The TIMELOGIC Temporal Reasoning System

Johannes A. G. M. Koomen

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 231
(Revised for Version 3.x)

March 1989

Abstract

The TIMELOGIC system is an interval-based forward-chaining inference engine and database manager of temporal constraints. Relational constraints, indicating relative order between intervals, are based on Allen's interval logic. The TIMELOGIC system also supports durational constraints, indicating relative magnitude between intervals, and reference links, used for the explicit or automatic construction of interval hierarchies. Constraints are posted and propagated in user-defined contexts with inheritance.

This work was supported by NSF research grants DCR-8351665 and DCR-8502481. We thank the Xerox Corporation University Grants Program for providing equipment used both in the development of the TIMELOGIC system and in the preparation of the original version of this paper.

Contents

Chapter 1

Introduction

Interval-based temporal logic, introduced by Allen [Allen, 1983], has proven to be a powerful paradigm. In many A.I. systems, information about the temporal relation between two events or propositions is as important, if not more so, as absolute length or position on a time line of any one such event or proposition. Recognizing that intervals can be related to other intervals in at most seven different ways (before, meets, overlaps, starts, during, finishes and equals) plus their inverses, Allen proposed a way of storing partial temporal knowledge by maintaining sets of possible temporal relations between any two intervals, and provided a temporal constraint propagation algorithm. Knowledge can be added to a network of interval constraints by reducing the set of possible temporal relations between two intervals, and propagating the effects of this reduction to all other intervals related to these two.

Given the relational constraint sets between intervals i and j and between j and k, the relational constraint set between i and k can be obtained through a transitivity function which successively indexes a transitivity table by the elements of the i–j and j–k sets and computes the union of the table entries. If there already existed a set of constraints between i and k, this set is intersected with the computed union to arrive at a new constraint set. If the cardinality of the resulting set is 1, then the constraint between i and k has been uniquely determined; if the intersection resulted in the empty set, then a conflict exists in the network.

This document describes the TIMELOGIC system, a stand-alone Common Lisp package that provides a facility for storing, retrieving and inferring temporal constraints between intervals. Intervals can be named by any Lisp object with the proviso that two objects which are EQL refer to the same interval. Two types of constraints can be manipulated, namely interval relations (as described above), and interval durations. Section 2 describes how these relational and durational constraints are specified and manipulated. The TIMELOGIC system provides facilities for the management of a hierarchical interval structure. Section 3 describes how intervals can be explicitly or automatically clustered by associating them with reference intervals, and how reference intervals are used to limit constraint propagation and to deduce inter-cluster interval constraints. Details of the automatic referencing algorithm can be found in Appendix A. Support for contextual reasoning and backtracking is documented in Section 4, and Section 5 provides a detailed account of the functional

interface to the TIMELOGIC system, including initialization, interval definition, relational and durational constraint posting and retrieval, and system customization.

Chapter 2

Interval Constraints

2.1 Specification of relational constraints

Relational constraints specify ordinal relationships between intervals, such as before, equals, during, *etc.* A total of 13 relational constraints are possible between any two intervals, as illustrated in Figure 2.1. These constraints are binary and pairwise disjoint, *i.e.*, two

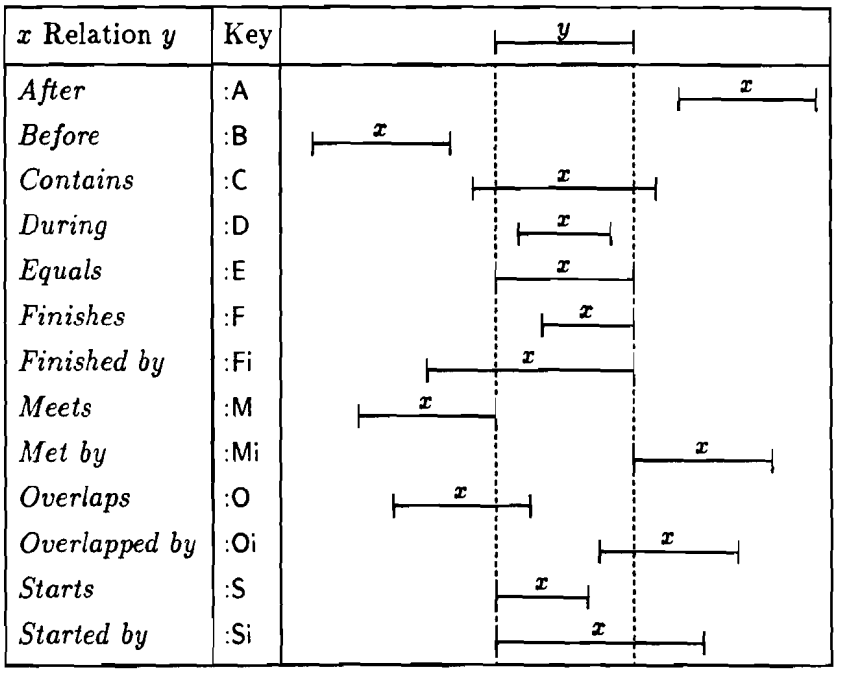

Figure 2.1: Possible relational constraints

intervals are related by exactly one of the constraints. A set of relational constraints denotes the "exclusive or" of the members of the set. In this text we indicate a relational constraint set such as "either before or after" as $\{before, after\}$.

In the TIMELOGIC system, a relational constraint set is specified either by the keyword of one of the following simple relations:

relation	key	inverse	key
before	:B	*after*	:A
during	:D	*contains*	:C
equals	:E	*equal*	:E
finishes	:F	*finished by*	:Fi
meets	:M	*met by*	:Mi
overlaps	:O	*overlapped by*	:Oi
starts	:S	*started by*	:Si

or by the keyword of one of the following compound relations:

relation	key	equivalence
somehow during	:DUR	(:D :F :S)
somehow contains	:CON	(:C :Fi :Si)
disjoint from	:DIS	(:A :B :M :Mi)
all	:ALL	(:DUR :CON :DIS :E :O :Oi)

or by a list of one or more of them.

2.2 Inferring relational constraints

If relational constraints exist between intervals X and Y and intervals Y and Z, it is possible to infer what the relational constraint is between intervals X and Z. For instance, if X {*meets*} Y and Y {*meets*} Z then it can be concluded that X {*before*} Z. This inference is accomplished by the transitivity function Constraints which indexes a 13x13 transitivity table [Allen, 1983]. Table 2.1 shows half of this transitivity table; the other half is easily obtained through the identity

$$Constraints(r_1, r_2) \equiv Inverse(Constraints(Inverse(r_2), Inverse(r_1)))$$

Note that this inference is limited in that the actual constraint between X and Z might be more precisely defined than could be concluded from the transitivity table. For instance, if X {*contains*} Y and Y {*during*} Z, then nothing can be concluded from this about the constraint between X and Z.

