships between two intervals that are summarized in fig. 1.

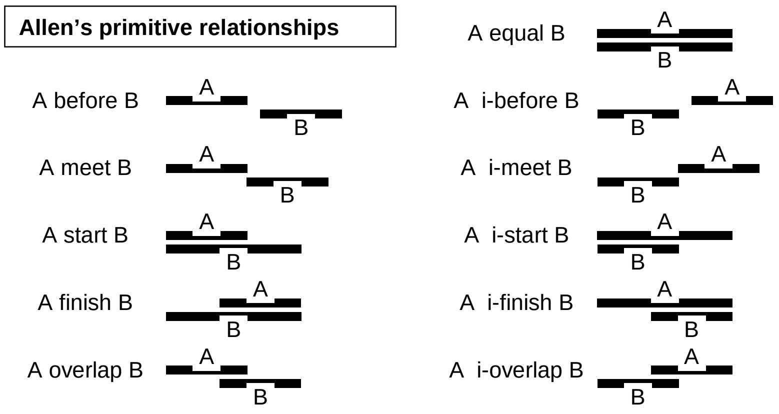


Figure 1. Allen's 13 primitive relationships between two time intervals.

Given two actions in the real world, their possible time relationship can always be described by a disjunction of the primitive time relationships. For instance the relation meet describes exactly the temporal constraint in the above example between the gesture interval and the interval associated to the entrance of the CG character. If the entrance of the character could start before the end of the gesture, the relation between the two intervals would be described by the disjunction overlap OR meet. Of course, in a real occurrence of the intervals, only one of the relationships actually happens.

We see several reasons to use Allen's algebra to describe relationships between intervals. First, no explicit mention of the interval duration or specification of relations between the intervals' starting and endings are required. Second, the existence of a time constraint propagation algorithm (describe in [2]) allows the designer to declare only the relevant relations, leading to a cleaner script. *Allen's path consistency algorithm* is able to process the definitions and to generate a refined version of the script containing only those relations and are consistent in time. Third, the notion of disjunction of interval relationships can be used to declare multiple paths and interactions in an story. Fourth, it is possible to determine whether an interval is or should be happening by properly propagating occurrence information from one interval to the others in linear time, as described later in this paper. This property is the basis of our run-time engine which takes relationships between intervals as a description of the interaction to occur and can determine which parts of the script are occurring, which are past, and which are going to happen in the future by considering the input from sensing routines.

Finally, Allen's algebra is a strong temporal algebra that allows the expression of mutually exclusive intervals. For instance, to state that a CG character does not perform action A and B at the same time we simply constrain the relation between the intervals *A* and *B* to be before OR after. That is, in every occurrence of the intervals, either *A* comes before *B* or after *B*, but never at the same time. The ability of expressing mutually exclusive intervals defines different classes of temporal algebras [23]. In general, algebras without that property allow fast constraint satisfaction but are not expressive [39]. In particular, all previous constraint-based interaction languages have used those weak temporal algebras [16, 19, 33, 40]. In contrast, Allen's algebra is very expressive and can be used in real time, in our case, just because we

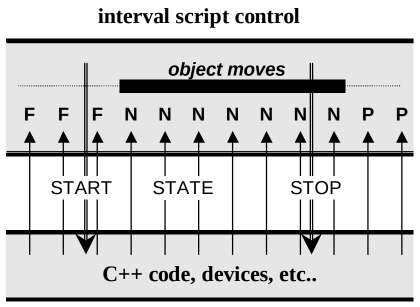


Figure 2. The structure of an interval.

have developed a fast method to compute approximations of the values that satisfy the constraints.

4. BASIC STRUCTURES OF INTERVAL SCRIPTS

In the following sections we describe the basic capabilities of the interval script paradigm. As mentioned before, there are multiple ways to implement the concepts described in this paper. We chose to develop a compiler that takes a text file containing the descriptions of the intervals and the temporal constraints between them and outputs a C++ file. The C++ file can be compiled and, through specially defined functions, the script can be executed at run-time.

It is arguable if a text file is the adequate format for scripting interaction and stories in comparison with a graphics user interface. We will return to this discussion later. For now, we will describe the fundamental structures of interval scripts assuming the syntax and format of our language. The grammar of the language is described in [28]. In most of the cases, the constructions we examine are completely implementation independent.

4.1 Start, Stop, and State Functions

Each action or state in interval script is described by three functions:

- START: a function that determines what must be done to start the interval; however, after its execution, it is not guaranteed that the interval has actually started.
- STOP: a function about what must be done to stop the interval; similar to START, it can happen that the interval continues after the STOP function is executed.
- STATE: a function that computes the actual current state of the interval in terms of three values, past, now, or future (written as *P*,*N*,*F*), corresponding respectively to the situation where the interval has already happened, is happening in that moment of the time, or it has not happened yet. STATE functions are oblivious to START and STOP functions and are designed to provide the best assessment of the actual situation.

As we can see from the above, in interval scripts we de-coupled the wish of the occurrence of an interval (represented by the START and STOP functions) from the actual happening of the interval. That is, an interval script describes how an interaction or story should happen and does not assume that response is perfect. That is simply because it is impossible to predict how characters or devices actually react in run-time due to delays, device constraints, or unexpected interactions. This follows the notion of “grounding” as proposed by Rodney Brooks for autonomous robots [8]. During run-time, the interval scripts engine examines the current state of the intervals (through the results of STATE functions) and tries to achieve a

```
"camera moves" =
  {
     START:    execute [> camera.Move(posA,posB); <];
     STOP:     execute [> camera.Stop(); <];
     STATE:    set-state pnf-expression
                [> (camera.isMoving() ? _N_ : P_F) <];
  }.
```

Figure 3. Interval describing the movement of the camera character.

set of states compatible with the temporal constraints by executing appropriately START and STOP functions.

Figure 2 shows a typical execution of an action in interval scripts. This action corresponds to a movement of a CG character. In this diagram time is running from left to right. As we see, the START function is executed some time before the STATE functions actually detects the occurrence of the action. Similarly, it takes time for the effects of the STOP call to be sensed. In the case of interactive environments, it is fairly common that some physical devices have significant delays. But also in computer graphics scenes it can take time for a character to start an action because pre-conditions must be have to be achieved first. For instance, a command for moving might be delayed because the character first must stand up. Figure 2 depicts the best case, where the action actually happens after the START call. It is easy to see that physical or spatial constraints can prevent an object to move. In the same way, a character might start to move because an object bumped into him. In both cases, we would expect that the STATE function to report the actual state of the character.

4.2 Encapsulating Code in Intervals

It is important that a scripting language communicates with low-level programming languages that provide basic functions and device access. In our current implementation of interval scripts, we allow the inclusion of C++ code directly into the script.

