The Logic of Time Representation

Peter Bernard Ladkin

November 1987

Thesis presented in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in the Group in Logic and Methodology of Science
at the University of California at Berkeley,
November 1987.

To my father, and the memory of my mother

Abstract

This investigation concerns representations of time by means of intervals, stemming from work of Allen *[All83]* and van Benthem *[vBen83]*. Allen described an Interval Calculus of thirteen binary relations on convex intervals over a linear order (the real numbers). He gave a practical algorithm for checking the consistency of a subclass of Boolean constraints.

First, we describe a completeness theorem for Allen's calculus, in its corresponding formulation as a first-order theory LM. LM is countably categorical, and axiomatises the complete theory of intervals over a dense unbounded linear order. Its only countable model up to isomorphism is the non-trivial intervals over the rational numbers.

Algorithms are given for quantifer-elimination, consistency checking, and satisfaction of arbitrary first-order formulas in the Interval Calculus.

A natural countable model of the calculus is presented, the TUS, in which clock- and calendar-time may be represented in a straightforward way.

Allen and Hayes described a first-order theory of intervals in *[AllHay85, AllHay87.1]*. It is shown that the models of the theory are precisely the interval structures over an arbitrary unbounded linear order.

An extension of the calculus to intervals which are union-of-convex is considered, introduced by concerns over the represention of interruptible processes. A taxonomy of necessary relations between union-of-convex intervals is given, and it is considered how to generalise to arbitrary non-convex intervals. Use of the extension is illustrated with an example of synthesising concurrent processes from high-level specifications.

The extended calculus may be implemented in a high-level system with logic programming and data-types of *sequence* and *set*, and this feature is illustrated with a partial implementation of the TUS in such a system.

Acknowledgements

I give my heartfelt thanks to all those who have helped me in the work which has ultimately become my dissertation. To Ralph McKenzie, who helped me to learn universal algebra, and who made time and effort beyond any duty to help turn this work into a dissertation. Thankyou Ralph. May you no longer puzzle over how to get me to work. To my dear friends Roger Maddux, with whose patient collaboration this work began, and Shaughan Lavine, whose intellectual companionship over more than a decade has matured from a delight to a necessity for me. Well Roger, at least I've learned how to play the whistle. To Leslie Lamport, whose persistent queries as to the state of my thesis embarrassed me into writing one, and whose skiing skill drove me to nefarious use of the color purple. To Richard Waldinger, skilled practitioner of the art of conjunctive application of encouragement and coffee, of which the effects I hope last longer than a year and a half. To Pat Hayes, whose unflagging enthusiasm allowed me to acknowledge that I had something to say, and from whom I learnt that there can be no such thing as too much encouragement. To James Allen, whose magnificent article started this research. To Stuart Russell, whose comments improved both the style and the quantity of work reported herein. To my Kestrel colleague Richard Jullig, whose support has enabled me to pursue this work, and Cordell Green, Director of Kestrel Institute, whose financial support has enabled me to finish it.

Finally, this work has been partially supported by the U.S. Navy under contract N00039-86-C-0221, by Rome Air Development Center under U.S. Government contract F30602-84-C-0109, the Defence Advanced Research Projects Agency under contract N00014-81-C-0582, and by Kestrel Institute. The views and conclusions expressed in this dissertation are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Kestrel Institute, the University of California at Berkeley, the U.S. Navy, Rome Air Development Center, or DARPA. But I hope they like it anyway.

Contents

1 Introduction **1**

1.1 Prolegomena 1

1.1.1 Summary 3

1.2 Binary Relations on Convex Intervals 4

1.3 The Interval Algebra 7

1.3.1 Related Work 8

1.4 Constraint Satisfaction Procedures 9

1.5 The Convex *TUS*. 10

1.6 Models of the Allen-Hayes Axioms 11

1.6.1 Background and Comparisons 12

1.7 The Extended Interval Model 13

1.8 Application to the Synthesis of Concurrent Processes 15

1.8.1 Refinement of The Specification 16

1.9 Defining and Implementing the Full *TUS* 19

1.9.1 The operators 19

1.9.2 Temporal Logic in the *TUS* 20

1.9.3 Axiomatising the Time Units 20

1.10 An Example Implementation 20

1.11 Definitions 21

1.11.1 Definitions: Theories and Axioms 21

1.11.2 Definitions: Models and Completeness 22

1.11.3 Definitions: Properties of Binary Relations 23

1.12 Results from Model Theory 24

2 The Interval Algebra **25**

2.1 Relation Algebras: Background 25

2.2 Relation Algebras and Theories 29

2.3 The Point Algebra 30

2.4 The Interval Algebra 31

2.5 Summary 34

3 Constraint Satisfaction Procedures **35**
3.1 Synopsis . . . 35
3.2 The Translations . . . 36
3.2.1 Comments and Definitions . . . 36
3.2.2 From INT to RAT . . . 37
3.2.3 From RAT to INT . . . 40
3.3 The Algorithms . . . 43
3.3.1 Quantifier-Elimination in $Th(INT)$ . . . 43
3.3.2 Consistency Algorithms . . . 45
3.3.3 A Satisfaction Procedure . . . 46
3.4 Summary . . . 47

4 The Convex TUS **48**
4.1 Synopsis . . . 48
4.2 The Time Unit System . . . 49
4.3 The Main Result . . . 50
4.3.1 The Language $\mathcal{I}$ . . . 51
4.3.2 The Sets $\mathcal{G}^{\mu}$ of Sequences . . . 52
4.3.3 The Mapping of $\mathcal{G}^{\mu}$ into Pairs of Rational Numbers . . . 53
4.3.4 The Extension of $\mathcal{G}^{\mu}$ . . . 54
4.3.5 Adaptation of the Satisfaction Procedure . . . 56
4.4 Summary . . . 56

5 Models of the Allen-Hayes Axioms **57**
5.1 Synopsis . . . 57
5.2 Summary of Related Work . . . 58
5.3 Technical Results . . . 58
5.3.1 The Allen-Hayes Axioms . . . 58
5.3.2 Existential M5 versus Functional M5 . . . 59
5.3.3 The Axiom M4 . . . 60
5.3.4 Analysing the Allen-Hayes Axioms . . . 60
5.3.5 Extending the Theory . . . 64
5.4 Summary . . . 64

6 Union-of-Convex Intervals **65**
6.1 Introduction . . . 65
6.2 The Specification of Time Dependencies . . . 66
6.2.1 A Simple Case . . . 66
6.2.2 Convex Intervals . . . 66
6.2.3 Extending the Interval Model . . . 67
6.2.4 Representing Context Switching . . . 68
6.2.5 Intervals are the Appropriate Abstraction . . . 68
6.3 Relation Primitives for Unions of Convex Intervals . . . 69
6.3.1 The Choice of Relation Primitives . . . 69

6.3.2 A List of the Relations . . . 70
6.4 The Definition of the Relation Primitives . . . 71
6.4.1 The Intended Calculus . . . 71
6.5 Examples of Relations Between Non-Convex Intervals . . . 76
6.6 Relations for General Non-Convex Intervals . . . 78
6.6.1 The Choice of Primitives . . . 78
6.6.2 A List of the General Non-Convex Relations . . . 78
6.6.3 The Informal Definitions of the Relations . . . 79

7 Synthesis of Concurrent Processes 82
7.1 Synopsis . . . 82
7.2 Some Relations on Unions-Of-Convex Intervals . . . 82
7.3 Synthesis of Concurrent Process Specifications . . . 84
7.3.1 The Readers/Writers Specification . . . 85
7.3.2 Specification in the Interval Calculus . . . 85
7.4 Refinement of The Specification . . . 86
7.4.1 The Target Language . . . 86
7.4.2 Guarded Commands . . . 86
7.4.3 The General Strategy . . . 87
7.4.4 Formalisation of Exclusion and Ordering . . . 88
7.4.5 A Small Readers/Writers Example . . . 88
7.4.6 Refining the Assertions . . . 88
7.4.7 Generating Process Skeletons . . . 91
7.4.8 Execution of The Lower-Level Specification . . . 91
7.4.9 What We Have Accomplished . . . 92
7.4.10 Sequencing Assumptions . . . 92
7.4.11 Completeness of The Interval Calculus . . . 93
7.4.12 Generalising the Method . . . 93
7.4.13 Limitations of the Method . . . 93
7.5 Summary . . . 94

8 The Full *TUS* and Its Implementation 95
8.1 Notation . . . 95
8.2 Operators for Intervals . . . 96
8.2.1 The operators . . . 96
8.2.2 Axioms . . . 97
8.2.3 Additional Noteworthy Properties . . . 100
8.3 Predicates . . . 100
8.3.1 Axioms . . . 100
8.4 Additional Defined Entities . . . 100
8.5 Temporal Logic . . . 101
8.5.1 Interval Constants . . . 101
8.5.2 Axioms . . . 101
8.5.3 The Temporal Operators . . . 102

8.5.4 Conversion Functions 102
8.6 Defining Time Units 103
8.6.1 Standard Time Units 103
8.6.2 Axioms for Units 104
8.6.3 Interval Types Definable From Unit Types 105

9 A Sample Implementation **106**
9.1 Summary 106
9.1.1 *REFINE* syntax 106
9.2 The Code Examples 111
9.2.1 Definition of the Time Units. 111
9.2.2 *Time-Unit* as a Type 111
9.2.3 Legitimate Time Periods 113
9.2.4 Defining The Ends of Periods 114
9.2.5 Ordering the Units 114
9.3 Comparing Units in the Order 116
9.4 Definition of Convex Time Periods 116
9.4.1 Overlapping Time Units 118
9.4.2 Definition of Weighted-Intervals 119
9.4.3 Cleaning Up the Definitions 121

Chapter 1

Introduction

1.1 Prolegomena

We started this research with the desire to develop a formal model of time that could be used within formalised reasoning in computer science and artificial intelligence. Chosen target applications were common-sense reasoning, the specification and synthesis of concurrent systems, and automated project management.

We did not intend to solve philosophical problems in the logic of time. This does not mean that philosophical considerations are unimportant, but it does entail that they are less important than efficiency and perceived naturalness to the casual user. Neither does it mean that we have not solved problems of interest to philosophers of time. At least half of this work consists of investigations in the logic of time. We use work in the philosophy of time in order to guide us in the search for adequate representations, in the belief that most good ideas have been thought of before.

We regard this work also as an exercise in applied logic. Most of the work reported here uses tools from logic. Chapter 2 may be regarded as an exercise in the model theory of algebraic logic, and chapters 3 and 4 and 5 are exercises in model theory of axiomatic theories. Chapters 4 and 8 include axiomatic specifications of operators in Horn Clause form, for use in a logic programming system.

One of the major problems to be solved was how to provide a formalism that gave structure to time knowledge, without committing to a particular model of time. For example, much, though not all, semantics for concurrency, the so-called *linear time* semantics, assumes that states are discrete points, ordered like the natural numbers, i.e. a discrete order type with one left endpoint and no right endpoint (see e.g. *[ManPnu81]*). (For an alternative, non-linear, tree ordering of states for a similar context, the so-called *branching time* semantics, see *[Fra86]*, and for comparisons *[Lam80, EmeHal83]*). In this context, the ordering of time is identified with the state sequence, since this is what is really of interest for verifying programs. An event is something that occurs between states. There are philosophical grounds for not identifying time structure in general with the structure

of events, argued strongly in *[New80]*, but in this limited domain, there seem to be no good reasons for not making this identification.

In contrast, time as it is in the real world is often assumed to have a dense ordering without endpoints. This is the kind of order type of time in physics, for example, where the order type of the reals is used. In common-sense reasoning, it is also often the case that an assumption is made that between any two time points there is a third.

It is clear that these order types have incompatible structure, and yet we wished to allow both of these situations as implementations of our general time structure. They are, of course, compatible in the sense that the discrete model can be embedded in the dense model. But neither of them seemed adequate as a structure for representing notions of time used by calendars and watches.

Considering calendars and watches, one can observe first that two different, non-simultaneous events may be assigned the same time label. For example, a digital watch may read the time as 7:21am, and 15 seconds later has the same reading (of course, watches read in seconds rather than minutes, but the point is better made with the courser grained units). If we were to use time points to represent time, then we are not able to use the same point to represent two non-simultaneous times, as argued in *[New80]*. So we should use a notation in which the same label may attach to two non-simultaneous events, in order to capture the clock-related notions. The notion of *time interval* therefore seemed more in keeping with the common sense use of temporal concepts than the mathematical abstraction of points from the real number line.