If the constraints between X and Y and between Y and Z are sets, the constraint between X and Z can be computed as the set union of the result of mapping the transitivity function over the cross product of the two sets. For instance, if X {*starts*, *met by*} Y and Y {*overlaps*, *equals*} Z, then it can be concluded that X {*before*, *during*, *finishes*, *finished by*, *meets*, *met by*, *overlaps*, *overlapped by*} Z, as follows:

If X {*starts*} Y and Y {*overlaps*} Z then X {*before*, *meets*, *overlaps*} Z;

If X {*starts*} Y and Y {*equals*} Z then X {*starts*} Z;

$r_1 \setminus r_2$	B	D	E	F	M	O	S
A	All	A D F Mi Oi	A	A	A D F Mi Oi	A D F Mi Oi	A D F Mi Oi
B	B	B D M O S	B	B D M O S	B	B	B
C	B C Fi M O	CON DUR E O Oi	C	C Oi Si	C Fi O	C Fi O	C Fi O
D	B	D	D	D	B	B D M O S	D
E	B	D	E	F	M	O	S
F	B	D	F	F	M	D O S	D
Fi	B	D O S	Fi	E F Fi	M	O	O
M	B	D O S	M	D O S	B	B	M
Mi	B C Fi M O	D F Oi	Mi	Mi	E S Si	D F Oi	D F Oi
O	B	D O S	O	D O S	B	B M O	O
Oi	B C Fi M O	D F Oi	Oi	Oi	C Fi O	CON DUR E O Oi	D F Oi
S	B	D	S	D	B	B M O	S
Si	B C Fi M O	D F Oi	Si	Oi	C Fi O	C Fi O	E S Si

Table 2.1: Half of the relational transitivity table

If X $\{met\ by\}$ Y and Y $\{overlaps\}$ Z then X $\{during, finishes, overlapped\ by\}$ Z;

If X $\{met\ by\}$ Y and Y $\{equals\}$ Z then X $\{met\ by\}$ Z;

Therefore conclude:

If X $\{starts, met\ by\}$ Y and Y $\{overlaps, equals\}$ Z then
X $\{before, during, finishes, meets, met\ by, overlaps, overlapped\ by, starts\}$ Z.

The following code fragment casts this last example in TIMELOGIC terminology:

```
(ADD-INTERVAL-CONSTRAINT 'X '(:Mi :S) 'Y)

(ADD-INTERVAL-CONSTRAINT 'Y '(:E :O) 'Z)

(GET-INTERVAL-CONSTRAINT 'X 'Z)

  → (:B :D :F :M :Mi :O :Oi :S)
```

2.3 Specification of durational constraints

Durational constraints specify relative magnitudes between intervals, such as half as long, equal or three times longer than, *etc.* Formally, a durational constraint is either a range of open or closed bounds, or a disjunctive set of ranges. A single bound may be used to specify a range whose lower bound is equal to its upper bound.[1] A bound is either a non-negative number or the symbol * meaning infinity. Open bounds, whose values are not included in the range they bound, are indicated as a one-element list containing a closed bound. The bounds 0 and * are by definition open bounds, and do not need to be embedded in a list. Disjunctive sets of durational constraints are indicated by a list whose first element is the symbol OR and whose remaining elements are closed bounds or ranges. The symbols =, <, <=, >, >= and <> are recognized as abbreviations for the ranges 1, (0 (1)), (0 1), ((1) *), (1 *) and (OR < >), respectively. Examples of durational constraints between intervals X and Y, and their meanings, are given in Table 2.2.

X constraint Y	X takes ... times as long as Y
6	exactly 6
(0 6)	at most 6
(0 (6))	less than 6
((3) 6)	more than 3 but at most 6
(0 *)	undetermined
(1.5 6.75)	between 1.5 and 6.75
(OR < (8 10))	either less than 1 or between 8 and 10
(OR (0 (5)) ((5) *))	either less than 5 or more than 5

Table 2.2: Durational constraint examples

All arithmetic on bounds of durational constraints results in a floating point number if either of the bounds is a floating point number and the :FLOATS property is enabled (see Section 5.8 on customization), otherwise rational arithmetic is used.

2.4 Inferring durational constraints

If durational constraints exist between intervals X and Y and intervals Y and Z, it is likewise possible to infer what the durational constraint is between intervals X and Z. For instance, if X takes 3 times as long as Y and Y takes between 4 and 5 times as long as Z then it can be concluded that X must take between 12 and 15 times as long as Z. This inference, which is accomplished using a multiplicative transitivity rule, is limited in that the actual constraint between X and Z might be more precisely defined than could be concluded from the transitivity rule. The product of two ranges R1 and R2 is defined as the range whose lower bound is the product of the lower bounds of R1 and R2 and whose upper bound is

[1] which obviously must be closed!

the product of the upper bounds of R1 and R2. The product of two bounds B1 and B2 is open unless both B1 and B2 are closed.

If the constraints between X and Y and between Y and Z are sets, the constraint between X and Z can be computed as the set union of the result of mapping the transitivity rule over the cross product of the two sets. For instance, if X takes either the same time or between 4 and 6 times as long as Y and Y takes either between 2 and 5 times or between 9 and 10 times as long as Z then conclude that X must take either between 2 and 5 times or between 8 and 30 times or between 36 and 60 times as long as Z. Note that set union over ranges collapses overlapping ranges. The following code fragment casts this last example in TIMELOGIC terminology:

```
(ADD-INTERVAL-CONSTRAINT 'X '(OR = (4 6)) 'Y :TYPE :DUR)

(ADD-INTERVAL-CONSTRAINT 'Y '(OR (2 5) (9 10)) 'Z :TYPE :DUR)

(GET-INTERVAL-CONSTRAINT 'X 'Z :TYPE :DUR)

  → (OR (2 5) (8 30) (36 60))
```

2.5 Inferring relational from durational constraints

Durational constraints can be inferred from certain relational constraint sets. If a relational constraint between intervals X and Y is a subset of ρ, then the durational constraint between X and Y must be a subset of δ, where the interpretation of ρ and δ are given in Table 2.3.

ρ	δ
(:E)	=
(:D :F :S)	<
(:D :E :F :S)	<=
(:C :Fi :Si)	>
(:C :E :Fi :Si)	>=
(:C :D:F :Fi :S :Si)	<>

Table 2.3: Relational to durational constraint map

2.6 Inferring durational from relational constraints

Similarly, relational constraints can be deduced from those durational constraints that leave no ambiguity with respect to one interval being longer than, shorter than or equal in length to another. If a durational constraint between intervals X and Y is a subset of δ, then the relational constraint between X and Y must be a subset of ρ, where δ and ρ are defined according to Table 2.4. Note that the order in which the subset tests are performed is

δ	ρ
=	(:DIS :E :O :Oi)
<	(:DIS :DUR :O :Oi)
<=	(:DIS :DUR :E :O :Oi)
>	(:DIS :CON :O :Oi)
>=	(:DIS :CON :E :O :Oi)
<>	(:DIS :CON :DUR :O :Oi)

Table 2.4: Durational to relational constraint map

significant: if a durational constraint is a subset of < then it is also a subset of <= and <>.