Let's examine an example from the script of one of our experiments, the art installation called *"It"* described in more detail in the applications section of this paper. In that installation a camera-like object appears on the screen and interacts with the user (notice that this is not the computer graphics virtual camera, but a CG object that looks like a photographic camera).

Figure 3 shows the definition of the interval *"camera moves"* in our language for interval scripts. The definition is comprised between curly brackets determining the basic functions of the interval. 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 "`Move`" of the object "`camera`" with parameters "`posA`" and "`posB`". These classes, variables, and objects are defined in separated C++ files that are linked together with the code generated by the interval script compiler.

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 PNF-value returned by the execution of the C++-code. In the case depicted in fig. 3, a method for the object "`camera`" determines if the computer graphics camera is moving or not. If true, the state of the interval is set to now, referred in the C++-code by the special constant "`_N_`"; otherwise, the state is set to be past OR future, symbolized by "`P_F`". We similarly define the constants "`P__`", "`__F`", "`PN_`", "`_NF`", and "`PNF`". The last constant stands for past OR now OR future, that is, there is no information available about the interval.

```
"camera moves" ={ . . . }.
"camera zooms" ={ . . . }.
"camera moving sound" = { . . . }.

better-if   "camera moves"   meet   "camera zooms".
better-if   "camera moving sound"   start OR equal   "camera moves".
```

Figure 4. Script with temporal constraints.

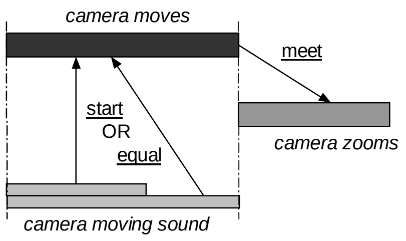


Figure 5. Diagram of the temporal constraints in the script of fig. 4.

4.3 Putting Constraints between Intervals

Let's examine how temporal constraints are defined. Continuing the example of fig. 3, suppose that after the camera moves, it should zoom, that is, move forward towards the user. Also, we would like that the camera movement to be accompanied by the sound of a pre-recorded file.

Figure 4 shows the script corresponding to the situation. It contains three intervals (whose definition, using C++ inline code, is omitted for clarity). The interval declarations are followed by two statements establishing temporal constraints between intervals. In the first, it is declared that *"camera moves"* should meet with the interval *"camera zooms"*. Notice the syntax better-if which was chosen to imply that this is a constraint that will be tried to be enforced but that is not guaranteed to occur.

Figure 5 shows a possible occurrence of the two intervals.

The last line of the script in fig. 4 establishes a constraint between the intervals *"camera moving sound"* and *"camera moves"*. They should always start together and but the former interval can end before or at the same time as *"camera moves"*. Figure 5 renders the two possible occurrences of *"camera moving sound"* in order to respect the constraint start OR equal.

4.4 Defining on Previous Intervals

Although the ability to include references to external code is very important, a key feature that we want to introduce with interval scripts is the possibility of defining a new interval solely based on other intervals. With this we can create hierarchies, abstract concepts, and develop complex, high-level scripts.

Continuing with our example, we have the situation where when the user makes a pose (as detected by a computer vision system) and the camera has finished moving and zooming, the

```
"camera moves" ={ . . . }.
"camera zooms" ={ . . . }.
"camera moving sound" = { . . . }.
better-if  "camera moves"  meet  "camera zooms".
better-if  "camera moving sound"  start OR equal  "camera moves".

"user is posing" =
  {  STATE:    set-state pnf-expression
                [> (user.isMoving() ? _N_ : P_F) <];  }.
"ready to click" =
  {  STATE:    if  "camera zooms"  is past
                   AND "user is posing"  is now
                set-state now
               endif. }.
"camera clicks" = { . . . }.


better-if  "ready to click"  start OR equal OR i-start ... "camera clicks".
```

Figure 6. Scripting based on previously defined intervals.

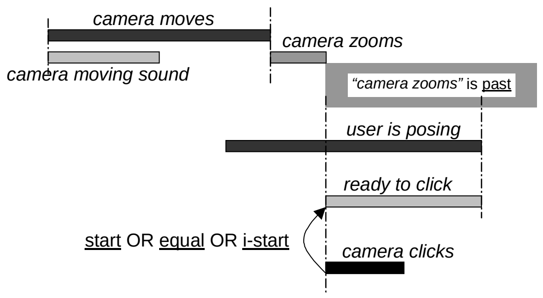


Figure 7. A possible occurrence of the script described in fig. 6.

camera “clicks” and “takes” a picture. Figure 6 shows the script corresponding to this interaction.

Initially the interval *“user is posing”* is defined as before, by reference to C++ code that communicates with the vision system. Then we define the interval *“ready to click”* that has only a STATE function. The state of the interval is determined by checking the state of two previously defined intervals: if *“camera zooms”* is in the past state and *“user is posing”* is happening than the state is set to now, otherwise is undetermined (*PNF*).

To accomplish the clicking of the camera and the “fake” taking of the picture the interval *“camera clicks”* is defined, again through invocation of C++ code. We want that whenever *“read to click”* starts, 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. That is, when the *“ready to click”* assumes the state now (by examining the state of its two defining intervals), the propagation of temporal constraints will request “camera clicks” to be running. In response to

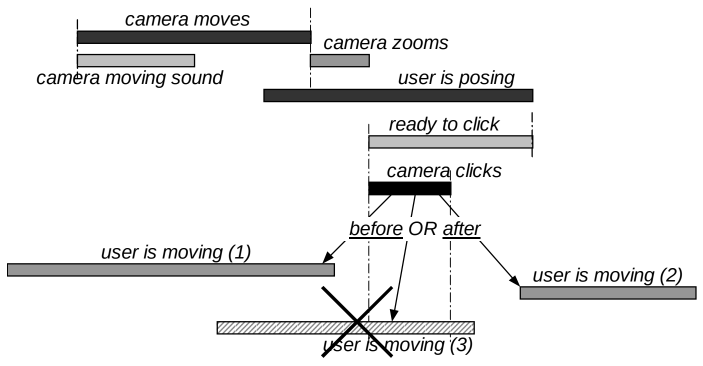


Figure 8. Possible occurrences of *"user is moving"*.

that the run-time engine can call the START function of *"camera clicks"* for as many cycles as needed until the state of this interval also becomes now.

A typical occurrence (start) is depicted in the diagram of fig. 7. Notice that *"ready to click"* is now in the intersection between *"user is posing"* and the time after *"camera zooms"* is finished.