The representation of time by means of intervals rather than points has a history in philosophical studies of time (*[Ham71, vBen83, Hum78, Dow79, Rop79, New80]*). Recently, in the context of computer science, James Allen defined a calculus of time intervals in *[All83]*, as a representation of temporal knowledge that could be used in AI. We shall call this the *Interval Calculus*. In this paper, Allen investigated specification and constraint satisfaction in the Interval Calculus. Allen and Pat Hayes in *[AllHay85, AllHay87.1, AllHay87.2]* reformulated the calculus as a formal theory in first-order logic.

Allen's calculus encompassed a new approach to temporal specification and theorem proving, even though the thirteen basic relations he investigated had been long recognised (e.g. *[vBen83]*). In our opinion, his major new contribution was his constraint satisfaction algorithm, which used the composition table for the relations to infer inconsistencies, and his subsequent use of the Interval Calculus for representing time in the context of automated planning *[All84, AllKau85, PelAll86]*.

It turns out that the structure is particularly amenable to treatment by the methods of mathematical logic, as we shall show. In particular the first-order theory is complete, countably categorical, and hence decidable (see later for definitions). It also turns out to admit elimination of quantifiers.

We have considered the use of the Interval Calculus for reasoning about all aspects of time. Two aspects that have concerned us are, firstly, that it is often unnatural to use pairs of real numbers for representing intervals of real time, especially when standard everyday time units are needed, as in project management or real-time programming; and, secondly, that using intervals without gaps (*convex* intervals) makes it hard to reason about interruptable processes, where the more natural model would use intervals with gaps, as in a context-switching multiprocessing computer.

We have addressed these issues in *[Lad86.1, Lad86.2, Lad87.1]*, and in chapters 4 and 6, respectively, below. We introduce a system of interval time units in chapter 4 *[Lad86.2]*, which allows us to represent all kinds of standard time intervals such as years, days, and picoseconds in a natural manner, and we introduced primitive operations for building up arbitrary intervals, both convex and non-convex, from these interval time units. We refer to this time unit system as the *TUS*. The *TUS*, as we show in chapter 4, is a canonical model for which the Interval Calculus is complete. We introduce in chapter 6 a taxonomy of binary relations on intervals that had gaps, called unions-of-convex intervals *[Lad86.1]*. We refer to this system as the *Extended Interval Calculus*. We provide an example of the use of the Extended Interval Calculus for the high-level specification and synthesis of concurrent processes in chapter 7 *[Lad87.1]*.

1.1.1 Summary

We use Allen's Interval Calculus as the basis for our current work.

In chapter 2, we investigate the logical properties of his Interval Calculus, proving a completeness theorem, and concluding amongst other things that the first-order theory of the calculus is decidable.

In chapter 3 we present a quantifier-elimination procedure for the interval calculus, two consistency-checking algorithms, and a satisfaction algorithm, for arbitrary first-order formulas in the interval calculus. The satisfaction algorithm constructs intervals that satisfy a given consistent first-order formula, or detects the inconsistency, whichever is the case.

Chapter 4 presents the convex fragment of a natural interval time unit system, the *TUS*, which is a countable canonical model for the Interval Calculus. The Interval Calculus is complete for the *TUS*, i.e. every formula of the Interval Calculus true in all interpretations in the *TUS* is a theorem of the Interval Calculus, and every non-theorem has a counter-example in the *TUS*.

In chapter 5, we characterise all countable models of the axiomatisation of intervals by of Allen and Hayes.

In chapter 6, we argue for the necessity of using union-of-convex intervals for specifications involving temporal dependencies. We extend the interval specification language to allow for non-convex intervals, and present a taxonomy of useful relations on such intervals. We call this system the *Extended Interval Calculus*.

In chapter 7, we apply this specification language to an example of program synthesis, for certain kinds of concurrent processes.

Chapter 8 investigates the implementation of the Extended Interval Calculus in a wide-spectrum language, using a declarative style (Horn clause axioms). We include axioms for the needed operators, and we assume a data type of `set` is available.

Finally, in Chapter 9 we include an example of code written to implement part of the *TUS*, using a procedural style with a `set` data type. The code is used as part of a training course for the $REFINE^{TM}$ [1] system, in which this example has been written, and in which the *TUS* and part of the *Extended Interval Calculus* have been implemented.

1.2 Binary Relations on Convex Intervals

James Allen gave an qualitative classification of the possible binary relations between intervals on a linearly ordered set in *[All83]*. We reproduce this qualitative classification in pictorial form below. Of course, it requires precise formalisation and proof before it can be acknowledged that this classification is correct. We provide that partly in chapter 2, where we detail the results of the author and Roger Maddux, and partly in chapter 4 where we investigate the axiomatised form of the Interval Calculus of Allen, and Pat Hayes *[AllHay85, 87.1, 87.2]*.

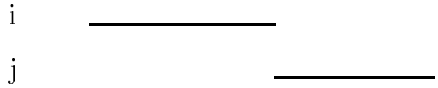
In the pictures below, we consider the linearly ordered set to be implicitly represented along the horizontal, and we draw a finite line segment to indicate an interval on this set. There is no metric assumed on the set, so relative position of multiple intervals is the only observable quality (again, proof of this follows from the characterisation in chapters 2 and 4).

A convex interval looks like this (with time going from left to right):

Allen's relations are:

- i equals j, strict identity of intervals

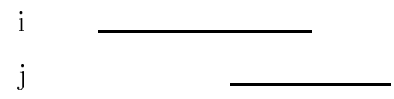
- i meets j, which means i is before j, but there is no gap (non-empty interval) separating them

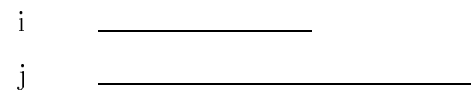
- i precedes j, which means i is before j, and there **is** a non-empty interval separating them

[1] $REFINE^{TM}$ is a trademark of Reasoning Systems Inc.

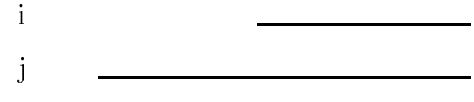
- i **overlaps** j, meaning i starts before j, and j ends after i



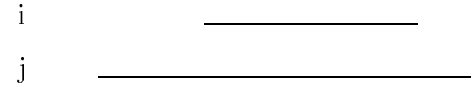
- i **starts** j, intuitively i has the same starting point as j, but is otherwise strictly contained in j (ends strictly earlier than j)



- i **ends** j, intuitively i has the same ending point as j, but is otherwise strictly contained in j (starts strictly later than j)



- i **during** j, intuitively i has a later starting point than j, and an earlier ending point



and there are six others; namely

- the converse relations to all these; except for **equals**, which is self-converse

Given a binary relation R, the converse relation $R^\smile$ is defined by the condition

$$aR^\smile b \Leftrightarrow bRa$$

All the relations above are intuitively disjoint, and they are also disjoint from their converses, i.e. no pair of intervals is in more than one of these relations. Furthermore, these relations partition the universal relation, i.e. every pair of intervals is in exactly one of these relations.

The intuition behind the pictures may be formalised in a more standard framework if we consider a linearly ordered set, whose objects we call 'points', and construct intervals

out of ordered pairs of distinct, ordered 'endpoints'. Ultimately, we wish to treat intervals as the primitive objects of which time is built up, and if there are points, they will be logical constructions out of the basic intervals (we show how this may be done for at least one axiomatic theory of intervals in chapter 4). But for our immediate purposes, we define the intervals and relations between them, over a given linear order $<$ with domain S:

The collection of *intervals over* S is

$$\{\langle a, b\rangle \ : \ a, b \in S \ \& \ a < b\}$$

We abbreviate the names of the relations, to just the first letter, for the purposes of the following definitions, except for *equality* which we denote $Id(S)$ to conform with subsquent usage:

$$Id(S) = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x = x' < y = y' \in \mathbf{S}\}$$

$$P = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < y < x' < y' \in \mathbf{S}\}$$

$$D = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x' < x < y < y' \in \mathbf{S}\}$$

$$O = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < x' < y < y' \in \mathbf{S}\}$$

$$M = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < y = x' < y' \in \mathbf{S}\}$$

$$S = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x = x' < y < y' \in \mathbf{S}\}$$

$$F = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x' < x < y = y' \in \mathbf{S}\}$$

It should be clear from these definitions that these are all the binary relations on ordered pairs of ordered distinct points of S that can be specified by giving the ordering of all four coordinates with respect to $<$ and $=$ in S.

The major thrust of Allen's work, and his joint work with Pat Hayes, was to get away from reliance on an underlying pointset over which to define intervals, and to treat intervals directly as first-class objects. Hence the qualitative pictures above serve as intuition for an axiomatic formulation and treatment of intervals, with binary relations on them.

Part of our work, which we discuss below, show that in these treatments, one can nevertheless recapture the notion of point from the models of the interval axioms, such

that intervals become ordered pairs of ordered distinct 'points', and the models have relations which satisfy the definitions given above in terms of these 'points'.

Thus, for mathematical power, it does not appear to matter whether one bases time concepts on intervals or points. However, it is a contention of *[All83, AllHay87.1, 87.2]*, and of our work presented here, that there is elegance and economy of expression in the representation of common time concepts by means of intervals directly.

We intoduce our work by giving detailed synopses of the contents of the chapters, following.

1.3 The Interval Algebra

We first present joint work of the author and Roger Maddux, which provides the foundation for much of the subsequent investigation.

Allen *[All83]* defined the interval calculus by considering pairs (a, b) of real numbers, with $a < b$, and enumerating all binary relations between such pairs which are definable using the natural ordering of the reals, specifying the ordering on all four coordinates. There are precisely thirteen such relations, including equality. Such pairs are natural models of non-trivial convex intervals of real numbers, so we simply refer to such pairs as *convex intervals*, as in *[Lad86.1, Lad86.2]*. We note first that the relations considered by Allen form a *proper Relation Algebra* in the sense of *Tarski [JonTar52, TarGiv86, Mad78]*, which we call the *Interval Algebra*. The Interval Algebra is finite and *atomic*.

We associate with an arbitrary finite simple relation algebra a certain first-order theory such that models of the theory are precisely representations of the algebra as a collection of binary relations over some universe set. In the case of the Interval Algebra, this theory has thirteen binary relation symbols, including the equality relation. We call this particular theory LM. LM has, as models, precisely the representations of the relation algebra as a collection of binary relations (sets of pairs) over some domain.

We consider the *Point Algebra*, which is the relation algebra generated by the relation of *less-than* on the rational numbers $\mathbf{Q}$. Any relation algebra generated from the order relation on a countable dense linear order without endpoints must be isomorphic to the Point Algebra.

We introduce the *Interval Algebra* which corresponds to Allen's definition of the interval relations, by defining equality and six of the other primitive relations over the domain of distinct pairs of ordered rational numbers. The remaining six primitive relations are the converses of the six non-self-converse primitives (equality is self-converse). These relations are entitled *precedes, meets, overlaps, starts, finishes* and *during*.

The Interval Algebra thus defined has exactly the same operation table as given by Allen *[All83]* for the interval structure over the real numbers (except for a misprint in the

printed table - see chapter 2). Since *converse* and *composition* are additive, this uniquely identifies the algebra and ensures that our definition over rational pairs yields the same algebra as Allen's definition over real-number pairs.

We show that the Interval Algebra has no finite representations, by embedding the Point Algebra into the Interval Algebra.

We indicate that the Interval Algebra may be generated, as an algebra, from any single one of six of the thirteen primitive elements (atoms).

From this result, using a theorem of Tarski and Givant, we can bound the complexity of the corresponding first-order definitions of the primitive relations from any of these six, in LM.

Finally, we indicate how any representation of the Interval Algebra as a collection of binary relations over some countable set must be isomorphic in a standard way to the collection of relations over the set of pairs of rational numbers.

It follows that LM is complete, countably categorical, and hence decidable. It also admits elimination of quantifiers, as we shall see in the next section *[Lad87.4]*.

1.3.1 Related Work

The definition of the thirteen primitive relations definable using the natural ordering $<$ on the rationals has been given by others, e.g. van Benthem in *[vBen83 p70]*, where he also shows that the first-order theory of the rational intervals (represented as pairs of rational numbers) is countably categorical (that is, has only one countable model, up to isomorphism). He considers the theory formulated using just two primitive relations, *precedes* and *contained-in*. Van Benthem's *contained-in* relation is the union of Allen's *starts*, *finishes* and *during*. We call van Benthem's theory VB. VB and LM are comparable by our result in chapter 2 that all primitives may be defined from the single primitive *precedes*, as we note next.