The TIMELOGIC system will perform these cross inferences automatically when both relational and durational subsystems are enabled.

Chapter 3

Interval Hierarchies

3.1 Constraint propagation: how bad does it get?

Each relational constraint can be asserted and updated at most 13 times, but propagating the effects of such an update is proportional to the branching factor, *i.e.*, the number of other intervals related to the two intervals whose constraint changed. Without imposing structure, a network of temporal relations between N intervals tends toward full connectivity. Hence, for each relational constraint added, Allen's basic algorithm attempts to propagate the effect to all other "Comparable" (*i.e.*, connected) intervals, $2n-2$ times invoking the (expensive!) transitivity function Constraints. Each time propagation reduces another constraint, this reduction too will be propagated across the network. As the number of intervals in the network grows, this becomes entirely insufferable.

Moreover, in a system using the interval logic, activity tends to focus on the most recently added intervals; new information does not affect old relations much, and propagating effects back to older intervals tends to be useful only for consistency checking.

3.2 Limiting propagation: reference intervals

Allen proposed adopting a notion of reference intervals similar to that of [Kahn and Gorry, 1977] to reduce the space requirements of an interval network. Clusters are formed within the network by associating intervals with one or more reference intervals. Constraint propagation takes place within each cluster only, while inter-cluster relations are obtained indirectly by applying the transitivity function along paths through the network, where paths consist of reference intervals only. For instance, consider the two activities *gardening* and *dinner* with associated intervals Gardening and Dinner. The intervals associated with all activities relating to this instance of gardening, such as Mowing and Raking, would be related directly to Gardening, and likewise, intervals such as BoilWater, CookSpaghetti, *etc.* would be directly related to this instance of preparing for and having dinner. The intervals BoilWater, CookSpaghetti, *etc.* would not be directly related to Gardening or any of its subintervals, however; those relationships would be determined by applying the constraint propagation algorithm along the path through Dinner. The effect of further constraining two dinner-related

intervals would not be propagated to any gardening-related intervals. This mechanism obviously can reduce the branching factor at each node considerably, resulting in a significant space savings (fewer relations to store) as well as a speed-up in adding new constraints due to propagation to fewer related intervals. The time to fetch inter-cluster relations is increased, but this can be kept manageable (*i.e.*, logarithmic) if the reference intervals are organized in a tree-like hierarchy.

Allen claims that imposing a reference hierarchy "can be done with little loss of information," as long as "care is taken." He does not make clear, however, of what this care consists. The fact is that it is rather easy to lose information due to the structure of the reference hierarchy, particularly between intervals in different, temporally overlapping clusters. To flesh out the example above, assume that the following constraints have been asserted:

c_1: Mowing $\{starts\}$ Gardening
c_2: Mowing $\{before\}$ Raking
c_3: Raking $\{finishes\}$ Gardening
c_4: BoilWater $\{starts\}$ Dinner
c_5: BoilWater $\{meets\}$ CookSpaghetti
c_6: CookSpaghetti $\{before, meets\}$ EatSpaghetti
c_7: EatSpaghetti $\{finishes\}$ Dinner
c_8: Dinner $\{meets\}$ Dishes

Now suppose I decided to boil the water while I'm raking the lawn:

c_9: Raking $\{equals\}$ BoilWater

In a "flat" network, the following additional constraints, among others, would be derived automatically:

c_{10}: Gardening $\{overlaps\}$ Dinner
c_{11}: Mowing $\{before\}$ Dinner
c_{12}: Mowing $\{before\}$ Dishes
c_{13}: CookSpaghetti $\{during\}$ Dinner

However, consider a structured system where Gardening is a reference interval for Mowing and Raking, and Dinner is a reference interval for BoilWater, CookSpaghetti and EatSpaghetti. The reference structure is depicted in Figure 3.1.

In this case constraint c_{10} would be derived, but the interval Mowing would not be comparable with Dinner and Dishes (they do not share a reference interval, nor have explicit relationships been asserted between them), and hence constraints c_{11} and c_{12} would not be derived. Unfortunately, computing the constraint between Mowing and Dinner along the path Mowing–Gardening–Dinner results in the constraint $\{before, meets, overlaps\}$ which is clearly weaker than constraint c_{11} obtained in the flat system. Note that overlapping reference intervals and multiple reference intervals per simple interval are by no means contrived situations; for instance, they can easily arise in planning situations [Allen and

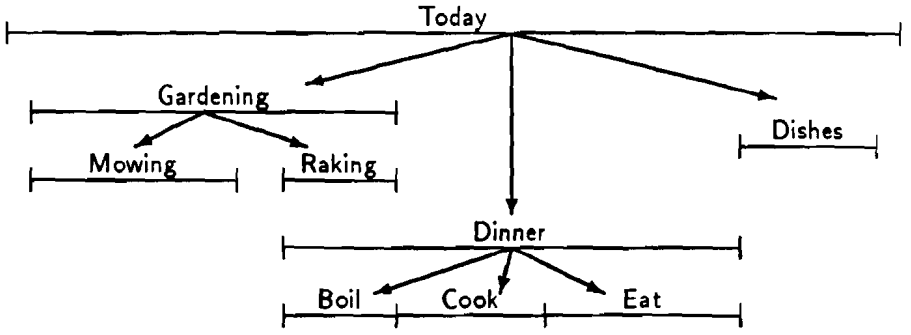

Figure 3.1: Example of interval reference structure

Koomen, 1983]. Moreover, this example shows that merely requiring that intervals are wholly contained in their reference intervals, as Allen suggests, is insufficient to assure proper inferencing. Allen already noted that merging two reference subtrees whenever such an overlap is detected tends to quickly flatten the entire reference hierarchy, defeating the purpose for reference intervals.

3.3 Automatic referencing

The TIMELOGIC system allows intervals to be explicitly designated as references for other intervals. However, devising a reference structure for a temporal constraint network such that no information is lost turns out to be a very nasty problem, and certainly one that a temporal reasoning system ought not to leave up to higher level reasoning systems, as Allen suggests. It is hard, if not impossible, to predict what effect predefining a reference structure has on temporal inferencing, and it certainly requires taking into account the order in which constraints are posted as well as detailed knowledge of the temporal constraint propagation algorithm, details of which would best be kept in the specialized temporal reasoning system. It would be very desirable if the temporal reasoning system itself could structure the network dynamically on the basis of posted and derived constraints, and do it without losing information. This and the following sections describe a method for doing this.

As Allen [1983] and Kahn and Gorry [1977] suggested, a reference hierarchy is naturally based on temporal containment: higher level intervals have a greater span in time, and lower level intervals are contained in higher ones. Temporal locality of reference is an obvious reason for basing the structure on containment: most queries to a temporal database concern relations between intervals associated with propositions currently under consideration, such as planning a task, where propagation is likely to be useful, whereas very few queries would be concerned with the constraints between two (semantically) unrelated and temporally distant propositions. Another reason a reference structure based on temporal containment is desirable is that it reflects propositional inheritance: any proposition that holds over an

interval X necessarily holds over any subinterval of X, and any relation between X and another, disjoint interval Y, is necessarily true between any subinterval of X and Y.