4.5 Mutually Exclusive Actions

We want to improve the scene described above by not allowing the taking of pictures if the user is moving. This is the classical case of mutually exclusive intervals mentioned earlier. Using interval scripts the expression of such constraints is easily accomplished by adding the following lines to the script of fig. 6:

```
"user is moving" = { . . . }.
better-if  "camera clicks"  before OR after  "user is
  moving".
```

Here the interval *"user is moving"* communicates with the vision system and assumes the state now if the user is not perceived as moving around. Then, a constraint is established determining that moving and clicking never happen at the same time.

Figure 8 shows three possible occurrences of the interval *"user is moving"*. The first two, before or after *"camera clicks"*, are compatible with *"camera clicks"*. The third occurrence overlaps with *"camera clicks"* and is not compatible. In this case, since by definition *"user is moving"* is only a STATE function and can not be stopped, the run-time system will not call the START function of *"camera clicks"*, preventing the undesirable situation to happen. To better understand how this happens, it is necessary first to examine how the run-time engine actually works, what is described in the next section.

5. THE INTERVAL SCRIPT ENGINE

As we see from above an interval script associates actions and states of an interactive environment to a collection of intervals with temporal constraints between them. In this section we explain the process that triggers the call of START and STOP functions. The basis of this process is a method described in [30] called *PNF propagation*. A good account of the theoretical foundations of PNF propagation as well as a more detailed analysis of the algorithms to be presented here can be found in [28]. In this paper we provide just the basic ideas of the method that we judged necessary to apprehend the possibilities and strengths of interval scripts.

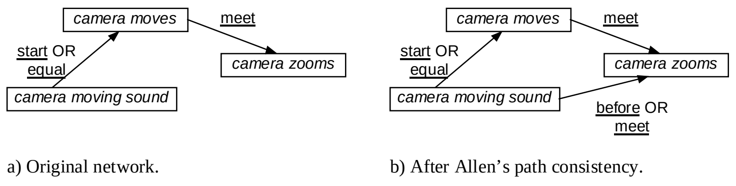


Figure 9. Interval algebra network associated with the script of fig. 4 before and after path consistency.

We start by observing that given an interval script we have two different kinds of information: how to compute the state, start, and stop each interval as represented in the basic functions; and the temporal constraints between them. The collection of temporal constraints constitute what is called an *interval algebra network* [2], that is, a network where the nodes are intervals and the temporal constraints are links between them. Figure 9.a displays the interval algebra network associated with the script of fig. 4.

5.1 First Step: Allen's Closure

To prepare an interval algebra network for run-time execution, we start by tightening the temporal relations by running Allen's *path consistency algorithm* (see [2]) on the network. The result is a network where the implied temporal relations are detected and explicitly incorporated to the network. For example, by looking at fig. 5 and fig. 9, we can see that since the interval *"camera moving sound"* starts or is equal to *"camera moves"*, and that *"camera moves"* meets with *"camera zooms"*, there are only two possible relations between *"camera moving sound"* and *"camera zooms"*, before OR meet. As shown in fig. 9, this is automatically detected by Allen's algorithm. Unfortunately, detecting all the possible implied relations is *NP-hard* (as shown in [39]). However, Allen's path consistency computes a very good, conservative, approximation in most practical cases [2], and it is polynomial in the number of constraints.

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 real 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 the minimal domain of a node contains a given instant of time then the node can be happening at that time, and if **all** the time intervals contain that given instant of time then the node **must** be happening at the 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 [39]). To overcome this difficulty we devised an optimization scheme called *PNF propagation* comprising two mechanisms, *PNF restriction* and time expansion.

5.2 PNF Restriction

The idea of simplifying interval algebra networks was first proposed by Pinhanez and Bobick in [29]. The key observation was that for control and recognition purposes there is almost no information coming from the duration of the intervals. Instead, the focus is to determine if an

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

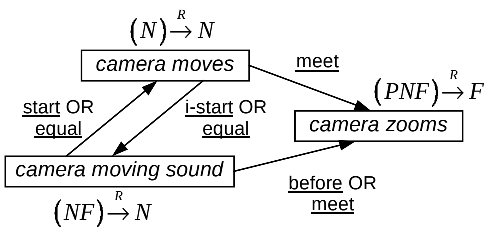

$(N)\overset{R}{\rightarrow}N$
camera moves
meet
$(PNF)\overset{R}{\rightarrow}F$
start OR equal
i-start OR equal
camera zooms
camera moving sound
before OR meet
$(NF)\overset{R}{\rightarrow}N$

Figure 10. An example of PNF restriction.

interval has happened (past), is happening (now), or is still to happen (future). Based on this idea, we project the 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*. In [28] the properties and efficient ways to project and manipulate these networks are detailed. Here we present the theory in a more intuitive form.

To simplify notation, let's define the set *M* of subsets of $\{\text{past}, \text{now}, \text{future}\}$

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

whose elements are abbreviated as $M = \{\varnothing, P, N, F, PN, NF, PF, PNF\}$

Now, consider that any information coming from STATE functions is translated into an element of *M*. For instance, an interval that is known to be happening has state *N*. On the other hand, intervals that are known to be not happening can be assigned the value *PF*, that is, they either already happened or will happen.

Given a primitive temporal relation *r* between two nodes A and B of a PNF-network, the values of the nodes must satisfy the constraints depicted in table 1. The construction of the table is justified in [28] but it should be noticed that the values correspond 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 values corresponding to the primitives. Using table 1, we define a *solution* of a PNF-network to be an assignment of values that satisfies the constraints of the table, and the *minimal domain* of a node as the set of all values that belong to at least one solution.

Given an initial set of values for each node, we call the *restriction of a PNF-network* the process of computing the minimal domain of the network restricted to the initial values. That is, restriction eliminates all the values that are incompatible with any solution. We call any initial set of values as *a component domain* of the PNF-network, and the restriction of a component domain *W* as $R(W)$. Notice that if the minimal domain of any node is $\varnothing$, then there are no solutions for the network.