Van Benthem defines his theory as the set of all sentences of his language with *precedes* and *contained-in* which are true in the structure of intervals over $\mathbf{Q}$. (In standard terminology, this is known as the complete theory of intervals over $\mathbf{Q}$). Since *contained-in*, as the union of *starts*, *finishes*, and *during*, is thus explicitly definable from *precedes*, by the result in chapter 2, we may define the relation *contained-in* in LM by the same explicit definition, and the theorems of VB become provable in LM. Similarly, since we may define each of the primitive relations of LM explicitly in terms of *precedes*, we may introduce these relations as defined relations in VB, and the theorems of LM become provable in VB. In other words, since *precedes* is a common primitive to both theories, the sets of primitive relations of VB and LM are each explicitly definable in the other. So LM and VB are equivalent theories in the sense that they are mutually interdefinable, and they are both countably categorical with the 'same' model (modulo the different sets

of primitive relations). Since LM is countably categorical, it follows that the axioms of LM axiomatise the complete theory of intervals over $\mathbf{Q}$.

Allen and Hayes in *[AllHay85, 87.1, 87.2]* formulated a first-order theory of intervals. Their axioms are formulated using the single primitive relation *meets*. In *[AllHay85, 87.2]*, they showed that all the primitive relations could be derived from the single relation of *meets*. In chapter 4 and *[Lad87.3]*, we have characterised all the models of this theory, showing that it is a logically weaker theory than LM/VB. (This follows since *meets* is also a primitive of LM from which all the other primitives may be explicitly defined, and we can follow the same reasoning as we did before for *precedes*). Therefore, the Allen-Hayes result for *meets* is prima facie stronger than our result, since they operate in a weaker theory. They have also shown that, in their weaker theory, the corresponding result for *precedes* doesn't hold (in contrast with our result for *precedes*) *[AllHay87.2]*, and the question for *overlaps* is still open *[Hay87.1]*.

1.4 Constraint Satisfaction Procedures

We present procedures for constraint satisfaction in the first-order theory of convex intervals. We know that the theory is the complete first-order theory of the intervals over the rational numbers, i.e. the first-order theory of the interval algebra **IA**, and that this theory is countably categorical. We proceed by showing first that the theory admits elimination of quantifiers. This means that any first-order formula is equivalent in the theory to a formula without quantifiers. Furthermore, there is a ready computational procedure for producing such a quantifier-free equivalent formula. This procedure is the key ingredient of the satisfaction procedure. We then give two algorithms for checking the consistency of arbitrary first-order formulas in the theory. Finally, we present an algorithm which constructs an assignment of intervals to free variables of a consistent first-order formula of the theory, such that the assignment satisfies the formula. A consistency check is incorporated in the algorithm.

These procedures rely on the quantifier-elimination method for the theory of unbounded dense linear orders, as in *[ChaKei73]*. Briefly, we can translate a sentence in the interval theory into a sentence in the theory of unbounded dense linear order that expresses the 'same' constraint. We may now eliminate quantifiers in the formula in the theory of unbounded dense linear order, and translate the formula back into an interval formula. The translation from atomic formulae in the theory of unbounded dense linear order does not introduce any quantifiers, so the resulting interval formula is quantifier-free, and equivalent to the original formula.

The resulting quantifier-free formula has a certain restricted form, and may be checked for consistency by a variety of known techniques that operate directly on restricted quantifier-free interval formulae. However, an intuitively better method for checking consistency does not translate back from the Boolean formula that is the result of the quantifier-elimination in the ordering of the rationals, but checks this formula directly for consistency, by a simple test.

Finally, we obtain a satisfaction procedure for arbitrary first-order formulae in the interval theory, by incorporating an assignment procedure for free variables into the quantifier elimination procedure for the interval theory. Suppose ϕ is a formula in the interval theory, and we translate into the rationals and eliminate quantifiers to obtain the formula ψ about the rational order. At this stage, ψ may be checked for consistency, and ψ is consistent if and only if ϕ is consistent in the interval theory. If ψ is consistent, the procedure assigns rational numbers to the free variables of ψ, extends this assignment, and turns it into an assignment of intervals to free variables of the translation of ψ back in the interval theory, such that the assignment satisfies the translation of ψ. This translation is equivalent to the original formula ϕ in the interval theory, so this assignment also satisfies ϕ.

The methods of this chapter extend consistency-checking and constraint-satisfaction procedures to a much wider class of expressions in the interval calculus than have been considered previously.

1.5 The Convex *TUS*.

We introduce the convex fragment of the natural time unit system, the *TUS*. The *TUS* contains *basic units*, which are designators of certain common-sense convex time intervals, and operators which allow both additional convex intervals and also unions-of-convex intervals to be built up from the basic units. In this chapter, we do not consider the operator of *duration* on the convex *TUS*. *Duration* was not considered in the Interval Calculus as formulated in *[All83]* and subsequently, although it is considered ultimately necessary to include it *[Hay87]*. We consider it one of our contributions to have included *duration* in a natural way, corresponding to our common-sense view, in the *TUS*. We reserve discussion of *duration*, and the full collection of operators until chapter 7. In this chapter, we discuss only the convex part of the *TUS*, which consists of the basic time units, and the images of these under the binary operator *convexify*, which takes the convex hull of its two interval arguments.

We show, as a consequence of theorems in chapter 2 *[LadMad87.1]*, that the part of the TUS which refers to convex intervals, is a countable representation of the Interval Algebra. Since there is, up to isomorphism, only one countable representation of the Interval Algebra (or, countable model of the Interval Calculus) *[LadMad87.1]*, we can indeed conclude that the TUS is a canonical model for the Interval Calculus.

Specifically, the TUS is complete for the Interval Calculus in the following sense: if a statement in the Interval Calculus is not a theorem of the Calculus, then there is a counterexample to that statement in the TUS, and if a statement in the language of the Interval Calculus is true for all the intervals in the TUS, then it is a theorem of the Interval Calculus.

The result entails that all reasoning about the Interval Calculus may be accomplished

using intervals in the TUS, rather than intervals represented as pairs of real numbers, or even pairs of rational numbers as in *[LadMad87.1]*.

As well as being more natural than pairs of points for common-sense notions of time, the sequence representation of time intervals is also much more richly structured. Now that we know that the *TUS* representation is complete in the sense above, we can design algorithms for the time system which exploit this rich structure, which will also be general algorithms for the Interval Calculus. We have found that for many purposes it is computationally more efficient to use simple procedurally defined algorithms that use the sequence structure, rather than using more general constraint-satisfaction algorithms, especially for trivial tasks which are performed often. We can conclude from the results presented in this chapter that these algorithms may be used as part of a general constraint-satisfier for the Interval Calculus, since the TUS is complete for the Calculus.

Finally, we show how to adapt the constraint satisfaction procedure of the previous chapter to the convex *TUS*, if it is desired to do so.

The TUS, and the Interval Calculus, have been implemented at Kestrel Institute, as part of the ongoing task of implementing and using the Extended Interval Calculus. Currently, the TUS is used as the base for the time package in a prototype project management system that has been developed as part of other research at Kestrel Institute.

1.6 Models of the Allen-Hayes Axioms

In this chapter, we characterise all models of the axioms for intervals presented in *[AllHay85, AllHay87.1, AllHay87.2]*. Allen and Hayes *[AllHay85]* introduced the axioms **M1 - M5** as a first-order logical formulation of the Interval Calculus. We investigate the axioms in the slightly different form in which they are presented in *[AllHay87.1]*. We present a complete categorisation of the models of the Allen-Hayes axioms. This enables us, via the results in chapter 2 and *[LadMad87.1]*, to directly compare the strengths of the various first-order theories of intervals in chapter 2 and *[vBen83, AllHay85, LadMad87.1]*.

We consider both the full theory, axiomatised by **M1 - M5**, which we call $\mathcal{I}_{\mathcal{AH}}$, and a subtheory, axiomatised by **M1 - M3, M5**, which we call $\mathcal{I}_{SUB}$.

We characterise the models of both of these axiom sets by considering certain syntactic definitions and the consequences of these in the theory. We introduce 'points' as a definable equivalence relation on pairs of intervals (the term '*intervals*' just refers to objects in the model). (Rather than develop a theory of pairs within the axioms, we use a syntactically definable relation with four interval arguments to define the equivalence relation on pairs of intervals). We call the equivalence classes *pointclasses*.

We show that pointclasses are linearly ordered by a definable relation (which again has to be a relation on four intervals rather than on pairs of intervals), as a consequence of the axioms.

We associate to each interval two pointclasses, representing the 'ends' of the interval, and show these pointclasses are unique, for a given interval. We show that axiom **M4** guarantees also that there is a unique interval corresponding to a given pair of pointclasses.

We can now show that the ordered pairs of ordered distinct elements from an arbitrary unbounded linear order S, a structure which we call *INT(S)*, forms a model of $\mathcal{I}_{\mathcal{AH}}$, and conversely that any model of $\mathcal{I}_{\mathcal{AH}}$ is of the form *INT(S)*, for some unbounded linear order S.

When the restriction of uniqueness ensured by axiom **M4** is dropped, there may be an arbitrary number of intervals with given *endpoint-classes*, and we show that the models of $\mathcal{I}_{SUB}$ are characterised by two parameters:

- the (unbounded) linear ordering of the pointclasses
- for each pair of pointclasses, the number of different intervals with that pair as the 'endpoints'.

Finally, we show how to complete the Allen-Hayes axioms so that they have *INT(Q)*, the rational intervals, as the only countable model up to isomorphism (see also the comment in *[AllHay87.1]*).

1.6.1 Background and Comparisons

Allen and Hayes formulated their theory as a set of five axioms in first-order logic *[AllHay85]*, and as a slightly different set in *[AllHay87.1]*. We only consider the more recent axiomatisation in *[AllHay87.1]*.

They arrive at their axiomatisation by considering desirable properties of the relation *meets*, and they derive all the other relations from *meets [AllHay87.2]*. *Meets* is another primitive relation from which all the primitives in *INT(Q)* may be defined *[LadMad87.1]*.

Allen and Hayes produce their axiomatisation by considering desirable properties that the relation of *meets* should have, and they are not committed to a complete theory, especially since they want *INT(Z)* (intervals with integer endpoints) as a model as well as *INT(Q)*. They are able to show that the other relations are definable from *meets* in the sense that the full table of relational compositions (the so-called *transitivity table* in *[All83]*) is derivable from their axioms, using intuitive definitions of the other relations from *meets*. (They arrive at their definitions by considering the structure of *INT(R)*).

We may compare the Allen-Hayes theory to LM/VB, as we mentioned in a previous section, by reformulating the LM axioms in terms of the primitive relation *meets*, instead of *precedes* (they are interdefinable, since each is a relation from which all the others may be syntactically defined).

Since the Allen-Hayes theory has *INT(Q)* as a countable model (this is easy to check), it must be either weaker, or as strong as, LM/VB, since the latter has *INT(Q)* as the only countable model (up to isomorphism). Since *INT(Z)* is also a model of the Allen-Hayes theory, this must be a strictly weaker theory.

The results of this chapter show that the Allen-Hayes axioms do exactly what they were devised to do, namely axiomatise precisely the theories of intervals over an arbitrary unbounded linear ordering. We also show how to complete the Allen-Hayes axioms so that they have *INT(Q)* with *meets* as the only countable model up to isomorphism. This completion therefore gives another axiomatisation of *Th(INT(Q))*, which is qualitatively different from that obtained in chapter 2 and *[LadMad87.1]*.

1.7 The Extended Interval Model

We depart from our previous concerns with showing the logical properties of certain representations of convex interval structures, in order to consider the adequacy of purely convex intervals for specification purposes in computer science and AI.

We consider an example in the specification of concurrent processes, and use it as evidence that the convex interval model is deficient for certain representative purposes. The example is that of representing the sharing of a single CPU by multiple processes in a multiprogramming environment. We conclude that we need to use intervals which are, not convex, but unions of convex intervals.

The general union-of-convex interval looks something like this:

We do not assume any bound on the number of **maximal convex subintervals**. (A maximal convex subinterval can be thought of as a single whole line segment in the picture. We call them **maxconsubints** for short). The unions-of-convex interval calculus can, in fact, express that the interval is the union of **n** maxconsubints, for each natural number **n**, if this is needed.

On to the example. We consider multiple processes running on a single CPU, and suppose we wish to associate a predicate running(P) with a process P, to assert that P has control of the CPU. For any process P, running(P) will be true at some times and false at others. The predicate does not yet contain sufficient information about the dependence of its truth value on the time at which it is evaluated. Without knowing anything about the structure of the time domain, we may try to include in the predicate a single parameter to express the dependency. The predicate becomes running(P,d), with the extra parameter d.