In the following discussion, the stored relational constraints between lower and higher level intervals in a containment-based reference hierarchy will be referred to as uplinks (with downlinks going the other way), and relational constraints between disjoint intervals as sidelinks, with the proviso that sidelinks are singular, *i.e.*, relational constraint sets with cardinality 1. See Appendix A for formal definitions of these links and detailed descriptions of the automatic referencing algorithms.

The automatic referencing method employed by TIMELOGIC revolves around two notions, namely, constructing the reference hierarchy on the basis of asserted uplinks and downlinks, and reducing the branching factor of an interval by breaking sidelinks and indirect uplinks with other intervals if those links can be derived from a path through the reference hierarchy. In the gardening/dinner example above, for instance, the same hierarchy is constructed dynamically as the constraints c_1, c_3, c_4 and c_6 are posted. The reference hierarchy is used only for computing indirect relations, and not for determining which intervals are considered for constraint propagation.

After the relation between intervals X and Y changes, propagation takes place to all intervals directly related to X or Y, including those that would span across reference subtrees, but excepting those whose link has been broken. If a sidelink relation is obtained, *i.e.*, some interval is disjoint from another and the constraint set is singular, an attempt is made to break the relation. This is successful if and only if the same relation can be obtained by applying the constraint propagation algorithm along a path through the reference hierarchy starting from the one interval and ending up at the other. In the example above, because propagation occurs unless a link is broken, the constraints c_{10} (Gardening $\{overlaps\}$ Dinner), c_{11} (Mowing $\{before\}$ Dinner) and c_{12} (Mowing $\{before\}$ Dishes) are all derived and recorded. Both c_{11} and c_{12} are sidelinks, so both are candidates for breaking. There exists a path along which c_{12} can be obtained exactly, hence the relation c_{12} can be broken. However, there exists no reference path that obtains the same relation as c_{11} and therefore the relation c_{11} cannot be broken without loss of information. Note also that c_2 cannot be broken; in general, this is true for all relations between disjoint intervals that have a common reference interval.

3.4 Empirical results

The automatic referencing mechanism implemented in TIMELOGIC suffers no loss of information, since each relation is initially computed, and it is not deleted until it has been uniquely determined and it can be obtained indirectly through the reference hierarchy. This initial computation of a relation followed by its deletion may seem to be a lot of extra and unnecessary work—how have we gained anything? It turns out that the greatly reduced average branching factor at each node (and hence reduced propagation) more than offsets any extra work entailed in the automatic management of the reference hierarchy. Unfortunately, it is difficult to get a handle on the kind of improvement this mechanism affords, since it is very dynamic in nature, and therefore very much dependent on the order and kind of relations asserted.

Nevertheless, two experiments have borne out that this approach is generally superior in performance to the flat system. The first experiment consisted of generating a network of 101 intervals and randomly assigning singular relations between any two intervals until all pairs of intervals have singular relations, either posted or derived, that are locally consistent [Vilain and Kautz, 1986]. In all, 450 relations were posted to obtain the total of 5050 singular relations. The results are given in Table 3.1, which compares average values per node and

Statistic	NoRefs	AutoRefs
Branching factor, average	100.0	18.3
# UpLinks, average	19.4	1.4
# SideLinks, average	47.3	1.6
Reference depth, average	1.0	13.8
Propagations, total	775,015.0	252,411.0
Constraints invocations	1,105,346.0	475,585.0

Table 3.1: Propagation statistics on 101 randomly related intervals

total values of several statistics for the flat system (NoRefs) and the automatic referencing system (AutoRefs). Note in particular the sharply reduced average number of uplinks and sidelinks per node.

The second experiment involved planning the task of exchanging the top two blocks of a stack using a two-armed robot, with the additional constraint that there is no room to place either block on the surface supporting the stack in an intermediate state (see [Allen and Koomen, 1983] for a more detailed description of this problem). The planner generates 33 intervals, asserting 65 temporal relations between them. In a flat system, another 1063 constraints are asserted due to constraint propagation, causing some 43000 invocations of the transitivity function Constraints and resulting in an average branching factor of 31, *i.e.*, the network ends up almost completely connected. Using the automatic referencing mechanism as described above, the average branching factor and the number of invocations of the transitivity function are roughly cut in half.

Chapter 4

Contexts and Backtracking

A computational reasoning system often needs to add temporary assertions to the knowledge base, such as during the development of alternative branches in a plan, or "what if" reasoning. The standard ways of achieving this is by backtracking over these temporary assertions, or by asserting them in contexts which may later be discarded. The TIMELOGIC system fully supports contextual reasoning as well as backtracking.

4.1 Contexts

The TIMELOGIC system starts out with a predefined context with the name T. This context forms the root of a tree of user-defined contexts. Exactly one context is designated the "current" context at all times. All TIMELOGIC database queries and assertions take place in the current context. Unless a constraint between two intervals is explicitly added to a context, the constraint is inherited from the nearest ancestral context, if there is one. Facilities are provided for defining and deleting contexts, pushing and popping contexts, and explicit context switching. In addition, most database query and assertion functions have an optional context parameter, which when given becomes the current context within the scope of the function. Adding assertions to a context with children is not allowed, as this may cause inconsistencies in the children.

4.2 Backtracking

The TIMELOGIC system keeps a full history of intervals defined and constraints asserted. Each successful assertion[1] results in a new backtracking "point." If such a backtracking point is given to the function TIMELOGIC-BACKTRACK, the TIMELOGIC system reverts to the state it was in *before* the assertion was made that resulted in the backtracking point.

[1] Note that a succesful assertion may not result in any changes to the database, but serve only as a consistency check.

This backtracking mechanism is also used to automatically undo all propagated effects from an assertion that resulted in a database inconsistency, provided the TIMELOGIC property :AUTO-BACKTRACK is :ON.

4.3 Auto-referencing and backtracking

The automatic referencing algorithm described in Section 3.3, as well as the algorithms in [Allen, 1983], were presented with the assumption of monotonic updates to the temporal database. In order to support efficient backtracking a forward-chaining constraint propagation system must store both the derived constraints themselves and adequate dependency information. Hence, removing relations from the network as suggested in Section 3.3. is out of the question. Limited to $O(n^2)$ space, we can still benefit from the auto-referencing algorithm as presented by simply marking relations "broken" where they would have been removed. Propagation does not need to take place across links marked "broken," and in addition, these marked links can still be used to obtain relational information, circumventing the need for traversing the reference hierarchy.

Chapter 5

Functional Interface

5.1 Terminology and conventions

5.1.1 Properties

Many TIMELOGIC system features can be enabled, disabled, or otherwise manipulated by the user through TIMELOGIC *properties*. See Section 5.8 for a description of these properties, their default values, and how to change them.