Figure 10 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, the fact that *"camera moves"* is happening requires the interval *"camera zooms"* to be in the future, according to the value for meet in table 1. In the case of *"camera*

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 $W_i \neq$ PNF        (1)
$\overline{W} \leftarrow W$                                          (2)
while $Q \neq \varnothing$                                           (3)
    $w_0 \leftarrow pop(Q)$                                          (4)
    for each node $w_i$                                              (5)
        $X \leftarrow F(\overline{W_0}, P_{i_0 i})$                  (6)
        if $\overline{W_i} \neq X \cap \overline{W_i}$               (7)
            $\overline{W_i} \leftarrow X \cap \overline{W_i}$        (8)
            $push(w_i, Q)$                                           (9)
return $\overline{W}$                                                (10)
```

Figure 11. Algorithm to compute arc-consistency.

moving sound", it is easier to examine the inverse relation (implied in the graph, and automatically computed by Allen's algorithm) i-start OR equal. According to table 1, the admissible values for *"camera moving sound"* are *PN* and *N*, respectively. Considering that the initial value of *"camera moving sound"* is *NF* the only value that can belong to any solution is *N*. Notice that given solely the information that *"camera moves"* is happening and that *"camera moving sound"* has not finished (*NF*), it is possible to infer from the temporal constraints that *"camera moving sound"* is, in fact, happening now, and that *"camera zooms"* is still to happen (future).

Computing the minimal domain of a PNF-network is, however, still a *NP-hard* problem (see [11]). Instead, we have been using in our applications a conservative approximation to the minimal domain based on the computation of the *arc-consistency* (as proposed by Mackworth [21]) of the PNF-network. The main advantage is that arc-consistency is $O(n)$ in the number of constraints *n*.

The procedure in fig. 11 shows an algorithm that computes the maximal arc-consistent network under a component domain *W*. This is a version of the arc-consistency algorithm AC-2 proposed in [21] and adapted to the component domain notation. The algorithm uses the function *F* which given a PNF state and a set of primitive relations, returns a PNF state that satisfies table 1.

In [28]we prove that this algorithm is sound and linear in the number of constraints. We also show that arc-consistency produces a reasonable approximation of the minimal domain. In fact, in our experiments we have encountered few situations where they were actually different (see [28]for details).

5.3 Time Expansion and PNF Propagation

From the way the interval now is defined, it is clear that PNF-restriction deals exclusively with determining feasible options of an action **at a given moment of time**. The question is how much information from one moment of time can be carried to the next?

In fact, information from the 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. Similarly, if the current value of the interval is now, in the next instant of time it can still be now, or the corresponding action might have ended, when the interval should be past. To capture these ideas, 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.

Formally, given the PNF-state of a variable in time, we want to define a function T, called the *time expansion function*, that considers a component domain W^t at time t and computes the component domain $W^{t+1} = \mathrm{T}(W^t)$ 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 $\{\text{past},\text{now},\text{future}\}$, such as:

$$\tau_m(\text{past}) = P \qquad \tau_m(\text{now}) = PN \qquad \tau_m(\text{future}) = NF$$

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, $\mathrm{T}_m\colon M \to M$ 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*, $\mathrm{T}_m\colon U \to U$ (abusing the notation), as the component-wise application of the original T_m on a $W = (W_i)_i$, $\mathrm{T}_m(W) = (\mathrm{T}_m(W_1), \mathrm{T}_m(W_2), \ldots, \mathrm{T}_m(W_n))$

To use time expansion, we typically make the initial component domain W^0 as being composed only of PNF states, $W^0 = (PNF)_i$. Then, for each instant of time *t*, we can determine through sensor information or external sources the PNF-state of some of the variables. Next, we create the component domain V^t containing all these known values and assigning PNF for the other variables. Then, given the previous component domain W^{t-1}, we can compute an upper bound of the current minimal domain of the PNF-network by making (see the proof in [28])

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

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 which is filtered through the intersection with the past information (safely provided by $\mathrm{T}_m\left(W^{t-1}\right)$). Then, information incompatible with the structure of the problem is removed by restriction, through the computation of the minimal domain. In practice, we have been employing the arc-consistency as defined in fig. 11 instead of the restriction to assure that computation occurs in linear time.

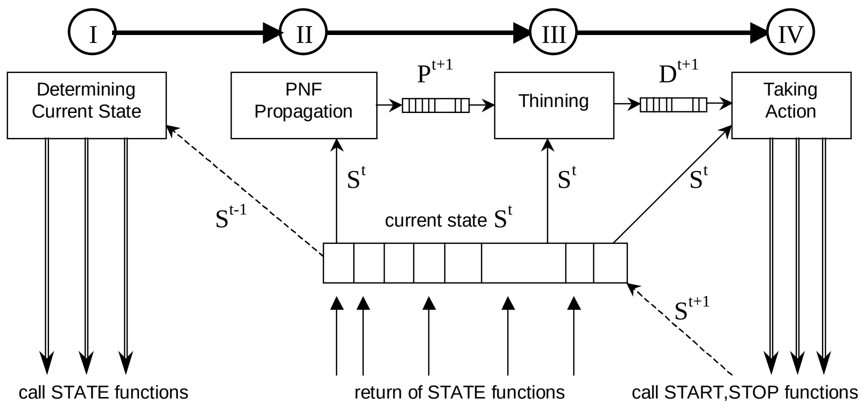


Figure 12. One cycle of the interval script engine.

5.4 Run-Time Engine Architecture

The formulation of PNF propagation described above was developed considering basic problems of action recognition [30]. For running interval scripts in the critical conditions of interactive environments we had to change some aspects in order to encompass some desired features. First, pure PNF propagation cannot recover from errors. Once an interval is assigned the past state it remains with that value. Second, as mentioned above, it may take time for intervals do actually start and stop; while waiting for this to happen, the run-time system might detect states with no solutions. For instance, if intervals *A* and *B* should meet, the end of *A* triggers the call for the START function of *B*, but while that does not happen we have a situation that can not be satisfied by any value of the nodes.

The solution for these issues is the decoupling of wish and reality alluded above. Notice that, by design, an interval script describes the situations that should happen but not the intermediate, unexpected states. The handling of those is left for the run-time engine.