What should the values of d look like? The predicate running(P,d) is intended to have a single truth value for appropriate choices of the parameter d. In particular, for some choice d_1, running(P,d_1) will be true, and for some other choice d_2, running(P,d_2) will be false. In this case, we intuitively want to pick out the 'maximal' (in some sense) d_1 such that running(P,d_1) is true over d_1, and hope that this will define the value of running(P,d) for all other d. d will be an interval, and the notion of 'maximal' that we need will be the notion of containment of intervals.

We consider two cases - a simple multiprogramming protocol, and a more complex context-switching environment, to show how this can be done. In the case of context-switching, since a process will have control of the CPU on an intermittent basis, we will find we need to associate a union-of-convex interval parameter to the process.

If we are to elevate union-of-convex intervals to first-class status, we need to extend the language of relations between intervals. It is not clear at this stage that the relations on convex intervals have any natural extension to the union-of-convex case. We show how some general operators on the relations can give union-of-convex relations from Allen's basic thirteen.

Further, we develop a taxonomy of binary relations between union-of-convex intervals, that can be used as a specification language for temporal relations of time-dependent processes, actions or predicates, for which the associated time interval does not have the simple convex form.

We make no theoretical claims for this taxonomy. We do make the claim that the collection of interval relations detailed here is adequate for a large class of examples; namely those that any of us could think of!

We obtain the following relations between unions of convex intervals:

- those generated by the functors *mostly, always, partially, sometimes*, and *disjunction* from convex relations (*always* may be defined in terms of *mostly*)
- *contains*
- *disjoint from*, which splits into:
  - *precedes* and *follows*
  - *meets* and *is met by*
  - *intermingles with*, which splits further into:
    - *disjointly-contains* and *disjointly-contained by*
    - *disjointly-overlaps* and *disjointly-overlapped by*
    - *begins preceding* and *begins following*
    - *ends preceding* and *ends following*
    - *surrounds* and *is surrounded by*
- *strictly intersects*, splitting into
  - *begins after* which splits into
    - *begins in*
    - *begins with*
    - *begins following*

  - *ends before* which splits into
    - *ends in*
    - *ends with*
    - *ends preceding*
  - *begins before* and *ends after*, with the corresponding case splits
  - *begins at* and *ends at*

- *bars*, which is a new relation not generalised from convex relations

We give the definitions of the relations, and follow with examples to show they are naturally occurring. We conclude that these relations are necessary for expressiveness in a calculus of intervals which are unions of convex intervals.

These relations are not atomic (not disjoint as sets of interval pairs), and some of them are definable from others. For example, *surrounds* is the conjunction of the relations *begins before* and *ends after*. We don't regard this as a disadvantage, since expressiveness is our main concern. However, it complicates the algebraic investigation, as does the fact that there are subrelations of the identity relation (known as *properties* to relation algebraists). The property of *convexity* is one such subrelation.

We call the system of relations on union-of-convex intervals the *Extended Interval Calculus*. It is a far more complicated creature than the convex Interval Calculus, as our preliminary investigations have shown.

Two open questions are

- precisely how many binary relations there are between two intervals which are respectively a union of n maxconsubints, and a union of m maxconsubints. We have obtained a formula, but no closed form for this expression. We do not include this in the current work.
- whether the interval relations presented here generate all the possible binary relations amongst unions-of-convex intervals with finitely many maxconsubints.

The algebraic investigation of the union-of-convex interval algebra lags far behind the work on the convex Interval Algebra. Current interest in it stems from its use as a specification language, and we pursue this aspect in the next chapter.

1.8 Application to the Synthesis of Concurrent Processes

We present an application of the Extended Interval Calculus to the specification and synthesis of concurrent processes from the specification.

The problem considered is the specification of, and synthesis of concurrent processes for, a readers/writers problem. The object of the specification is to be as general, and as

straightforward as possible. The simpler the better. Ideally, then, the synthesis procedure should detect the critical sections of the concurrent processes that will correspond to the specification, and synthesise code that will correctly handle the critical sections.

We need to make assumptions about the target language, and about a scheduler for critical sections. The assumptions we make are comparable with those made in other works on the subject, e.g. *[ManWol84]*.

We do not claim to have solved the problem of synthesising concurrent processes from specifications. However, our technique may be used to handle a wide variety of the more common situations found in concurrent contexts, namely mutual exclusion and bounded-resource problems.

1.8.1 Refinement of The Specification

The Target Language

Our target language for the refinement is very rich. It consists of a subset of *Concurrent* $REFINE^{TM}$ [2], along with certain constructs from the language of the *TUS* time system. $REFINE$ allows assertional as well as procedural specification, and allows compilation of combined specifications which include assertions about predicates as well as procedural definitions of functions, into running code. Our refinement technique was developed to illustrate how very-high-level specifications of concurrent systems could be transformed into specifications that could in principle be accomodated in an existing system. We shall need to build a sequencer for our target language that satisfies the assumptions discussed later.

The assertional part of the language has the ability to maintain conditions specified in the interval calculus. The procedural part uses *guarded commands* as synchronisation primitives.

Although the target language has the capabilites mentioned, it does not know about mutual exclusion, or indeed about any concurrency protocols. (Of course the sequencer uses some mutex primitive). The major contribution of our refinement technique is to explicate the knowledge about the concurrent protocols implied by the very-high-level specification, so that a language with
truth-maintenance capability and the ability to execute conditional statements with delay is sufficient to implement the protocol, and to do this all automatically.

Guarded Commands

The procedural part of the processes which we synthesise will employ guarded commands of the following form:

- P!foo
  Wait until predicate P is true, then execute procedure foo.

[2] *Concurrent* $REFINE^{TM}$ is a trademark of Reasoning Systems Inc.

The General Strategy

We know:

- The predicate P in a guarded command P!foo should be true precisely when it is safe to execute foo
- The procedures foo for which we need guards are the critical sections in the process

The refinement proceeds by

- using a precise formulation of the notion of *critical section*, in the interval calculus.
- synthesising precise constraints for the guard predicate P, in the interval calculus

The semantics of a guard is such that a process executing a critical-section procedure foo will first test the guard and wait for it to become true. We call this the request-foo part of critical section execution. When the guard becomes true, then foo may be executed by the process, and we call this the execute-foo stage. The exclusion constraints clearly should only apply to the execute-foo stage. We call the guard predicate for the critical section enable-foo.

When we have identified critical sections and synthesised assertions for the guard predicates, we have to assemble the synchronisation primitives into the skeleton for a process. In some scenarios, such as a message-send / confirmation-receive situation, it will be implied by the specification that the message send must occur before the receive of confirmation for that message, in order for the system to operate correctly. Hence the synchronisation primitives associated with a process will have some implied order. We comment on how to do this in the next subsection.

In our readers/writers example, there will be only one synchronisation statement in each process, so the question of identifying order will not arise.

Our technique consists in the following steps:

- Introducing all the auxiliary predicates we need, such as the -ing predicates
- For a given procedure foo, constructing the set of procedure executions with which it is mutually exclusive. If this set is non-empty, then foo will be a **critical section** and we shall need to include it as a guarded command in the skeleton for its calling process
- For all critical sections foo, we formally decompose foo into request-foo and execute-foo parts
- We construct all the conditions which assert that the guard predicate enable-foo is true when and only when it is safe to execute foo
- Finally, we order the guarded statements within the calling process as needed

Formalisation of Exclusion and Ordering

Procedures which are mutually exclusive may be specified using the relation disjoint-from in the interval calculus. If the two procedures may not execute concurrently, then the interval in which one is executing will be disjoint-from an interval in which the other is executing. Thus critical sections may be identified by constructing for each procedure the set of procedures provably disjoint from it. We call this set the disable-set of a critical section.

We have reduced the problem of identifying critical sections not explicit in the specification to the problem of deducing the domain of the particular relation of disjoint-from from a given specification in the interval-calculus.

We may order critical sections within a process by identifying, for each critical section, the set of critical sections in the process that always-(meet-or-precede) it. We may then use standard sorting algorithms (the specification lends itself naturally to the formulation of Quicksort in [*ManWald*], but any method may do) to determine the implied order in which these critical sections shall appear in the process skeleton.

Sequencing Assumptions

We make the following four assumptions:

- The knowledge required by each process of the global state of the concurrent system is at least the current truth value of all its guard predicates.
- When a process requests entry into a critical section, it may not terminate until it has entered and exited its critical section.
- When a process X requests entry into a critical section foo, if the truth of enable-foo(X) is consistent with the predicates that are currently true, the sequencer makes enable-foo(X) true.
- Arbitration of the following kind:
  - Suppose guard predicate P_1 is true, and excludes the two other guard predicates S_1 and S_2. Also suppose that $S_1 \Longrightarrow \neg S_2$, i.e. these two guards are also mutually exclusive. For example, P_1, S_1 and S_2 are mutex guards for different processes on a single resource. At most one may be true at any given time.

  We therefore require the following kind of arbitration:

  When the procedure guarded by P_1 is finished, the sequencer has to choose one of S_1 or S_2 to enable, and leave the other one false.

Arbitration generalises to multiple guards. Suppose we have a collection of guard predicates all incompatible with the current state, and of which some are incompatible with others. When the current state changes, the arbiter picks a consistent subset of them

to make true, leaving all others false. (The solution where only one is picked is sufficient, but inefficient).

Additionally, we need to assume that arbitration is accomplished in a **fair** manner, i.e. , if a process is waiting for a guard to become true, it will not be locked out for ever by the sequencer if it is at all possible for the guard to become true. This property of solutions to concurrent problems is known as **liveness**. **Liveness** properties are usually guaranteed under the assumption of a fair sequencer, as we have above, and as in [*ManWol84*].

1.9 Defining and Implementing the Full *TUS*

We consider implementing the full *TUS*, which includes operators necessary for implementing the Extended Interval Calculus, within a system which incorporates a wide-spectrum specification language, and declarative forms of specification. We assume some logic programming capability, namely the ability to compile Horn clause specifications. We are able to axiomatise the operators by means of Horn clause axioms. The high-level data structure that we assume is just some finite set theory. All this is available in the *REFINE* system in which we implemented a prototype of the *TUS*.

In addition to the pure interval concepts, we also incorporate the notion of *duration* into the *TUS*. *Duration* is given in terms of length units such as days, although we discuss it here in the abstract, namely by means of axioms that will be true no matter what length units are used.

1.9.1 The operators

We use the word *type* to indicate a domain of objects of the same sort. There is no implied type theory in the use of this terminology.

interval-of(P): *P is of type task / action / event / assertion*
returns *i* of *type interval* such that *occupies(P,i)* (see below)
Since we reason assertionally about time intervals, this gives us a way of passing between the task domain and the time domain.

dissect(i): the *set* of maximal convex subintervals of *i*. *Dissect* is somewhat like a selector for the data domain of non-convex intervals even though it returns a *set*, not an interval.

combine(S): S is a *set* of intervals
makes an interval out of *S*, rather like a *union* operator. In general, this interval will be non-convex. All the intervals in *S* are subintervals of *combine(S)*. *Combine* is the constructor for the data domain of non-convex intervals.

duration(i): *type real*, the measure of *i*

convexify(startint,endint): *type interval*, the smallest convex interval containing *startint* and *endint*. Note that it follows that *convexify* is commutative and associative.

alltime: *type interval*, the global time interval that includes all time.

Note that we have omitted from the list of primitives the function

diameter(i): *type real*, the largest distance between two subintervals of i (including the duration of the subintervals)

Diameter may be defined in terms of *duration* by the equation

- *diameter(i) = duration(convexify(i,i))*

We do not include *diameter* as a primitive, since it is definable, but it is a basic part of the constraint-expression language.

Similarly, *duration* need only be defined for *convex* intervals, since its extension to union-of-convex intervals follows from its additive property.

1.9.2 Temporal Logic in the *TUS*

We also discuss how standard linear time temporal logic may be expressed in the *TUS*. The *TUS* is extended by means of three interval constants, *past, now* and *future*. Properties of these three constants are defined by means of axioms, in Horn clause form. The standard temporal operators $\Box$ and $\Diamond$ are defined in terms of these constants. This shows that the *TUS* is at least as expressive as temporal logic.

1.9.3 Axiomatising the Time Units

Finally, we axiomatise the *TUS* basic units. We assume that the basic units are sequences, and we use the sequence operation of concatenation, denoted by $\frown$. Additionally, we use Allen's relations on convex intervals, since, as we have noted in an earlier section, all the basic units are conve intervals. The axioms arrange the sequences in a format isomorphic to that of the time units they intuitively represent, e.g.