5.1.2 Contexts

If any function documented below takes the keyword argument *context*, the function may be invoked with NIL, T or the name of a previously defined context as the value for this argument, where NIL refers to the current context and T refers to the root context. The supplied context, if any, becomes the current context within the function's scope.

Note that asserting constraints in a non-leaf context may lead to inconsistencies. Certain constraints may have been inferred in a leaf context given some assertions and inherited constraints. These inferences may no longer be valid if the inherited constraints change. As TIMELOGIC is not able to detect and account for such inconsistencies, asserting constraints in a non-leaf context signals an error if the TIMELOGIC property :LEAVES-ONLY is :ON, or causes a warning to be printed if this property is :WARN.

5.1.3 Types

If any function documented below takes the keyword argument *type*, the function may be invoked with :REL, :DUR or NIL as the value for this argument. If :DUR is given, queries and assertions are assumed to be of the durational type; otherwise they are assumed to be of the relational type.

5.1.4 Errors

TIMELOGIC can run into trouble in many places. In general, if an error is detected, the user is given an error message and is then asked whether the Common Lisp ERROR function should be invoked. (Presumably facilities are provided for inspecting the current program state, data structures, *etc.*.) If the user answered no, or control somehow returns to the TIMELOGIC error function, the user is asked if the most recent changes to the TIMELOGIC database, if any, should be undone. Either way, TIMELOGIC then proceeds by returning NIL to wherever the TIMELOGIC error function was invoked. *This may or may not work!* The only exception is the case where a temporal inconsistency is detected during constraint posting, and the TIMELOGIC property :AUTO-BACKTRACK is :ON. In this case, all database changes, both direct and inferred, that occurred because of the call to the function ADD-INTERVAL-CONSTRAINT are undone without user interaction and the function returns NIL. See the function description for more detail.

5.2 Initialization

The TIMELOGIC system is loaded by either using the Lisp implementation's system wrapper facilities, or, if this is not available, by loading the file "timelogic.lisp" directly. For instance, one can type ":Load System TimeLogic" on the Symbolics machines. Once TIMELOGIC is loaded, all functions documented below as well as the durational constraint symbol <> are available as external symbols in the TIMELOGIC package (which has "TL" as a convenient nickname). The following function must be called once before attempting any other TIMELOGIC operation, and may be called again as often as desired.

(TIMELOGIC-INIT) [function]
Initializes the TIMELOGIC system, resets the database, creates the root context and makes it the current context. Returns a list of TIMELOGIC properties and their values currently in effect (see the function TIMELOGIC-RESET-PROPS).

5.3 Manipulating contexts

(DEFINE-CONTEXT &Optional *name parent*) [function]
Defines *name* as a TIMELOGIC context with *parent* as it's parent context. If *name* is not given, generates a new context name. If *parent* is not given, the current context becomes the parent context. Signals an error if *parent* is not a previously defined context or if *name* is an existing context with a different parent. Returns the new context's name.

(DEFINED-CONTEXTS) [function]
Returns a list of all defined TIMELOGIC contexts.

(CONTEXT-DEFINED-P *name*) [function]
Returns the context's name if *name* is defined context, otherwise NIL.

(DELETE-CONTEXT *name*) [function]
Erases all recorded constraints in context *name* from the database, deletes context *name* and returns it's name if *name* is a leaf context (*i.e.*, if it doesn't have any children), otherwise signals an error.

(SWITCH-CONTEXT *name*) [function]
Makes *name* the current context and returns it's name if *name* is a previously defined context, otherwise signals an error.

(PUSH-CONTEXT) [function]
Same as (SWITCH-CONTEXT (DEFINE-CONTEXT)).

(POP-CONTEXT &Optional *dont-delete-p*) [function]
Switches the current context *cc* to the parent of *cc*, then deletes *cc* provided *dont-delete-p* is NIL. Signals an error if *cc* does not have a parent (*i.e.*, *cc* is the root context).

(CONTEXT-TREE &Optional (*name* T)) [function]
Returns a tree node for context *name* (or the root context if omitted), where a tree node is either the name of the context if it is a leaf context (*i.e.*, has no children), or a list whose CAR is the name of the context and whose CDR is a list of tree nodes, one for each of the context's children.

5.4 Manipulating intervals

(DEFINE-INTERVAL &Optional *int ref* &Key *context*) [function]
Unless previously defined, defines *int* as the name of an interval. If *int* is not given, generates a new interval name. If *ref* is given, it is implicitly defined as an interval; if the TIMELOGIC property :AUTO-REFERENCE is :OFF, *ref* also becomes a reference interval for *int* in *context*. Returns the interval's name.

(INTERVAL-DEFINED-P *int*) [function]
Returns T if *int* was previously defined as an interval, otherwise NIL.

(DEFINED-INTERVALS) [function]
Returns a list of all currently defined intervals.

(RELATED-INTERVALS *int* &Key *context type*) [function]
Returns a list of all currently defined intervals for which some explicit constraint with *int* of type *type* has been recorded in *context*. If *type* is NIL, the constraint can be of either type.

(REFERENCE-INTERVALS *int* &Optional *inverse-p* &Key *context*) [function]
Returns a list of reference intervals of *int* in *context*. If *inverse-p* is non-NIL, returns a list of intervals for which *int* is itself a reference interval.

5.5 Manipulating constraints

(ADD-INTERVAL-CONSTRAINT *x constraint y* &Key *context* (*type* :REL)) [function]
If the :AUTO-DEFINE property is :ON, implicitly defines *x* and *y* as intervals with initial refererence interval :ROOT; otherwise signals an error if *x* or *y* have not been defined previously. Asserts *constraint* as a new constraint between intervals *x* and *y*. Signals an error when *constraint* cannot be parsed according to *type*. Let *oldcon* be the constraint currently known in *context*. Let *newcon* be the intersection of *constraint* and *oldcon*. Signals an error if *newcon* is the null set, unless the :AUTO-BACKTRACK property is :ON, in which case any intermediate database updates are removed and NIL is returned. If *newcon* is not the null set, propagates *newcon* to all constraints between comparable[1] intervals *x* and *z* such that there exists an explicit constraint between *y* and *z*, and to all constraints between comparable intervals *w* and *y* such that there exists an explicit constraint between *w* and *x*, and returns a TIMELOGIC backtracking point. If this backtracking point is given to the function TIMELOGIC-BACKTRACK (see below), the TIMELOGIC database reverts to the state it was in just before this particular invocation of ADD-INTERVAL-CONSTRAINT. Note that a backtracking point is returned even if adding *constraint* did not actually result in any database changes (*i.e.*, *newcon* is identical to *oldcon*).

(GET-INTERVAL-CONSTRAINT *x y* &Key *context* (*type* :REL)) [function]
If either *x* or *y* are not defined intervals, returns NIL; otherwise, returns the currently known relational constraint between *x* and *y*, unless *type* is given as :DUR in which case the durational constraint is returned.