Figure 12 shows a diagram of a cycle of the 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. It is important to notice that, unlike PNF propagation, the current state is not affected by time expansion or PNF restriction.

Each cycle *t* is composed of the following four stages:

I. <u>Determining the Current State:</u> all the STATE functions are called and the engine waits until all are the results are reported back and stored in the component domain S^t. For this computation, the STATE functions are allowed to use the previous states of any other interval, available in the previous component domain S^{t-1}. After this stage, the current state S^t remains unchanged for the rest of the cycle.

II. <u>PNF Propagation:</u> in this stage the engine tries to determine what changes can be made so in the next cycle, $t+1$, the constraints are satisfied. Unlike in the original time expansion, we consider for expansion only those intervals that can be started or stopped,

$$\overline{\mathrm{T}_m}\left(S_i^t\right)=\begin{cases}\mathrm{T}_m\left(S_i^t\right) \text{ if the interval } i \text{ has STARTor STOP functions}\\ S_i^t \text{ otherwise}\end{cases}$$

Using this definition, this stage PNF-propagates the current state, by computing P_i^{t+1}

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

III. Thinning: the result of stage II is normally too big and undetermined. To have a more specific forecast of the next stage, we apply an heuristic where the current state of an interval should remain the same unless (1) it contradicts the result of the PNF-propagation and (2) the final result is still feasible. This is accomplished by taking a special intersection operation between the prevision of the next state P^{t+1} and the current state S^t. For each node the special intersection is computed by

$$S_i^t \,\overline{\bigcap}\, P_i^{t+1} = \begin{cases} S_i^t \bigcap P_i^{t+1} & \text{if} \quad S_i^t \bigcap P_i^{t+1} \neq \emptyset \\ P_i^{t+1} & \text{otherwise} \end{cases}$$

The result is them passed through PNF restriction to assure that there are solutions and to remove impossible states,

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

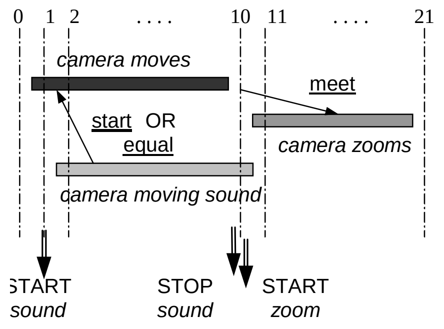
IV. Taking Action: The result of thinning, D^{t+1}, is compared to the current state S^t, and START and STOP functions are called if a need to change the state of an interval is needed. This follows the following table:

$x \subseteq S_i^t$	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, $D_i^{t+1} = N$, then the START function of the interval is called.

The interval script language provides mechanisms by which a START or a STOP function can set the state of an interval for the next cycle $t+1$. This was included to facilitate some constructions. In the diagram, this is show as a dashed arrow bringing results from the run of the START and STOP functions to S^{t+1}.

Figure 13 shows an example of run of the interval script of fig. 4. In the first instant of the run, *t=0*, the state S^0 of all intervals is *F*. Although the result of PNF propagation, P^1, allows the first two intervals to be either in the now or in the future states, the result of the thinning process, D^1, suggests to keep the state as it is and no action to be taken. In the next instant of time, *t=1*, the interval *"camera moves"* starts, as detected by its STATE function. When PNF propagation is run, the fact that *"camera moves"* and *"camera moving sound"* should start together constrains the desired state of the later interval to be *N*, and provokes a call for its START function. In the next instant, *t=2*, we assume that *"camera moving sound"* is already running, and therefore all the constraints are satisfied. The system remains in this state up to *t=9*.



interval	t=0			t=1			t=2			t=10			t=11			t=21		
	S^0	P^1	D^1	S^1	P^2	D^2	S^2	P^3	D^3	S^{10}	P^{11}	D^{11}	S^{11}	P^{12}	D^{12}	S^{21}	P^{22}	D^{22}
camera moves	F	NF	F	N	PN	N	N	PN	N	P	P	P	P	P	P	P	P	P
camera moving sound	F	NF	F	F	N	N	N	PN	N	N	P	P	P	P	P	P	P	P
camera zooms	F	F	F	F	F	F	F	NF	F	F	N	N	N	PN	N	P	P	P

Figure 13. Example of a run of the interval script of fig. 4.

When *t=10*, *"camera moves"* finishes and assumes the *P* state. Because of the constraints, *"camera moving sound"* should stop and *"camera zooms"* should start. Notice that the result of PNF propagation, P^{11}, shows exactly that configuration, and the appropriate actions are taken. In *t=11* the desired state is reached for both intervals; if it was not the case, we would have a state similar to *t=10*, and the START and STOP functions would be called again. Finally, *"camera zooms"* ends at *t=21* and all intervals assume the past state.

5.5 Surviving Conflicts and Errors

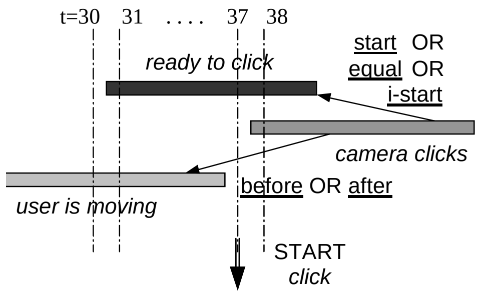
An interval script describes an interaction as it should happen. The example shown above is a simple situation where no conflict is happening. However, conflicts happen commonly and in the run-time engine they are detected by the PNF restriction algorithm. Unfortunately, like any constraint propagation method, it is impossible to detect which individual interval or constraint is the source of the problem.

To handle conflicts we included in the engine some recovery mechanisms that are employed in the case the restriction algorithm finds that the PNF network has no solutions. They are included in the following stages:

II- PNF Propagation: if the computation of $P^{t+1} = R\left(\overline{\mathrm{T}_m}(S^t)\right)$ detects no solution, the engine tries to enlarge the space of possibilities by using simple PNF propagation, that is, expanding all states regardless of the existence of START or STOP functions, $P^{t+1} = R\left(\mathrm{T}_m(S^t)\right)$.

Although this normally handles most problems, there are situations where sensors report incorrectly and produce a state with no 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: if the computation of $D^{t+1} = R\left(S^t \overline{\cap} P^{t+1}\right)$ yields a state with no solutions, we simply ignore the restriction, $D^{t+1} = S^t \overline{\cap} P^{t+1}$. Normally this prevent any action to be taken since the states of P^{t+1} tend to be not as thin as required to call start and stop functions.



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}
ready to click	F	F	F	N	**N**	**N**	N	N	N	N	N	N	N
camera clicks	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