- ([**x,1**] *starts* [**x**]) $\wedge$ ([**x,12**] *ends* [**x**])
  January is the first month and December is the last month of any arbitrary year

1.10 An Example Implementation

To show how straightforward it is to implement the *TUS* in an appropriate system, we include an example of code that is currently used in the training course for the *REFINE* system, given by *Reasoning Systems Inc.*

The sample code is written in a subset of the language of *REFINE*, and develops the convex *TUS*, to the granularity of days. It also introduces a new data type, *weighted interval*, that we are currently using to investigate the synthesis of scheduling algorithms within the *TUS* framework.

We present the example in the format of specification/solution, similarly to the way it is presented in the course.

1.11 Definitions

Throughout this work, we shall frequently be using concepts from logic and model theory. In this section and the succeeding one, we define the more common concepts and results that we use in more than one place. Some concepts are used in a local way only in certain chapters, so for ease of reading we leave those definitions in the relevant chapters, close to their use. We assume that the reader has some familiarity with the basic notions of first-order logic.

The reader familiar with model theory may prefer to skip this section. For further clarification of concepts we refer the reader to *[ChaKei73, ManWal85]*. The only non-standard concept we use is that of an *atransitive* binary relation.

1.11.1 Definitions: Theories and Axioms

A *first-order language* (which we shall abbreviate to *language*) is a set of *variable symbols, relation symbols, function symbols and constant symbols* along with the standard propositonal connective symbols, quantifier symbols, and suitable punctuation (usually parentheses). We assume that the language has a *countable* set of variables (that is, the set may be put in one-to-one correspondence with the non-negative integers); that there are a finite number of relation, function and constant symbols; and furthermore that the relation and function symbols have fixed *arity* (number of argument places); so, for example, a *binary* relation symbol will have *arity 2*, that is, it will always be followed by precisely two constant or variable symbols in any *well-formed formula*. We refer to *[ChaKei73, ManWal85]* for a formal definition of *well-formed formula*, which intuitively is a string of symbols of the language that makes sense when you read it.

We assume a standard set of infinitely many variables in the first-order languages we shall discuss, and hence we can specify a language by giving only the relation, function, and constant symbols. These symbols are called the *primitive symbols* of the language. We shall be concerned exclusively with the language of the time interval theories. This language, in the Allen-Hayes version, has a single binary relation symbol $\|$ for *meets*. In the Ladkin-Maddux theory in *[LadMad87.1]*, there are twelve binary relation symbols in addition to the equality symbol. But it is noted in the paper that all these other binary relations are explicitly definable from the single relation meets, as is known to Allen and Hayes *[AllHay87.2]*. We shall therefore use the convention that the language contains the single primitive binary relation symbol $\|$, and that the other relations are defined relations.

The standard definition of a *theory* T in first-order logic is a *set of sentences*, in some first-order language, that is *closed under deduction*; that is, every deductive consequence of a sentence in the theory is also in the theory. This definition implies that theories are always infinite sets of sentences. There are different formulations of the notion of *deductive consequence*, which we can define as anything in the transitive closure of T plus a standard set of axioms for first-order logic under the inference rules of *modus ponendo ponens* and *generalisation*. (This definition may be found in *[ChaKei73]*).

One consequence of this definition is that the deductive closure of an arbitrary set of

sentences is a theory. Note that theories may be *inconsistent*, that is, they include every sentence whatever, in the language. A theory is *consistent* if it doesn't include every sentence in the language. This definition is the same as requiring that, for no sentence P do both P and its negation $\neg P$ belong to the theory, where P is in the language of T. A *complete theory* T is a consistent theory such that, for every sentence P in the language of T, either P or its negation $\neg P$, but not both, are in the theory. This definition is equivalent to requiring that T be a *maximal consistent* set of sentences, namely that if we add any other sentence in the language, and then close under deductive consequence, we will obtain an inconsistent theory.

An *axiomatisation* of a theory T is a set of sentences S such that T is the set of deductive consequences of S.

Thus there may be many different axiomatisations of the same theory T. In fact, one of our completions of the axioms of Allen and Hayes will lead to the same theory as the canonical translation technique of *[LadMad87.1]*, although the axiomatisations look completely different.

We know that the theories are the same, because both sets of axioms have the same unique (up to isomorphism) countable model, and we can thus conclude by the completeness theorem for first-order logic (see below) that the theory is precisely the set of sentences that are true in that model.

1.11.2 Definitions: Models and Completeness

A *structure* is a set of objects, along with with relations on those objects and total functions on the set, with all arguments and values in the set, and distinguished objects called *constants*. We denote structures by giving the domain set, followed by a list of the relations, then the functions, then the constants, between angle brackets $\langle, \rangle$. So, for example, the rational numbers with the relation of *less-than* on the rationals form a structure, denoted by $\langle Q, < \rangle$. So do the rational numbers with *less-than*, the operations of *addition* and *multiplication*, and the distinguished objects *0* and *1*, denoted by $\langle Q, < , +, \times, 0, 1 \rangle$.

The *language of a structure* M is the first-order language with a relation symbol of the appropriate *arity* for each relation in M, a function symbol of the appropriate arity for each function of M, and a constant symbol for each constant of M. So, for example, the language of the structure $\langle Q, < \rangle$ would be a first-order language with a single binary relation symbol corresponding to *less-than*; and the language for the structure $\langle Q, <, +, \times, 0, 1 \rangle$ would also include two binary function symbols for *addition* and *multiplication*, and two constant symbols for *0* and *1*.

A sentence of the language of a structure S is *true in* S just in case the sentence describes a truth about S when you interpret the symbols by the actual relations, functions, and constants corresponding to the symbols. (The definition was given formally by Tarski, and is given by induction on the subformulas of a sentence. The formal definition need not concern us here, and may be found in *[ChaKei73, ManWal85]*).

A *model* of a theory T is a structure whose language is the same as that of T, such

that all of the sentences in T are true in S. The class of all structures that are models of T is denoted *Mod(T)*. (For those familiar with set theory, it should be noted that *M(T)* is a proper class, not a set, for any consistent theory T. This is a consequence of the Loewenheim-Skolem theorem, along with Cantor's theorem in set theory).

The *theory of the model* M is the set of all sentences in the language of M that are true in M. The theory of M is denoted *Th(M)*. *Th(M)* is always complete, because for every sentence P in the language of M, either P is true in M, or P is false in M, in which latter case $\neg P$ is true in M. Note that M is a model for *Th(M)*.

An *isomorphism* of models M and M' is a one-to-one onto function from M to M' such that any statement about elements of M, made in the language of M and M', is true also about their images in M', and vice versa. In other words, M and M' cannot be distinguished by any statements in their common language, even ones that refer explicitly to individual elements of the model. Formally, all that is required is that the primitive symbols of the language of M satisfy this condition. It's provable that the rest of the condition follows from this *[ChaKei73]*. The importance of the notion of isomorphism follows from the fact that since we cannot distinguish isomorphic models of a theory, by statements in the language of the theory, we may regard them as 'the same model, up to isomorphism'.

A *homomorphism* is any function that *'preserves the structure'* of M and M'. It's not required to be one-to-one or onto.

From the discussion above, we need only the following statements to define these notions:

A function $\theta : M_1 \to M_2$ is a *homomorphism* of models $\langle M_1, \|_1\rangle$ and $\langle M_2, \|_2\rangle$ if and only if $(\forall x, y \in M_1)(x\|_1 y \leftrightarrow \theta(x)\|_2\theta(y))$.

An *isomorphism* is a one-to-one, onto function that is also a homomorphism.

Two models are *isomorphic* iff there is an isomorphism between them (it is easy to show that the inverse of an isomorphism is also an isomorphism, so we need be no more specific than this).

A theory T is *categorical* in a cardinal number α iff all models of T of cardinality α are isomorphic. In particular, T is *countably categorical* or $\aleph_0$*-catgorical* iff all countable models are isomorphic. Hence a countably categorical theory has only one countable model, up to isomorphism.

1.11.3 Definitions: Properties of Binary Relations

We give definitions of properties of binary relations that we shall refer to later.

Definition: A binary relation R, which we write infix, is

- **reflexive** iff $(\forall x)(xRx)$.
- **irreflexive** iff $(\forall x)(\neg xRx)$.
- **symmetric** iff $(\forall p, q)(pRq \to (qRp))$.

- **asymmetric** iff $(\forall p, q)(pRq \rightarrow (\neg qRp))$.
- **antisymmetric** iff $(\forall p, q)(pRq \ \& \ qRp \rightarrow (q = p))$.
- **transitive** iff $(\forall p, q, r)(pRq \ \& \ qRr \rightarrow (pRr))$.
- **atransitive** iff $(\forall p, q, r)(pRq \ \& \ qRr \rightarrow (\neg pRr))$.
- **linear** iff $(\forall p, q)(p < q \ \vee \ q < p \ \vee \ q = p)$.
- an **ordering** iff it is irreflexive, asymmetric and transitive.
- an **unbounded** ordering iff it is an ordering, and $(\forall p)(\exists q)(pRq) \ \& \ (\forall p)(\exists q)(qRp)$.
- a **weak ordering** iff it is reflexive, antisymmetric and transitive.
- a **linear ordering** iff it is an ordering and linear.
- a **dense ordering** iff it is an ordering and $(\forall p)(\forall q)(pRq \ \rightarrow \ (\exists s)(pRs \ \& \ sRq))$
- an **equivalence relation** iff it is reflexive, symmetric and transitive.

There are standard ways that an ordering and a weak ordering are interrelated, in a theory with equality, and we refer the reader to the treatment in *[ManWal85]*.

1.12 Results from Model Theory

There are some results from model theory that we shall need. They are:

Theorem 1 (Completeness Theorem) *Any consistent set of sentences of a first-order language has a model.*

Theorem 2 (Countable Completeness) *Any consistent set of sentences of a countable first-order language has a countable model.*

An important consequence of the Countable Completeness Theorem is:

Theorem 3 *A theory which is $\aleph_0$-categorical is also complete.*

We shall use two theorems about unbounded linear orderings. Definitions of the terms used appear in the next section.

Theorem 4 (Categoricity of Dense Linear Ordering) *All countable models of the theory of unbounded dense linear orders are isomorphic to the rational numbers with the natural ordering, $\langle Q, < \rangle$.*

Theorem 5 (Models of Linear Ordering) *There are uncountably many non-isomorphic countable models of the theory of unbounded linear orderings.*

Chapter 2

The Interval Algebra

Conventions and Background

We give attributions to all theorems except theorems of the author and Roger D. Maddux, with whom this is joint work. A theorem that appears in many sources will be given the attribution *(standard)*. Standard references for these theorems are *[JonTar52], [ChiTar51], [Mad78], [TarGiv87]*. Lemmas are given no attribution.

We assume some familiarity with the concepts of algebra, and in particular universal algebra, as found in e.g. *[BurSan81]*. In particular, we assume the reader is familiar with the notion of a *Boolean Algebra* (*[BurSan81, Hal74, Sik69]*).

The reader should note that the notion of *relation algebra*, defined by Tarski and used in this paper, is *not* the same as the notion of *relational algebra* familiar from work on relational databases, although it is related to it (pun intended). The latter notion is more closely related to another kind of algebra introduced by Tarski, known as *cylindric algebra* (*[ImiLip84]*).

2.1 Relation Algebras: Background

A *Relation Algebra* is a Boolean algebra with some additional operators, defined below. Relation algebras have been extensively studied, e.g. *[JonTar52, ChiTar51, Mad78, TarGiv87]*. A relation algebra is intended to be an algebra of binary relations conceived of as sets of pairs. (Such relation algebras are called *representable*, and it turns out that not all relation algebras are representable).

Given binary relations $R(x,y)$, $S(x,y)$, the composition $R\,;S$ of the relations, the converse $R^\smile$ of R , and the identity relation id have the first-order definitions

$$R\,;S(x,y) \Leftrightarrow (\exists z)(R(x,z)\ \&\ S(z,y)) \tag{2.1}$$

$$R^\smile(x,y) \Leftrightarrow R(y,x) \tag{2.2}$$

$$id(x,y) \Leftrightarrow (x = y) \tag{2.3}$$

These operators on relations are taken as primitives in the algebraic formulation.