(TEST-INTERVAL-CONSTRAINT *x constraint y* [function]
&Key *context* (*type* :REL) (*test* :INTERSECT))
Let *curcon* be the currently known constraint between *x* and *y*. Tests whether *constraint* is compatible with *curcon*. Three kinds of tests are possible, indicated by the argument *test*:

test	*true (returns* T*) iff*
:EQUAL	*curcon* = *constraint*
:SUBSET	*curcon* $\subset$ *constraint*
:INTERSECT	*curcon* $\cap$ *constraint* $\neq \emptyset$

(TIMELOGIC-BACKTRACK &Optional *btpoint*) [function]
Backs up the TIMELOGIC constraint database to *btpoint* (see the functions ADD-INTERVAL-CONSTRAINT, above, or TIMELOGIC-CHECKPOINT, below). If *btpoint* is omitted, backs up to the last backtracking point generated. Returns T if succesful, NIL otherwise.

(TIMELOGIC-CHECKPOINT) [function]
Returns an object which, when given to the function TIMELOGIC-BACKTRACK, returns the TIMELOGIC database to the current state.

[1] Two intervals are comparable (in the current context) if an explicit constraint exists between them or they have one or more reference intervals in common.

(TIMELOGIC-CHECKPOINT-P &Optional *btpoint*) [function]
Returns T if *btpoint* is a valid backtracking point, *i.e.*, if TIMELOGIC-BACKTRACK would succeed if given *btpoint*.

5.6 Tracing constraint propagation

(SHOW-INTERVAL-CONSTRAINTS *int* &Key *with-ints context* (*type* :REL)) [function]
Prints to the standard output stream a listing of directly recorded constraints between *int* and all intervals to which it is related, or between *int* and *with-ints* (which is either an interval or a list of intervals) if given.

(DISPLAY-INTERVALS &Key *ints context* (*clear* T)) [function]
Puts up a graphic display of all *ints*, lined up properly to reflect the relational constraints between them (relative to *context*). If *clear* is T (the default), the display area is cleared first. If *ints* is T, all intervals explicitly traced with TRACE-INTERVAL will be displayed. If *ints* is NIL, all defined intervals will be displayed.

(GRAPH-INTERVALS &Optional *ints*) [function]
Puts up a graph of the reference structure (see Section 3) over *ints*, which defaults to all defined intervals. *Note: this is currently implemented only in the Xerox and Symbolics environments.*

(TRACE-INTERVAL *int* &Optional *with-ints*) [function]
Any subsequent activity between *int* and any of the intervals directly related to it (or any in the list *with-ints* if given) will cause TIMELOGIC to print to the standard output stream a line indicating the activity. Note that this tracing occurs only if the :TRACE property is enabled.

(UNTRACE-INTERVAL &Optional *int*) [function]
Any subsequent activity between *int* and any of the intervals directly related to it (or any in the list *with-ints* if given previously) will no longer be printed to the standard output stream. If *int* is not given, disables tracing of all intervals explicitly traced with TRACE-INTERVAL. If *int* is a list, all intervals on this list will be untraced.

5.7 Statistics

(TIMELOGIC-STATS) [function]
Prints to the standard output stream a summary of various statistics kept when the TIMELOGIC property :STATS is :ON. For both the relational and durational types, prints the number of constraints attempted, "multiplied" (referring to the number of invocations of the Constraints function) and propagated. Also given is the number of singular constraints (*i.e.*, constraint sets of cardinality 1), the number of broken constraints (*i.e.*, those implied by a path through the reference hierarchy), and the average branching factor (*i.e.*, the number of explicit constraints between intervals). The statistics counters can be reset by setting the TIMELOGIC property :STATS to :RESET.

5.8 Customization

(TIMELOGIC-RESET-PROPS &Rest *keywords*) [function]
Resets all TIMELOGIC properties to their default values, unless overriden by entries on *keywords*, and returns a list of all TIMELOGIC properties and their current values. For instance, (TIMELOGIC-RESET-PROPS :TRACE :ON :DISPLAY :ON :SORT :ON) will give you a standard TIMELOGIC setup, except that whenever a change is made to the constraint between two traced intervals, a trace message is printed, and the relations between all traced (and defined) intervals are displayed graphically.

(TIMELOGIC-PROP *propname* &Optional *newpropvalue*) [function]
If *newpropvalue* is given, sets the property value accordingly. Always returns the current (old) value of the property. Some properties (designated *init* properties) may only be set immediately following a call to TIMELOGIC-INIT. They generally affect the operation of TIMELOGIC such that changing them in mid-session would have unpredictable results. The following properties control the TIMELOGIC system:

:ALL-PATHS [TIMELOGIC init property]
When :ON, TIMELOGIC computes the constraint between two indirectly related intervals as the intersection of the constraints along all paths through the hierarchy. When :OFF, causes TIMELOGIC to compute such a constraint by traversing the first path found through the reference hierarchy. Note that traversing a single path is much faster but may result in an under-constrained relation! Initially set to :ON.

:AUTO-BACKTRACK [TIMELOGIC property]
When :OFF, TIMELOGIC leaves the current state of the database alone when an error or inconsistency occurs. When :ON, the current assertion and all consequences computes sofar will be undone automatically. Initially set to :ON.

:AUTO-DEFINE [TIMELOGIC property]
When :OFF, TIMELOGIC signals an error when a constraint is posted between undefined intervals. When :ON, any undefined interval is implicitly defined with initial reference interval :ROOT. Initially set to :ON.

:AUTO-REFERENCE [TIMELOGIC init property]
When :OFF, TIMELOGIC relies on the user to provide a reference hierachy of intervals. When :ON, a reference hierarchy is automatically and incrementally defined (see Section 3). Initially set to :ON.

:DEPTH-FIRST [TIMELOGIC property]
When :OFF, TIMELOGIC processes constraint inferences in a breadth-first fashion. When :ON, a depth-first strategy is employed. Initially set to :OFF.

:DISPLAY [TIMELOGIC property]
When :ON, and the TIMELOGIC property :TRACE is not :OFF, TIMELOGIC provides a dynamically changing display of relational interval constraints, updated whenever a relevant

constraint changes.[2] If :TRACE is :ON, only those intervals are displayed which have been explicitly traced. If :TRACE is :ALL or :VERBOSE, all intervals are displayed. Initially set to :OFF.

:DURATIONS [TIMELOGIC init property]
When :ON, TIMELOGIC allows durational constraints to be posted, and infers them from given durational and relational constraints. Initially set to :OFF.

:FLOATS [TIMELOGIC init property]
When :ON, TIMELOGIC allows durational constraints to be real-valued; otherwise, all reals will be rationalized. Initially set to :OFF. Note that round-off errors in floating-point arithmetic may cause TIMELOGIC to fall into an infinite loop. Setting the :TOLERANCE property (see below) to an appropriate value can alleviate (but not eliminate) this problem.