Figure 14. Example of run with conflict.

These methods to recover from errors have been designed and tested in our applications. The basic principle is to avoid taking actions so problems do not cascade.

An example of a conflict situation is show in fig. 14. Here, we consider the case where the intervals *"camera clicks"* and *"user is moving"* are defined as mutually exclusive and a situation where both were expected to happen is occurring. At *t=31*, a conflict is detected by the PNF propagation stage since there is no way to satisfy the requirements that *"camera clicks"* should start together with *"ready to click"* and while *"user is moving"* is occurring. In this situation the first level of recovery succeeds and find a set of states that is compatible with the constraints as show in fig. 14. However, P^{32} is quite non-specific and the thinning process basically keep the current situation as it is, without taking any action. Later, at *t=37*, when *"user is* moving" finishes, and since *"ready to click"* is still happening, *"camera clicks"* is started.

As we see, the run-time engine of interval scripts was designed to avoid halts by searching for feasible states with the less amount of change. This strategy is not guaranteed to succeed always but has been proving 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 and devising mechanisms to detect and isolate the source of conflicts.

In [28] we describe in details these features of the interval scripts language. Also, we present other features of the language such as: (1) the declaration of nested intervals; (2) a simple implementation of timers; (3) the possibility of defining START and STOP function based on previously defined intervals; (4) handling of events; (5) a simple method to re-run intervals and the action or sensor activity associated to them; (6) mechanisms to represent and recognize complex human actions (based on [29]; and (7) mechanisms to contextualize action and sensing activity. The combination of all these features make the interval scripts language extremely expressive, as shown in the many examples described in [28] and in the projects described in the next section.

6. WORKING WITH INTERVAL SCRIPTS

Evaluating a scripting method or a programming language is always difficult. We believe that the previous exposition of the basic structures of interval scripts has provided evidence for our claim of simplicity.

The experiments listed below are evidence towards our belief that the language is more expressive than current paradigms. We do not see how these systems (especially the last two) could have been programmed using, for instance, event-loops without major problems in debugging, execution, and maintenance. Particularly, the control structure of the systems described was developed in very short periods of time. Based on these experiments, we argue that our main objectives when designing intervals scripts were achieved, that is, that the paradigm provides expressiveness and simplicity beyond current scripting methods.

The first two experiments, *"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 (in [27] Pinhanez details the concept of computer theater, the origins of the term, and related works). Our research in computer theater has concentrated on building automatic, story-aware computer-actors able to interact with human actors on camera-monitored stages.

6.1 *"SingSong":* a First Experiment

"SingSong" was our first experiment with interval scripts [31]. *"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 [41].

All the interaction is physical, non-verbal: the user or performer gestures and the CG-characters sing notes and move. There is a CG-object — a pitching fork — which the user employs during one of the scenes. *"SingSong"* is an environment which immerses the performer in the following simple story which unfolds as the interaction proceeds:

> *Singers of a chorus (the CG-creatures) are animatedly talking to each other. The conductor enters and commands them to stop by raising her arms. One of the singers — #1 — keeps talking until the conductor asks it to stop again. Singer #1 stops but complains (by expanding and grudging sounds). Following, a pitching fork appears on the screen and the conductor starts to tune the chorus: he points to a singer and "hits" the pitching fork by moving his arm down. Any singer can be tuned at any time. However, singer #1 does not get tuned: it keeps giving back the conductor a wrong note until the conductor knees down and pleads for its cooperation. After all the singers are tuned, a song is performed. The conductor controls only the tempo: the notes are played as he moves her arms up. When the song is finished, applause is heard, and when the conductor bows back the singers bow together with him. Just after that singer #1 teases the conductor again and the singers decide to go back to talking to each other.*

"SingSong" was produced and performed in the summer of 1996. Although it is a short play of about 4 minutes, the interval script involved around 70 intervals (although the resulting PNF network was three times bigger) and included a great deal of low-level control of I/O devices. It took about two days of work to write and debug the script. More details can be found in [31]. Figure 15 shows a sequence of scenes from *"SingSong"*.

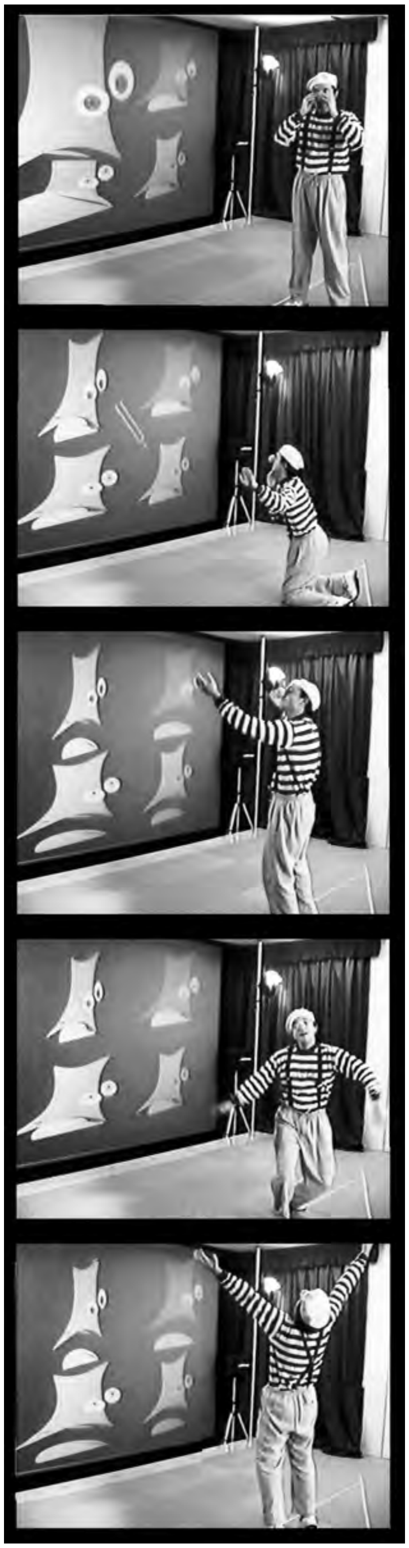


Figure 15. Scenes from *"SingSong"*.

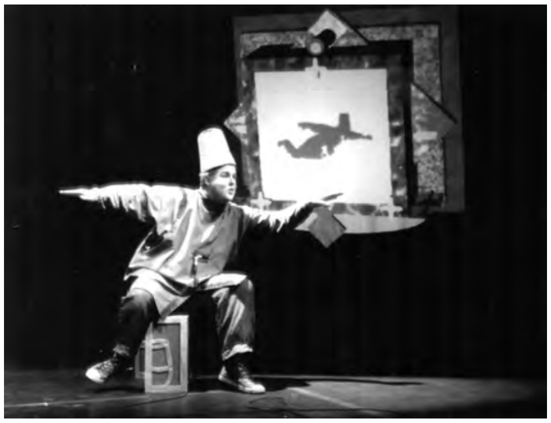


Figure 16. Scene from "It/I". The computer graphics object on the screen is autonomously controlled by the computer character *It*.

6.2 *"It / I":* a Computer Theater Play

Following *"SingSong"* we decided that it was necessary to develop a real test both for the idea of computer theater and for interval scripts. Particularly, we were interested whether a computer-actor could sustain the interest of an audience for long periods of time, enough to go beyond the novelty of the technology and to achieve real dramatic content. On the other hand we were interested in testing the robustness of the interval script paradigm in a condition where 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. 16). 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 repetition of a basic cycle where *"I"* is lured by *"It"*, is played with, gets frustrated, quits, and is punished for quitting. For example, the second scene of the play is :