To axiomatise these notions algebraically, we start with a Boolean algebra. A *Boolean algebra* is a structure

$$\langle B, +, *, 0, 1 \rangle$$

where B is a set, and $+, *, 0, 1$ have their standard meanings, and standard axioms *[Bur-San81], [Sik69], [Hal74].*

In an arbitrary Boolean algebra, we may define a partial ordering by the definition

$$x \leq y \;\Leftrightarrow\; x * y = x$$

We refer to this ordering as the *standard* $\leq$.

To turn Boolean algebra into Relation algebra, there is an additional binary operator, (denoted by ';') $x, y \mapsto x;y$ for the composition of relations; an additional unary operator $x \mapsto x^{\smile}$ to denote the operation of forming the *converse* relation of a given relation; and a constant $1'$, intended to denote the *identity* relation. The notation is due to Tarski (*[JonTar52]*). The additional axioms required for a relation algebra are:

$$(x;y);z = x;(y;z) \tag{2.4}$$

$$(x+y);z = x;z + y;z \tag{2.5}$$

$$x;1' = x \tag{2.6}$$

$$(x^{\smile})^{\smile} = x \tag{2.7}$$

$$(x+y)^{\smile} = x^{\smile} + y^{\smile} \tag{2.8}$$

$$(x;y)^{\smile} = y^{\smile};x^{\smile} \tag{2.9}$$

$$x^{\smile};-(x;y) + -y = -y \tag{2.10}$$

The first six axioms assert that $\langle A, ;, ^{\smile}, 1' \rangle$ is an involuted monoid whose operations distribute over the Boolean addition operator, and the seventh ties these structures together in an additional way. The definition first appeared in print in *[JonTar48].*

Relation algebras thus form a *variety*, or equationally defined class of *algebraic structures.* An *algebraic structure* is a structure with function symbols and constant symbols

but no relation symbols. (It is important to note that the relations in a relation algebra are the *objects* in the algebra, and that there are operators on those objects, but no relations on them).

We shall need some concepts from Boolean Algebra and Relation Algebra which we now define.

An *atom* in a Boolean algebra is a smallest non-zero element in the algebra, with respect to the standard $\leq$ i.e. there are no other elements between it and zero in the standard $\leq$. For an example, consider the algebra of all subsets of a set. The standard $\leq$ is the containment relation $\subseteq$, and the atoms are the singleton sets. Note that every object in this algebra is the union of the singleton sets of its elements, that is the Boolean sum of the atoms below it in the standard $\leq$. This motivates the next definition.

A Boolean algebra is *atomic* if every element is the sum of the atoms below it.

The next three definitions are included because they are important in the statement of some of the theorems below. We do not give motivation for these definitions.

An algebra is *simple* if it has no non-trivial homomorphic images.

A relation algebra is *integral* if it has at least two elements and satisfies the sentence

$$(\forall x)(\forall y)(x;y = 0 \leftrightarrow (x = 0 \vee y = 0)) \tag{2.11}$$

A Boolean algebra is *complete* if it has all sums and products of arbitrary sets of elements.

Representability and Representations

It is not guaranteed that the intended models of a set of axioms are the only models that there are. In the case of Boolean algebras, *Stone's Theorem* asserts that every Boolean algebra is isomorpic to an algebra of sets, and by the *Jonsson-Tarski Theorem [JonTar52]*, this result may be extended to any variety of Boolean algebras with additional additive operators, such as relation algebras *[JonTar52]*.

However, we wish our relation algebras to be represented not just by sets, but by *sets of pairs*, as befits binary relations. Roughly, a relation algebra is *representable* if it is isomorphic to a sets-of-pairs algebra, with the domain of the pairs chosen from some universe set.

We now give the definitions that enable us to make precise the concept of representability. We define the *algebra of subrelations* of an equivalence relation, which is the algebra obtained by considering all possible sets of pairs that are subsets of the given equivalence relation. The most natural, and probably the most easily visualisable, example of such a creature is the case where the equivalence relation we start from is the universal relation (the set of all pairs) on a set V. Then the algebra is just the collection of all binary relations on the set V, with the universal relation as $\mathbf{1}$. This is known as the *algebra of relations on* V. Subalgebras of these creatures are known as *proper relation algebras*, and an arbitrary relation algebra may be represented as an algebra of binary relations just in case it can be embedded in an algebra of subrelations, equivalently that it is isomorphic to a proper relation algebra.

We use the following notation. For any set V and any binary relation E,

$$Id(V) = \{\langle x, x\rangle : x \in V\} \tag{2.12}$$

$$Fd(E) = \{x : \exists y(xEy \vee yEx)\} \tag{2.13}$$

We define the *algebra of subrelations* of an equivalence relation E as

$$Sb(E) = \langle Subrel(E), \cup, \sim, \circ, ^\smile, 1'\rangle \tag{2.14}$$

where $SubrelE$ is the family of all subrelations contained in E, while $\cup$, $\sim$, $\circ$, $^\smile$, and $1'$ are respectively union, complementation with respect to E, composition, converse, and the identity relation over the field of E, i.e. $1' = Id(Fd(E))$.

For any set V, we define the *algebra of relations on* V:

$$Re(V) = Sb(V \times V) \tag{2.15}$$

Theorem 6 (standard) *For every set V, and every equivalence relation E, $Sb(E)$ and $Re(V)$ are relation algebras*

An algebra is called a *proper* relation algebra if it is a subalgebra of $Sb(E)$ for some equivalence relation E.

An algebra is called a *proper relation algebra on the set V* if it is a subalgebra of $Re(V)$.

A relation algebra is *representable* if it is isomorphic to a proper relation algebra.

A *representation* of a relation algebra A is an isomorphism which embeds A into a proper relation algebra.

We use the notation RRA for the class of all representable relation algebras.

Theorem 7 (Tar55, Mon64) *RRA is a variety (i.e. equationally axiomatisable), but has no finite axiomatisation*

Theorem 8 (Lyn50) *There are relation algebras which are not representable, i.e. not in RRA.*

This shows that the desired analogy to *Stone's Theorem* doesn't go through for representations of relation algebras.

We are interested mainly in the representations of relation algebras, in particular of the Interval Algebra, which was defined by giving an uncountable representation over the reals *[All1]*.

Theorem 9 (standard) *Let V be any set, and $A_0, ..., A_{n-1}$ be a partition of $(V \times V)$ into n disjoint relations. This partition is the set of atoms of a proper relation algebra on V iff the converse of every relation in the partition is also in the partition, the composition of any two relations in the partition is a union of sets in the partition, and the identity relation on V is a union of sets in the partition.*

2.2 Relation Algebras and Theories

In this section, we present a theorem showing the duality between an arbitrary finite, simple relation algebra and models of a particular first-order theory, defined by the algebra. We call this theory the *associated theory* of the algebra.

Let A be a simple finite relation algebra with n atoms, say

$$At(A) = \{r_0, ..., r_{n-1}\}$$

and, for some $K \leq n$,

$$1' = r_0 + ... + r_k$$

Let FL be a first-order language with equality and binary relation symbols $R_0, ..., R_{n-1}$. Let $T(A)$ be the theory in FL consisting of sentences which express the following statements:

(1) $R_0, ..., R_{n-1}$ form a partition of the universe of pairs,

(2) Two elements are equal iff they are related by one of $R_0, ..., R_k$

(3) The converse of each R_i is some R_j (according to the structure of A),

(4) for $i, j = 0, ..., n-1$, $R_i; Rj$ is a union of certain other relations among $R_0, ..., R_{n-1}$ (according to the structure of A).

More specifically,

(1′)

$$\forall x \forall y (R_0(x,y) \vee ... \vee R_{n-1}(x,y))$$

is in $T(A)$,

and

$$\forall x \forall y (R_i(x,y) \rightarrow \neg R_j(x,y))$$

is in $T(A)$, whenever $i \neq j$;

(2′)

$$\forall x \forall y (x = y \leftrightarrow R_0(x,y) \vee ... \vee R_k(x,y))$$

is in $T(A)$;

(3′) If $r_i = r_j^{\smile}$ for any $i, j < n$, then

$$\forall x \forall y (R_i(x,y) \leftrightarrow R_j(x,y)^{\smile})$$

is in $T(A)$;

(4′) if $i, j < n$ and $r_i; r_j = r_{k(0)} + ... + r_{k(m-1)}$,

then

$$\forall x \forall y (\exists z (R_i(x,z) \& R_j(z,y)) \leftrightarrow R_{k(0)}(x,y) \vee ... \vee R_{k(m-1)}(x,y))$$

is in $T(A)$.

(Note that no sentence goes into $T(A)$ in the case that $r_i; r_j = 0$)

Theorem 10 (Maddux) *If A is a finite simple relation algebra, then A is representable iff $T(A)$ has a model. In fact, if $U = \langle U, R'_0, ..., R'_{n-1}\rangle$ is a model of $T(A)$, then the function h defined by $h(r_i) = R'_i$ for all $i < n$ has a unique extension to an isomorphism H of A into $Re(U)$. Conversely, if A is representable, then there is an isomorphism H from A into $Re(U)$ for some set U, and $\langle U, H(r_0), ..., H(r_{n-1})\rangle$ is a model of $T(A)$*

This theorem says that all models of the associated theory *T(A)* are representations of the algebra A, and that if A is representable, then *T(A)* has at least one model, which has to be a representation of A.

Definition: If A is the Interval Algebra IA, which we shall introduce below, we shall refer to $T(A)$ as LM, and the corresponding axioms above as *the axioms for LM*.

Our later result that the Interval Algebra has just one countable representation, up to isomorphism, will enable us to conclude that LM has just one countable model up to isomorphism (i.e. it is *countably categorical [ChaKei73]*).

2.3 The Point Algebra

Let $\mathbf{Q}$ be the rational numbers, and L the natural relation $<$ on $\mathbf{Q}$. L generates a proper relation algebra, the smallest subalgebra of *Re(*$\mathbf{Q}$*)* containing L. We call this algebra $\mathbf{P}$.

Lemma 1 *(1)* $\mathbf{P}$ *has three atoms, hence eight elements.*
(2) The atoms are L, $L^\smile$ and **1'**.
(3) The tables for $^\smile$ and **;** *are:*

$^\smile$	
$Id(\mathbf{Q})$	$Id(\mathbf{Q})$
L	$L^\smile$
$L^\smile$	L

;	$Id(\mathbf{Q})$	L	$L^\smile$
$Id(\mathbf{Q})$	$Id(\mathbf{Q})$	L	$L^\smile$
L	L	L	$\mathbf{1}$
$L^\smile$	$L^\smile$	$\mathbf{1}$	$L^\smile$

Suppose $\mathbf{P'}$ is any relation algebra on three atoms, with an atom l such that the atoms of $\mathbf{P'}$ are **1'**, l and $l^\smile$, and the operators $^\smile$ and **;** satisfy the tables above.

Lemma 2 *For any such algebra* $\mathbf{P'}$,

(1) **P'** *is isomorphic to* $\mathbf{P}$

(2) Both $\{\mathbf{1'} \mapsto Id(\mathbf{Q}), l \mapsto L, l^\smile \mapsto L^\smile\}$ *and* $\{\mathbf{1'} \mapsto Id(\mathbf{Q}), l \mapsto L^\smile, l^\smile \mapsto L\}$ *are isomorphisms of* **P'** *to* $\mathbf{P}$*, and these are the only isomorphisms*

(3) **P'** *is integral and simple*

We will refer to any of these isomorphic algebras as the **Point Algebra**, **PA**.

Theorem 11 *Let* **R** *be any representation of the Point Algebra* **PA**, *embedding it into* $Re(V)$ *for some* V.

(1) V *is infinite, and* $\langle V, R(l)\rangle$ *is a dense linear order without endpoints*

(2) If $\langle W, L\rangle$ *is a dense linear order without endpoints, then there is a representation* S *embedding* **PA** *into* $Re(W)$ *such that* $S(l) = L$

(3) If S *is some other representation of* **PA**, *embedding it into* $Re(W)$, *and* W *has the same cardinality as* V, *then there is an isomorphism* $I : Re(V) \longrightarrow Re(W)$ *such that* $R \circ I = S$, *and* I *is induced by a one-to-one correspondence between* V *and* W.