:LEAVES-ONLY [TIMELOGIC init property]
When :ON, TIMELOGIC will signal an error if an attempt is made to change a constraint between two intervals in a non-leaf context (*i.e.*, a context with children). When :WARN, this attempt would succeed (caveat emptor!) but you will be warned. When :OFF, such attempts succeed without fuss (but not necessarily without causing inconsistencies). Initially set to :WARN.

:RELATIONS [TIMELOGIC init property]
When :ON, TIMELOGIC allows relational constraints to be posted, and infers them from given durational and relational constraints. Initially set to :ON.

:SORT [TIMELOGIC property]
When :ON, TIMELOGIC attempts to sort intervals along the time axis before displaying them. Initially set to :OFF. Note: only operative when the TIMELOGIC property :DISPLAY is :ON.

:STATS [TIMELOGIC property]
When :ON, TIMELOGIC starts (or continues) to keep track of various statistics during its inferencing. Initially set to :OFF. Execute (TIMELOGIC-PROP :STATS :RESET) to reset the statistics counters.

:TOLERANCE [TIMELOGIC init property]
When the :FLOATS property is :ON, and the :TOLERANCE property is a floating point number f, TIMELOGIC will not add a new constraint if the resulting constraint *newcon* would be within a factor f of the existing constraint *oldcon*. Initially set to 0.001.

:TRACE [TIMELOGIC property]
Unless :OFF, TIMELOGIC prints various tracing information regarding the state of the temporal database. More detail is obtained by setting this property to :ON, :ALL, or :VERBOSE, successively. When :ON, a trace line is printed whenever a constraint changes between some interval and an interval which has been explicitly traced using the function TRACE-INTERVAL, or when such an interval is defined, or gets or becomes a reference interval.

[2] Obviously, since intervals are generally only partially ordered, this display is only partially correct! It is true that two intervals whose endpoints are known not to coincide do not line up in the display, but the inverse is not true.

When :ALL, a line is printed tracing each top level activity taking place (*i.e.*, TIMELOGIC changes initiated by the user). When :VERBOSE, a trace line is printed whenever anything happens to any interval, direct or inferred. Initially set to :OFF. Each trace line starts with one of the following indicators:

TL+ interval: ***int***	Defining interval *int*
TL+ context: ***name***	Defining context *name*
TL- context: ***name***	Removing context *name*
TL> context: ***name***	Making context *name* the current context
TL+ reference: ***int ref***	Adding *ref* as reference interval for *int*
TL- reference: ***int ref***	Removing *ref* as reference interval for *int*
TL+ relation [*i*]: ***xint yint rel***	Adding *rel* as new constraint between intervals *xint* and *yint* on level *i* (if $i = 1$ then user-defined, otherwise inferred)
TL^ relation [*i*]: ***xint yint rel***	Restoring *rel* as backtracked constraint between intervals *xint* and *yint* on level *i*
TL~ relation [*i*]: ***xint yint***	Breaking direct constraint between intervals *xint* and *yint* on level *i* (see Section 3)
TL& relation [*i*]: ***xint yint***	Reconnecting broken constraint between intervals *xint* and *yint* on level *i*
TL+ duration [*i*]: ***xint yint dur***	Adding *dur* as new constraint between intervals *xint* and *yint* on level *i*
TL^ duration [*i*]: ***xint yint dur***	Restoring *dur* as backtracked constraint between intervals *xint* and *yint* on level *i*

:WAIT [TIMELOGIC property]

Either :ON, :OFF or a positive number. Unless :OFF, TIMELOGIC waits (indefinitely if :ON or for this many seconds if a number) between displaying intervals and continuing its next inference. Note: only operative when the TIMELOGIC property :DISPLAY is :ON.

Bibliography

[Allen, 1983] James F. Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, 26(11):832–843, 1983.

[Allen and Koomen, 1983] James F. Allen and Johannes A. Koomen, "Planning Using a Temporal World Model," In *Proc. 8th IJCAI*, pages 741–747, Karlsruhe, W. Germany, 1983.

[Kahn and Gorry, 1977] K. Kahn and G. A. Gorry, "Mechanizing Temporal Knowledge," *Artificial Intelligence*, 9:87–108, 1977.

[Vilain and Kautz, 1986] Mark Vilain and Henry Kautz, "Constraint Propagation Algorithms for Temporal Reasoning," In *Proc. 5th AAAI*, 1986.

Appendix A

Automatic Referencing Algorithm

In presenting the algorithms, the same nomenclature as in [Allen, 1983] will be adopted to facilitate comparison. For any two intervals i and j, let $N(i,j)$ be the constraint along the arc between i and j in the network, and let $R(i,j)$ be a new constraint between i and j to be added to the network. $N(i,j)$ may be the result of applying the transitivity function along a path through the network involving intervals other than i and j. Let $C(i,j)$ be the explicitly stored constraint between i and j, and $C*(i,j)$ be $C(i,j)$ if it exists, otherwise the result of applying the transitivity function along a path through the network from i to j if such a path exists, otherwise $\emptyset$. If no path exists between i and j, $N(i,j)$ is defined to be the complete set of possible temporal constraints; if no explicit constraint between i and j is stored in the network, $C(i,j)$ is undefined. Hence, $N(i,j)$ can be defined in terms of $C*(i,j)$. The function $N_3(i,k,j)$ will be used to refer to the network relation between intervals i and j forced along the path through the interval k; in other words,

$$N_3(i,k,j) \equiv Constraints(N(i,k), N(k,j))$$

Modifications to two of Allen's original algorithms are required, namely Add R(i,j) which adds a new constraint between intervals i and j, and Comparable(j,k) which determines whether a changed constraint between the intervals i and j should be propagated to the relation between intervals j and k. Here is the original algorithm for posting a new constraint, slightly edited to isolate the action of altering the network:

```
to Add R(i,j)
        Assert R(i,j)
        while the queue ToDo is not empty do begin
           Get next <p,q> from queue ToDo
           for each node k such that Comparable(k,q) do
              Assert R(k,q) ← N(k,q) ∪ N3(k,p,q)
              for each node k such that Comparable(p,k) do
                 Assert R(p,k) ← N(p,k) ∪ N3(p,q,k)
        end
     end
```

where Allen would have defined the procedure Assert as follows:

```
to Assert R(i,j)
      if R(i,j) ⊂ N(i,j) then begin
        C(i,j) ← R(i,j)
        Add <i,j> to the queue ToDo
      end
    end
```

The automatic referencing mechanism can now be embedded in the procedure Assert, which will be amended later when the rest of the required mechanisms are in place. Note that this specification is functionally not quite the same as in [Allen, 1983], because changes in constraints take effect immediately rather than during the processing of the corresponding queue entry. The distinction is not crucial except for performance—Allen's formulation eventually computes the same relations, but tends to waste effort during propagation processing intermediate constraints that are already subsumed by other entries on the queue.