"I" is sitting on the stage, indifferent to everything, bathed by blue light. To attract his attention, "It" projects a

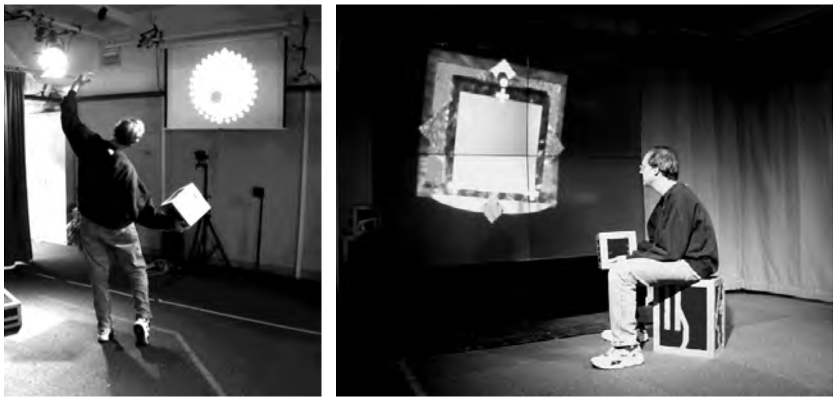


Figure 17. Scenes from a run of *"It"*.

picture of a wedding on the stage screen. When "I" pays attention to the picture, the image is removed from the screen, the lights change, and a CG-object similar to a photographic camera appears on the other screen, following "I" as he moves around. When "I" makes a pose, the camera shutter opens with a burst of light and the corresponding clicking sound. Following, a CG-television appears on the other screen and, when "I" gets close, it starts to display a "slide show" composed by silhouette images "taken" by the camera. After some pictures are shown, the camera "calls" "I" to take another picture. This cycle is repeated until "I" refuses to continue to play with the machine and remains in front of the television; this refusal provokes an irate reaction from "It", which throws CG-blocks at "I" while storming the stage with lights and playing harsh loud noises. Then, "I" is left on a desolated, silent stage.

The above segment exemplifies the complexity of the interaction in a typical scene of *"It/I"*. The scenes have a complexity level quite beyond previous full-body interactive systems [20, 22]. 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, first in front of the audience, and individually afterwards.

The interval script of each scene contained approximately 100 intervals (for a total of about 400 in the whole play). The development of the interval script took approximately two weeks including here rehearsal time.

6.3 *"It": an Interactive Installation

When the spectators went on the stage to re-enact the scene from *"It/I"* they displayed a variety of reactions. Some of them could easily remember the sequence of actions in the play and could navigate through the scene without external help but others were partially confused about what to do.

To overcome these problems we decided to re-create the environment of the play *"It/I"* in a version for users called *"It"*. Inhabited by the ***It*** character, the space tries to trap the user inside it, under the disguise of a game of taking and showing pictures. *"It"* uses the same visual (images, computer graphics) and sound elements as *"It/I"*, but it was designed to be a self contained, stand-alone piece that can be enjoyed by users completely unfamiliar with the play. The construction of the installation *"It"* was finished in March of 1999 in a laboratory space at the MIT Media Laboratory. Since then there have been dozens of users experiencing the feeling of being trapped by ***It***. Figure 17 shows pictures of a user playing in *"It"*.

"It" employs 4 control modules (following the SCD architecture as described in [26]). The most complex module runs an interval script composed of more than 200 intervals. All the examples of interval scripts shown in this paper come from the script of *"It"*. They already incorporate some of the significant improvements in the interval script language that were developed after the experience with *"It/I"*.

7. FUTURE DIRECTIONS

Although the interval script language is already reaching a stable state, there is still work to be done in at least three different directions. First, we would like to get the system to a state where it could be released to other researchers and designers interested in using interval scripts to build interactive systems. We expect to reach this level soon. In this context it becomes possible to evaluate how difficult the language is to be learned and which features different kinds of designers would like to have added to the language.

Second, we want to investigate which is the ideal interface for the paradigm: text-based or graphical. In the case of a graphical interface, a possible approach is to allow the user to construct temporal diagrams similar to fig. 17. However, since we allow disjunction of temporal constraints there are always multiple possibilities for the temporal arrangement of the intervals what might create more confusion than provide help. Nevertheless, we found always useful to draw the diagrams while thinking about the temporal constraints.

Finally, we would like to integrate into the interval script language mechanisms that represent actions in forms that the machine can reason about. Our work described [28], based on Roger Schank's [32] conceptualizations, is a serious candidate for such a representation.

8. CONCLUSION

In this paper we propose the interval script paradigm for scripting interaction that is based on declarative programming of temporal constraints. Unlike previous constraint-based languages, we employed a strong temporal algebra with mutually exclusive intervals. Through the examples written in the interval script language we have shown that the paradigm allows good expressiveness and significantly facilitates the management of context, story, and history in an interactive environment.

From the experience acquired in the use of the language for the implementation of three different projects, we believe that scripting systems incorporating the interval script paradigm can significantly ease the design and building of interactive systems.

ACKNOWLEDGEMENTS

"SingSong" was produced at ATR Media Integration and Communication Laboratories in Kyoto, Japan. The performance was written, directed, and performed by Claudio Pinhanez. *"It / I"* was produced at the MIT Media Laboratory and performed at the Villers Experimental Facility. The play 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, and the actor Joshua Pritchard. Claudio Pinhanez was partially supported by a scholarship from CNPq, process number 20.3117/89.1.0

REFERENCES

[1] *Director's User Manual*. MacroMind Inc. 1990.

[2] J. F. Allen. "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM,* vol. 26 (11), pp. 832-843. 1983.

[3] B. Bailey, J. A. Konstan, R. Cooley, and M. Dejong. "Nsync - A Toolkit for Building Interactive Multimedia Presentations", *Proc. of ACM Multimedia'98*, Bristol, England, pp. 257-266. 1998.

[4] J. Bates, A. B. Loyall, and W. S. Reilly. "An Architecture for Action, Emotion, and Social Behavior", *Proceedings of the Fourth European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, S. Martino al Cimino, Italy. July. 1992.