2.4 The Interval Algebra

The Interval Algebra was intended to have domain $\{\langle x, y\rangle : x < y, x, y \in \mathbf{R}\}$ *[All83]*. We present it instead by defining the algebra over pairs of ordered distinct *rational* numbers, i.e. over $\{\langle x, y\rangle : x < y, x, y \in \mathbf{Q}\}$. We observe that this domain is just L itself, since a binary relation is just the set of pairs of elements bearing that relation to each other. The Interval Algebra is thus presented as a proper relation algebra on L itself.

$$L = \{\langle x, y\rangle : x < y, x, y \in \mathbf{Q}\} \tag{2.16}$$

The relations are thus pairs of pairs of rationals. The following definitions are those implicitly used by Allen, but over the reals **R** rather than **Q**, and are also to be found in *[vBen83], p70*, as noted in the introduction to this paper.

$$Id(L) = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x = x' < y = y' \in \mathbf{Q}\} \tag{2.17}$$

$$P = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < y < x' < y' \in \mathbf{Q}\} \tag{2.18}$$

$$D = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x' < x < y < y' \in \mathbf{Q}\} \tag{2.19}$$

$$O = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < x' < y < y' \in \mathbf{Q}\} \tag{2.20}$$

$$M = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < y = x' < y' \in \mathbf{Q}\} \tag{2.21}$$

$$S = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x = x' < y < y' \in \mathbf{Q}\} \tag{2.22}$$

$$F = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x' < x < y = y' \in \mathbf{Q}\} \tag{2.23}$$

Allen's names for these relations are, respectively, *equals, precedes, during, overlaps, meets, starts, finishes*. These relations are disjoint, and except for $Id(L)$ are disjoint from their converses. If we include the converses, we obtain a set of thirteen relations. (*Id(L)* is self-converse).

Lemma 3 *$\{Id(L), P, P^\smile, D, D^\smile, O, O^\smile, M, M^\smile, S, S^\smile, F, F^\smile\}$ is a partition of L, hence by a standard theorem above is the set of atoms of a proper relation algebra on $L \times L$.*

We call this algebra the **Interval Algebra**, **IA**. The following lemma is easy to check:

Lemma 4 *The Interval Algebra satisfies the operation table for composition of atoms given in [All83].*

The composition table is called the *transitivity table* in *[All83]*. (Note that the entry for $d^\smile; d$ in the table in *[All83]* is incorrect). Both *composition* and *converse* are additive relations (that is, they commute with $+$) *[JonTar52, Mad78]*. This entails that, once they are defined on the atoms of an atomic algebra, they are determined on the whole algebra. Allen's relations were defined on pairs of real numbers, but if we correlate each relation with its corresponding relation over rational pairs in the Interval Algebra, we note that the operation tables on the atoms are identical. Thus the operation tables on the whole algebras must be identical, and so we can conclude that the Interval Algebra is isomorphic to the algebra defined by Allen in *[All83]* over real-number pairs. Henceforth we refer to either as the Interval Algebra. We note it is therefore possible to define a finite relation algebra by giving the atoms and the tables for *composition* and *converse* on the atoms. We refer to this as a *presentation* of the algebra.
In *[AllHAy85]* it is pointed out that all the relations in

$$\{Id(L), P, P^\smile, D, D^\smile, O, O^\smile, M, M^\smile, S, S^\smile, F, F^\smile\}$$

are first-order definable in $\langle L \times L, M\rangle$. We show that much more is true.

Theorem 12 (Maddux) *Each atom in $\{P, P^\smile, O, O^\smile, M, M^\smile\}$ generates* **IA** *without using* **1'**

Corollary 1 *The first-order definitions of the other relations in*

$$\{Id(L), P, P^\smile, D, D^\smile, O, O^\smile, M, M^\smile, S, S^\smile, F, F^\smile\}$$

from M (or the others in the theorem above) may be accomplished with formulas containing no more than three (free or bound) distinct variables and no occurrence of the equality symbol.

Theorem 13 *There is a subalgebra of* **IA** *isomorphic to* **PA**, *with the atoms*

$$\mathbf{1'},\ \mathbf{b},\ \mathbf{b}^\smile$$

where

$$\mathbf{b} = (P + D^\smile + O + M + S + F^\smile)$$

and

$$\mathbf{b}^\smile = (P^\smile + D + O^\smile + M^\smile + S^\smile + F).$$

Corollary 2 **IA** *has no finite representations*

The main theorem asserts the uniqueness up to isomorphism of countable representations of **IA**, thus assuring they are isomorphic to the pairs-of-rationals representation with which we introduced **IA**. We assume for the statement of the theorem that we are given a presentation of **IA** as the atoms $\{id, p, p^\smile, d, ...\}$ and the table for composition of the atoms. We shall identify **IA** with this presentation, in line with our earlier comments on presentations.

We show that any representation of **IA** must be isomorphic to the representation given by $\langle L, < \rangle$ and the relations $\{Id(L), P, P^\smile, ...\}$.

Let $h : V \longrightarrow W$ be one-to-one onto.

Let $H : Sb(V \times V) \longrightarrow Sb(W \times W)$ be defined by

$$H(R) = h^{-1} \circ R \circ h$$

for every R in $Sb(V \times V)$.

Lemma 5 *H is an isomorphism $Re(V) \longrightarrow Re(W)$*

We say that H is the isomorphism *induced by h.*

Let R be any representation of **IA** mapping it into $Re(V)$ for some countable (and necessarily infinite) set V. So $R : \mathbf{IA} \rightarrow Re(V)$.

Theorem 14 (Main Theorem) *There is a one-to-one correspondence*

$$h : V \longrightarrow L$$

with induced isomorphism H, such that

$$\begin{aligned} H(R(p)) &= P, \\ H(R(d)) &= D, \\ H(R(o)) &= O, \\ H(R(m)) &= M, \\ H(R(s)) &= S, \\ H(R(f)) &= F \end{aligned}$$

Recall that LM is the first-order theory associated with **IA**. The following two corollaries about LM are immediate. The second corollary follows from the first by *Vaught's Test [ChaKei73].*

Corollary 3 *All countable models of LM are isomorphic.*

Corollary 4 *LM is complete.*

Corollary 5 *LM is decidable.*

The decidability follows from the countable categoricity, and the finite axiomatisability. Van Benthem *[vBen83]* first noted that *Th(INT(Q))* is countably categorical and finitely axiomatisable, and hence he knew it was decidable. LM also admits elimination of quantifiers, as we shall see below.

2.5 Summary

We have formulated the interval calculus of *[All83]* as a Relation Algebra. This algebra is identical to the algebra formed by using the same definitions of primitive relations over pairs of distinct ordered rational numbers, called the Interval Algebra. We have noted that all countable representations of the Interval Algebra are isomorphic to the pairs-of-rationals representation. We have also noted that the entire algebra may be defined from any single one of six of the original atoms.

We have associated a first-order theory with an arbitrary finite, simple, atomic relation algebra, such that the models of the theory are precisely the representations of the algebra. Our results above thus have the consequences that the first-order theory associated with the Interval Algebra, LM, is complete and decidable, and all countable models are isomorphic to the collection of interval relations over pairs of rational numbers from which the Interval Algebra was defined. Furthermore, LM may be axiomatised from any single one of six of the primitive relation symbols, by using definitions of the other twelve relation symbols. These definitions may be given using no more than three variables, by a result of Tarski and Givant.

The complete theory LM contains the theory of *INT(Q)* in *[vBen83]*, which we have called VB, provided the extra primitive *contains* is introduced by explicit definition, and is identical with it on the common vocabulary. (and vice versa, of course, if the primitives of LM are introduced in VB by explicit definition) By results in *[Lad87.3]*, LM/VB is logically stronger than the Allen-Hayes theory *[AllHay85, 87.1]*.

Chapter 3

Constraint Satisfaction Procedures

3.1 Synopsis

In this chapter, we present procedures for quantifier-elimination, consistency checking (and therefore deciding), and providing satisfying assignments for consistent formulas in the first-order theory of **IA**, which we have called LM, and for which an axiomatisation was given in the previous chapter. LM is countably categorical, and decidability was already noted as a consequence of this and the finite axiomatisability. We comment on the procedures in order.

The quantifier elimination proceeds by producing a translation $(-^*)$ from the first-order language of the **IA** structure to the first-order language of the rational numbers under the natural ordering. The latter structure is also countably categorical. The translation says the 'same thing' in the rationals as was said by the original formula about intervals. We then apply the standard technique to eliminate quantifiers in the ordering theory of the rationals *[ChaKei73]*, which gives us a mapping $(-^\sharp)$ from formulas in the language of RAT to a subclass of the quantifier-free formulas in the language of the rational ordering. Finally, we give a translation $(-^\dagger)$ from Boolean formulas in the rationals to Boolean formulas in LM, which again says the 'same thing', giving us a quantifier-free formula in the interval theory that is equivalent in the theory to the original formula with which we started. This Boolean formula has a particular form - it is a disjunction of constraints of the kind considered by Allen *[All83]*, and therefore amenable to a parallel application of his technique, or others, for checking consistency. However, this is not the best means of doing so, since consistency of the original formula may already be checked in the rationals without translating back.

There is practical interest not just in checking constraints for consistency alone, but in trying to satisfy the constraints, i.e. find an interpretation of the free variables in the constraining formula that will render this formula true in the intended model. The elimination of quantifiers in the interval theory is an important computational procedure, since it reduces arbitrary first-order constraint satisfaction to Boolean constraint satisfaction,

for a restricted class of Boolean formulas. We may combine the quantifier-elimination with a simple assignment procedure to obtain an assignment to free variables of a consistent interval formula that satisfies the formula. We call such a procedure a *satisfaction procedure*. The satisfaction procedure utilises the translation into the rational order, finds rationals satisfying a given order relation (see below for definition), which may be done easily if and only if the order relation is consistent, and then uses these rationals as the endpoints of intervals which are assigned to the free variables of the original interval formula.

The consistency procedure is also a decision procedure, since a given formula without free variables in LM is consistent if and only if a theorem, and a given formula with free variables is consistent if and only if satisfiable.

It is a matter for further research, beyond the immediate scope of this work, to decide between the various available techniques for satisfying Boolean formulas in the interval theory. Various polynomial-time and other algorithms have been recently suggested as practical consistency algorithms, besides Allen's algorithm, and those he was aware of *[All83, Bel87, Dav87, MacFre85]*. We do not discuss the relative efficiency of practical methods for checking Allen formulas for consistency. The problem is known to be NP-hard in general *[VilKau86]*, which makes practical comparisons generally harder. Our contribution to this research is to present not just a consistency algorithm but a satisfaction algorithm for a much larger class of constraints than has been considered before in the context of interval theories.

3.2 The Translations

3.2.1 Comments and Definitions

In this section, we introduce a translation $(-^*)$ from formulae of the language of INT to formulae of the language of RAT, and another translation $(-^\dagger)$ from Boolean formulae of the language of RAT to Boolean formulae of the language of INT that preserves satisfiability, and furthermore such that objects satisfying the image of a formula ϕ under one of these mappings are easily computable from objects satisfying ϕ. First, we define some terminology.

Let INT be the structure in a language with equality with twelve binary relation primitives, introduced as the IA in the previous chapter. *Th(INT)* is therefore the first-order theory of this structure. Since LM is countably categorical with INT as its only countable model up to isomorphism, LM and *Th(INT)* must be identical. Thus *Th(INT)* may be finitely axiomatised by the axioms for LM introduced previously. We shall refer to the theory in this chapter mainly as *Th(INT)*, for similarity with the notation for the theory of the rational order which we introduce below.

Let RAT be the structure $\langle Q, < \rangle$, where Q is the rational numbers and $<$ the natural ordering. Thus *Th(RAT)* is the usual theory of unbounded dense linear order. It should be noted that *Th(RAT)* is also countably categorical, and finitely axiomatisable *[ChaKei73]*.

We assume standard notions of satisfaction and truth in a model. The satisfaction relation is normally defined in terms of an assignment of an element of the domain of the model to each variable in the language. We define an *assignment* to be a function with domain the set of variables in the language, and codomain the domain of the model. We define an *assignment of i to x* is any assignment whose value at the variable x is the object i. It is not required that an assignment be an *onto* function. The range of the function may be a strict subset of the codomain. We shall assume knowledge of standard theorems that say that satisfaction of ϕ by an assignment α in structure M is invariant under different assignments to variables that do not appear in the formula ϕ. Additionally, we suppose that in the assertion of a satisfaction relation, the languages of the theory and the model are compatible *[ChaKei73]*.

We use the notation

$$M \models \phi\{x_1 \leftarrow a_1, ..., x_n \leftarrow a_n\}$$

where the free variables of ϕ are included in the list $x_1, .., x_n$, and $a_1, ..., a_n$ are elements of the domain of M, to mean that the formula ϕ is true in structure M under all assignments that assign a_1 to x_1, ... , a_n to x_n. We shall implicitly assume that whenever we write a statement of this form, the variables $x_1, ..., x_n$ are all distinct. This notion is well-defined, since all assigments with the same values on $x_1, .., x_n$ will yield the same truth value for ϕ in M.