The comparability condition is modified as follows:

```
Comparable(i,j) ≡   C(i,j) exists or not SpecialLinkp(C*(i,j))
```

Finally, the various link predicates are defined as follows:

```
UpLinkp(rel) ≡   rel ⊆ {starts, during, finishes}

DownLinkp(rel) ≡   rel ⊆ {started by, contains, finished by}

SideLinkp(rel) ≡
      rel = {before} or rel = {after} or rel = {meets} or rel = {met by}

SpecialLinkp(rel) ≡   UpLinkp(rel) or DownLinkp(rel) or SideLinkp(rel)
```

A.1 Reducing the branching factor

When a sidelink is being asserted between two intervals i and j, there may be a path from i to j through the reference hierarchy going through a reference interval $k \in \text{Refs}(i)$. If the cumulative constraint along the path from i through k to j is the same sidelink, then the relation between i and j may be deleted, since, by nature of the way we define and construct the reference hierarchy, this relation is obtainable indirectly without loss of information. The only exception is the case when i and j share a common reference interval k. In this case the relation from i through k to j is under-constrained and cannot be deleted without loss of information. For instance, if i $\{starts\}$ k and j $\{finishes\}$ k and i $\{before\}$ j, then the relation i–k–j is $\{before, meets, overlaps\}$. This check and possible deletion is accomplished by the procedure CLAssertSideLink:

```
to AssertSideLink(i,j)
      if Refs(i) ∪ Refs(j) = ∅ then
        for each interval k ∈ Refs(i) do
          if C(i,j) = N3(i,k,j) then begin
```

```
            Remove C(i,j) from the network
            Exit
          end
    end
```

In the context of the gardening/dinner example of Section 3.2, after asserting constraint c_9 (Raking $\{equals\}$ BoilWater), a sidelink would be asserted between Mowing and Dinner (c_{11}) and between Mowing and Dishes. The latter assertion would be broken because Gardening $\in$ Refs(Mowing) and N3(Mowing,Gardening,Dishes) = $\{before\}$ = C(Mowing,Dishes), but the former assertion would not be broken because N3(Mowing,Gardening,Dinner) = $\{before, meets, overlaps\}$ which is $\neq$ C(Mowing,Dinner).

A.2 Constructing the reference hierarchy

When an uplink is asserted between intervals i and j, j is made a reference interval of i; or when a downlink is asserted between intervals i and j, i can be made a reference interval of j. This is accomplished by the procedure AddRef(i,j), which has to consider a number of special cases:

- j may already be a reference interval of i, since uplinks are not necessarily singular so they can be refined and re-asserted.
- j may already be higher up the reference tree above i; this can happen, for instance, if this uplink is asserted through propagation. In this case, the current relation between i and j can be deleted.
- i may have a reference interval k which is above j, *i.e.*, i $\{during\}$ k was known as well as j $\{during\}$ k, so asserting i $\{during\}$ j implies i $\{during\}$ j $\{during\}$ k, which implies the relation i $\{during\}$ k can be obtained indirectly, hence can be deleted. The same holds for any other interval below i.
- i may have a sidelink to an interval k, which could be obtained indirectly through the new reference interval j, hence the current relation between i and k can be deleted. The same holds for any other interval below i.

Let Refs*(i) be the closure of the Refs function, *i.e.*, the set of all reference intervals directly or indirectly above i.

```
to AddRef(i,j)
      if j ∈ Refs(i) then
         Exit
      else if j ∈ Refs*(i) then
         Remove C(i,j) from network
      else begin
         DeleteIndirectRefs(i,j)
         Refs(i) ← Refs(i) ∩ {j}
```

```
        DeleteDiagonals(i,j)
      end
    end
```

The recursive procedure DeleteIndirectRefs(i,j) removes all reference intervals k from the reference sets of i and the intervals below i, if k is j or k is above j. Let RefdBy(i) be the set of all intervals k such that $i \in$ Refs(k).

```
to DeleteIndirectRefs(i,j)
    for each node k ∈ Refs(i) do
      if k = j or k ∈ Refs*(j) then begin
        Refs(i) ← Refs(i) − {k}
        if C(i,k) is singular then
          Remove C(i,k) from the network
      end
    for each node k ∈ RefdBy(i) do
      DeleteIndirectRefs(k,j)
  end
```

In general, making an interval j a reference interval of an interval i implies the grafting of two trees, one headed by i, and one which has j as a leaf. Until this point, i may have had a sidelink to some intervals k to which there existed no reference path. When j is added to the set of reference intervals of i it is possible that there now exists a reference path from i to k going through j, in which case we can delete the sidelink from i to k. The same holds true for any node below i. Removing now superfluous sidelinks is accomplished by the recursive procedure DeleteDiagonals:

```
to DeleteDiagonals(i,j)
    for each node k such that SideLinkp (C(i,k)) do
      if Refs(i) ∪ Refs(k) = ∅ then
        if C(i,k) = N3(i,j,k) then
          Remove C(i,k) from the network
    for each node k ∈ RefdBy(i) do
      DeleteDiagonals(k,j)
  end
```

A.3 Asserting relations

All ingredients are now available to modify the procedure Assert(i,j), which adds a newly posted or derived relation to the network. If an uplink or a downlink is asserted then update the reference hierarchy; otherwise, if a sidelink is asserted, then see if it can be removed as described in section A.1:

```
to Assert R(i,j)
    if R(i,j) ⊂ N(i,j) then begin
```

```
        Add <i,j> to the queue ToDo
        C(i,j) ← R(i,j)
        if UpLinkp(C(i,j)) then
          AddRef(i,j)
        else if DownLinkp(C(i,j)) then
          AddRef(j,i)
        else if SideLinkp(C(i,j)) then
          AssertSideLink(i,j)
      end
    end
```

A.4 Propagating equality

Equality between intervals can occur frequently in a logic-based reasoning system. Intervals are associated with non-grounded terms which are said to hold over the intervals. In such a system, unification of terms requires unification of intervals, in other words, asserting their equality. Temporal equality can be viewed as a form of temporal containment, and so should be usable for automatic referencing. The only obstacle is the fact that equality is symmetric, which begs the question: is it an uplink or a downlink? This can be resolved by assigning sequence numbers to intervals as they are introduced, indicating their age. Now the above mechanism for automatic referencing can be extended to include equality by redefining the predicate UpLinkp as follows:

```
UpLinkp(rel,from,to) ≡
      if Age(from) > Age(to)
        then Return rel ⊆ {starts, during, finishes, equals}
        else Return rel ⊆ {starts, during, finishes}
```

This definition will make the older of two equal intervals a reference interval for the younger one. The predicate DownLinkp is modified analogously. Using this extended definition increases the number of potential reference intervals, which in turn tends to reduce the overall network connectivity. Unfortunately, incorporating equality in the referencing mechanism like this will also have the effect of linearizing equivalence sets. Additional gain can be had by using a UNION-FIND algorithm to make the oldest interval in an equivalence set the reference interval for the set, and the youngest the reference interval for all subintervals.

Index