[5] B. B. Bederson and A. Druin. "Computer Augmented Environments: New Places to Learn, Work and Play", *Advances in Human-Computer Interaction*, vol. 5. Ablex, Norwood, New Jersey. 1995.

[6] B. M. Blumberg and T. A. Galyean. "Multi-Level Direction of Autonomous Agents for Real-Time Virtual Environments", *Proc. of SIGGRAPH'95*. 1995.

[7] A. Bobick, S. Intille, J. Davis, F. Baird, C. Pinhanez, L. Campbell, Y. Ivanov, A. Schutte, and A. Wilson. "The KidsRoom: A Perceptually-Based Interactive Immersive Story Environment", *PRESENCE: Teleoperators and Virtual Environments*, vol. 8 (4), pp. 367-391. 1999.

[8] R. Brooks. "Intelligence without Reason", *Artificial Intelligence*, vol. 47, pp. 139-159. 1991.

[9] M. C. Buchanan and P. T. Zellweger. "Automatic Temporal Layout Mechanisms", *Proc. of ACM Multimedia'93*, Ahaheim, California, pp. 341-350. August. 1993.

[10] M. H. Coen. "Building Brains for Rooms: Designing Distributed Software Agents", *Proc. of IAAI'97*, Providence, Connecticut, pp. 971-977. August. 1997.

[11] R. Dechter. "From Local to Global Consistency", *Artificial Intelligence*, vol. 55 (1), pp. 87-107. 1992.

[12] C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi. "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications", *Proc. of SIGGRAPH'94*, Orlando, Florida, pp. 421-434. July 24-29. 1994.

[13] T. A. Galyean. *Narrative Guidance of Interactivity*. Ph.D. Thesis. Media Arts and Sciences Program: Massachusetts Institute of Technology, Cambridge, Massachusetts. 1995.

[14] R. Hamakawa and J. Rekimoto. "Object Composition and Playback Models for Handling Multimedia Data", *Proc. of ACM Multimedia'93*, Ahaheim, California, pp. 273-281. August. 1993.

[15] M. Johnson, A. Wilson, C. Kline, B. Blumberg, and A. Bobick. "Sympathetic Interfaces: Using a Plush Toy to Direct Synthetic Characters", *Proc. of CHI'99*, Pittsburgh, Pennsylvania. May. 1999.

[16] M. Jourdan, N. Layaida, C. Roisin, L. Sabry-Ismail, and L. Tardif. "Madeus, an Authoring Environment for Interactive Multimedia Documents", *Proc. of ACM Multimedia'98*, Bristol, England, pp. 267-272. September 12-16. 1998.

[17] K. I. Kakizaki. "Generating the Animation of a 3D Agent from Explanation Text", *Proc. of ACM Multimedia'98*, Bristol, England, pp. 139-144. 1998.

[18] J. K. Kalita. *Natural Language Control of Animation of Task Performance in a Physical Domain*. Ph.D. Thesis. Department of Computer and Information Science: University of Pennsylvania, Philadelphia, Pennsylvania. 1991.

[19] M. Y. Kim and J. Song. "Multimedia Documents with Elastic Time", *Proc. of ACM Multimedia'95*, San Francisco, California, pp. 143-154. November. 1995.

[20] M. W. Krueger. *Artificial Reality II*. Addison-Wesley. 1990.

[21] A. K. Mackworth. "Consistency in Networks of Relations", *Artificial Intelligence*, vol. 8 (1), pp. 99-118. 1977.

[22] P. Maes, T. Darrell, B. Blumberg, and A. Pentland. “The ALIVE System: Full-Body Interaction with Autonomous Agents”, *Proc. of the Computer Animation'95 Conference*, Geneva, Switzerland. April. 1995.

[23] I. Meiri. “Combining Qualitative and Quantitative Constraints in Temporal Reasoning”, *Artificial Intelligence,* vol. 87 (1-2), pp. 343-385. 1996.

[24] R. Pausch, T. Burnette, A. C. Capeheart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White. “A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality”, *IEEE Computer Graphics and Applications*. 1995.

[25] K. Perlin and A. Goldberg. “Improv: A System for Scripting Interactive Actors in Virtual Worlds”, *Proc. of SIGGRAPH'96*. August. 1996.

[26] C. Pinhanez. “The SCD Architecture and its Use in the Design of Story-Driven Interactive Spaces”, *Proc. of 1st Internation Workshop on Managing Interactions in Smart Environments (MANSE'99)*, Dublin, Ireland. 1999.

[27] C. S. Pinhanez. “Computer Theater”, *Proc. of the Eighth International Symposium on Electronic Arts (ISEA'97),* Chicago, Illinois. September. 1997.

[28] C. S. Pinhanez. *Representation and Recognition of Action in Interactive Spaces*. Ph.D. Thesis. Media Arts and Sciences Program: Massachusetts Institute of Technology. 1999.

[29] C. S. Pinhanez and A. F. Bobick. “PNF Propagation and the Detection of Actions Described by Temporal Intervals”, *Proc. of the DARPA Image Understanding Workshop*, New Orleans, Louisiana. May. 1997.

[30] C. S. Pinhanez and A. F. Bobick. “Human Action Detection Using PNF Propagation of Temporal Constraints”, *Proc. of CVPR'98*, Santa Barbara, California, pp. 898-904. June. 1998.

[31] C. S. Pinhanez, K. Mase, and A. F. Bobick. “Interval Scripts: A Design Paradigm for Story-Based Interactive Systems”, *Proc. of CHI'97,* Atlanta, Georgia, pp. 287-294. March. 1997.

[32] R. C. Schank. “Conceptual Dependency Theory”, in *Conceptual Information Processing*. North-Holland. pp. 22-82. 1975.

[33] K. C. Selcuk, B. Prabhakaran, and V. S. Subrahmanian. “CHIMP: A Framework for Supporting Distributed Multimedia Document Authoring and Presentation”, *Proc. of ACM Multimedia'96*, Boston, Massachusetts, pp. 329-339. November. 1996.

[34] C. Shaw, M. Green, J. Liang, and Y. Sun. “Decoupled Simulation in Virtual Reality with the MR Toolkit”, *ACM Transactions on Information Systems*, vol. 11 (3), pp. 287-317. 1993.

[35] N. M. Thalmann and D. Thalmann. *Synthetic Actors in Computer Generated 3D Films*. Springer-Verlag, Berlin, Germany. 1990.

[36] N. Tosa, H. Hashimoto, K. Sezaki, Y. Kunii, T. Yamada, K. Sabe, R. Nishino, H. Harashima, and F. Harashima. “Network-Based Neuro-Baby with Robotic Hand”, *Proc. of IJCAI'95 Workshop on Entertainment and AI/Alife*, Montreal, Canada. August. 1995.

[37] X. Tu and D. Terzopoulos. “Artificial Fishes: Physics, Locomotion, Perception, Behavior”, *Proc. of SIGGRAPH'94*, Orlando, Florida, pp. 43-50. July 24-29. 1994.

[38] G. van Rossum, J. Jansen, K. Mullender, and D. Bulterman. “CMIFed: a Presentation Environment for Portable Hypermedia Documents”, *Proc. of ACM Multimedia'93*, California. 1993.

[39] M. Vilain and H. Kautz. “Constraint Propagation Algorithms for Temporal Reasoning”, *Proc. of AAAI'86*, Philadelphia, Pennsylvania, pp. 377-382. 1986.

[40] S. Wirag. “Modeling of Adaptable Multimedia Documents”, *Proc. of the European Workshop on Interactive Distributed Multimedia Systems and Telecommunications Services*, Darmstadt. September. 1997.

[41] C. Wren, A. Azarbayejani, T. Darrell, and A. Pentland. "Pfinder: Real-Time Tracking of the Human Body", *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 19 (7), pp. 780-785. 1997.