Finally, given an interval $i = (x, y)$ in the domain of *INT(Q)*, we define $i_L = x$ and $i_R = y$ to be the projections of i onto its left and right endpoints in Q.

We shall move freely back and forth between truth in INT and provability in LM, and similarly for RAT and the theory of dense unbounded linear order, without further comment.

3.2.2 From INT to RAT

We define a translation $-^*$ from a formula ϕ in the language of INT to a formula ϕ^* in the language of Q such that

$$INT \models \phi\{z_1 \leftarrow i_1, ..., z_n \leftarrow i_n\}$$

iff

$$RAT \models \phi^*\{x_1 \leftarrow (i_1)_L, ..., x_n \leftarrow (i_n)_L, y_1 \leftarrow (i_1)_R, ..., y_n \leftarrow (i_n)_R\}$$

where the x_i and y_i are new variables not occurring in ϕ.

For convenience, we use three infinite sequences of distinct variables $e_1, ..., e_n, ...$, $f_1, ..., f_n, ...$, $g_1, ..., g_n, ...$, We shall consider formulas in the language of INT to contain variables only from the list of e_n, and formulas in the language of RAT only to contain variables from the two other lists. The intuitive reason for the three different lists of variables is that we shall be translating assertions about intervals into assertions about

their left and right endpoints, and vice versa, and so we shall associate each interval variable e_n with corresponding variables f_n for its left endpoint, and g_n for its right endpoint. This is a useful piece of bookkeeping. We use the metavariables z, w to range over the list of e_n's, metavariables x, x' to range over the list of f_n's, and metavariables y, y' to range over the list of g_n's.

Roughly speaking, we are translating an assertion about $i_1, ..., i_n$ in INT into an assertion about the endpoints $(i_1)_L, ..., (i_n)_L, (i_1)_R, ...(i_n)_R$ in RAT. The translation is given by looking at the definition of the relations in Section 2.4 of the previous chapter. The defining formula in each of the relation definitions is a formula $\phi_R(x, y, x', y')$ in the language of RAT, for each of the thirteen primitive relations R of **IA**.

Hence two intervals i, j in $INT(Q)$ are in the relation R iff the predicate $\phi_R(x, y, x', y')$ is true for the endpoints of i and j. This is stated more formally in the following lemma:

Lemma 6

$$INT \models R(w, z)\{w \leftarrow i, z \leftarrow j\} \Leftrightarrow RAT \models \phi_R\{x \leftarrow i_L, y \leftarrow i_R, x' \leftarrow j_L, y' \leftarrow j_R\}$$

Consequently, if ϕ is an atomic formula of the form $R(z, w)$, we define ϕ^* to be the formula $\phi_R(x, y, x', y')$, where if $z = e_n$, then $x = f_n$ and $y = g_n$, and similarly if $w = e_m$, then $x' = f_m$ and $y' = g_m$ (we need not worry about variable clashes since all of the formulas $\phi_R(x, y, x', y')$ are quantifier-free).

The lemma thus ensures that if $INT \models \phi$ where $\phi = R(z, w)$ then $RAT \models \phi^*$ as defined above, and vice versa. We extend this property to Boolean combinations and quantifiers inductively. (The reader will note that we are assuming a proper definition of well-formed formulas, and are being somewhat sloppy with parentheses. We assume that these details are well understood).

- If $\phi = (\neg\psi)$ then $\phi^* = (\neg\psi^*)$.
- If $\phi = (\psi \wedge \rho)$ then $\phi^* = (\psi^* \wedge \rho^*)$.
- If $\phi = (\psi \vee \rho)$ then $\phi^* = (\psi^* \vee \rho^*)$.
- If $\phi = (\psi \rightarrow \rho)$ then $\phi^* = (\psi^* \rightarrow \rho^*)$.
- If $\phi = (\forall z)\psi$ and $z = e_n$ then $\phi^* = (\forall x \forall y)(x < y \ \rightarrow \ \psi^*)$ where $x = f_n$ and $y = g_n$
- If $\phi = \exists z\psi$ and $z = e_n$ then $\phi^* = (\exists x \exists y)(x < y \ \wedge \ \psi^*)$ where $x = f_n$ and $y = g_n$

The Boolean clauses just pass the translation through the Boolean connectives. The significant clauses are the quantifier clauses. The idea of the translation is that, if an interval satisfies ψ then its endpoints satisfy ψ^*. The choice of variables is bookkeeping.

Lemma 7 *Let ϕ be a formula in the language of INT. Then*

$$INT \models \phi\{e_1 \leftarrow i_1, ..., e_n \leftarrow i_n\}$$

iff

$$RAT \models \phi^*\{f_1 \leftarrow (i_1)_L, ..., f_n \leftarrow (i_n)_L, g_1 \leftarrow (i_1)_R, ..., g_n \leftarrow (i_n)_R\}$$

Proof: We proceed by induction on the length of a formula. Suppose that satisfaction is preserved for all formulas of length less than n and let ϕ be a formula of length n. If ϕ is formed from one or two shorter formulas by the use of a Boolean connective (i.e. if one of the first four clauses of our definition above applies) then it is trivial that ϕ is satisfiable if and only if ϕ^* is satisfiable. Now assume that $\phi = \exists z\psi$ and $z = e_n$ and furthermore

$$INT \models \psi\{e_1 \leftarrow i_1, ..., e_n \leftarrow i_n\}$$

iff

$$RAT \models \psi^*\{f_1 \leftarrow (i_1)_L, ..., f_n \leftarrow (i_n)_L, g_1 \leftarrow (i_1)_R, ..., g_n \leftarrow (i_n)_R\}$$

where we assume without loss of generality that the quantified variable e_n in ϕ is the variable with largest index in ψ, and we also assume without loss of generality that the list of variables indicated by the ellipsis contains all the variables from e_1 to e_n.

Then $\phi^* = \exists x \exists y(x < y \ \wedge \ \psi^*)$ where $x = f_n$ and $y = g_n$.

Suppose

$$INT \models \phi\{e_1 \leftarrow i_1, ..., e_{n-1} \leftarrow i_{n-1}\}$$

then there is some $i_n \in INT(Q)$ such that

$$INT \models \psi\{e_1 \leftarrow i_1, ..., e_n \leftarrow i_n\}$$

hence

$$RAT \models \psi^*\{f_1 \leftarrow (i_1)_L, ..., f_n \leftarrow (i_n)_L, g_1 \leftarrow (i_1)_R, ..., g_n \leftarrow (i_n)_R\}$$

and also

$$RAT \models (f_n < g_n)\{f_n \leftarrow (i_n)_L, g_n \leftarrow (i_n)_R\}$$

hence

$$RAT \models (f_n < g_n \ \wedge \ \psi^*)\{f_1 \leftarrow (i_1)_L, ..., f_n \leftarrow (i_n)_L, g_1 \leftarrow (i_1)_R, ..., g_n \leftarrow (i_n)_R\}$$

and hence

$$RAT \models$$
$$(\exists f_n \exists g_n)(f_n < g_n \ \wedge \ \psi^*)\{f_1 \leftarrow (i_1)_L, .., f_{n-1} \leftarrow (i_{n-1})_L, g_1 \leftarrow (i_1)_R, .., g_{n-1} \leftarrow (i_{n-1})_R\}$$

The argument vice-versa is equally straightforward.

The argument for the universal quantifier proceeds by noting that formulas involving the universal quantifier are logically equivalent to formulas involving the existential quantifier only, in the presence of the Boolean connectives. (The clause defining the translation ϕ^* in the case where ϕ is universally quantified is in fact redundant, and is included for general expositional completeness).

End of Proof.

3.2.3 From RAT to INT

First, we state a normal form theorem for formulas in the language of RAT. The theorem may be found in *[ChaKei73]*. Define an *order relation* to be a formula of the form

$$u_{i_1} < u_{i_2} < ..... < u_{i_k}$$

where the iterated $<$ is shorthand for the obvious conjunction of atomic formulas involving adjacent variables, and each u_{i_j} is either some f_m or some g_n. Say a formula is in *rational order normal form* (*RONF* for short) if it is a disjunction of order relations.

Theorem 15 (standard:) *Every formula ϕ in the language of RAT is equivalent in $Th(RAT)$ to a formula $\phi^\sharp$ in RONF; furthermore there is an algorithm for obtaining such a RONF formula $\phi^\sharp$ from an arbitrary formula ϕ in the language of RAT, and the free variables of $\phi^\sharp$ are a subset of those of ϕ, and possibly a proper subset.*

See *[ChaKei73]* for the proof, and details of the algorithm. This theorem says that $Th(RAT)$ admits elimination of quantifiers, since every formula in the language of RAT is computably equivalent to a quantifier-free formula. Furthermore, this quantifier-free formula is in RONF.

We now consider the atomic formulas in the language of RAT, and their translation into Boolean formulas of the language of INT. First we consider some equivalences between truths in RAT about the primitive relations, and truths in INT. We note that any atomic formula of the language of RAT is of the form $x < y$ or $x = y$, and therefore we shall consider correspondences for all possible assertions involving a primitive relation and rationals i_L, i_R, j_L, j_R corresponding to endpoints of intervals i, j. We shall use the notation $i(R_1 + R_2 + ... + R_p)j$ to assert that the interval i is in one of the relations R_q to j.

It is easy to check the truth of the following statements about RAT and INT:

List A:

- $i_L < j_L \;\Leftrightarrow\; i\,(P + M + O + F^\smile + D^\smile)\,j$
- $i_R < j_R \;\Leftrightarrow\; i\,(P + O + S + D)\,j$
- $i_L < j_R \;\Leftrightarrow\; i\,(P + M + O + Id(L) + S + F + D)\,j$
- $i_R < j_L \;\Leftrightarrow\; i\,P\,j$
- $i_L = j_L \;\Leftrightarrow\; i\,(S + Id(L) + S^\smile)\,j$
- $i_R = j_R \;\Leftrightarrow\; i\,(F + Id(L) + F^\smile)\,j$
- $i_L = j_R \;\Leftrightarrow\; i\,M^\smile\,j$
- $i_R = j_L \;\Leftrightarrow\; i\,M\,j$

It is easy to convert these equivalences into assertions about formulas and satisfaction relations. First, we define a translation $-^\dagger$ from atomic formulas ϕ in the language of RAT into formulas $\phi^\dagger$ in the language of INT.

We use the small roman letters, some with superscripts,

$$p, d, o, m, s, f, p^\smile, d^\smile, o^\smile, m^\smile, s^\smile, f^\smile$$

to denote the twelve primitive predicate symbols in the language of INT.

The list of corresponding formulas ϕ and ψ in the languages of RAT and INT respectively are:

List B:

- If $\phi = (f_m < f_n)$ then
  $\phi^\dagger = (p(e_m, e_n) \vee m(e_m, e_n) \vee o(e_m, e_n) \vee f^\smile(e_m, e_n) \vee d^\smile(e_m, e_n))$
- If $\phi = (g_m < g_n)$ then
  $\phi^\dagger = (p(e_m, e_n) \vee o(e_m, e_n) \vee s(e_m, e_n) \vee d(e_m, e_n))$
- If $\phi = (f_m < g_n)$ then
  $\phi^\dagger = (p(e_m, e_n) \vee m(e_m, e_n) \vee o(e_m, e_n) \vee s(e_m, e_n) \vee e_m = e_n \vee f(e_m, e_n) \vee d(e_m, e_n))$
- If $\phi = (g_m < f_n)$ then $\phi^\dagger = p(e_m, e_n)$
- If $\phi = (f_m = f_n)$ then
  $\phi^\dagger = (s(e_m, e_n) \vee e_m = e_n \vee s^\smile(e_m, e_n))$
- If $\phi = (g_m = g_n)$ then
  $\phi^\dagger = (f(e_m, e_n) \vee e_m = e_n \vee f^\smile(e_m, e_n))$
- If $\phi = (f_m = g_n)$ then
  $\phi^\dagger = m^\smile(e_m, e_n)$
- If $\phi = (g_m = f_n)$ then
  $\phi^\dagger = m(e_m, e_n)$

Now, $i_L < j_R$ is true in RAT iff $RAT \models (f_m < g_n)\{f_m \leftarrow i_L, g_n \leftarrow j_R\}$, and *mutatis mutandis* for the other atomic formulae. (Recall the convention that ensures $m \neq n$). Similarly, in $INT(Q)$, $i(R_1 + R_2 + ... + R_p)j$ iff

$$INT \models (R_1(e_m, e_n) \vee .... \vee R_p(e_m, e_n))\{e_m \leftarrow i, e_n \leftarrow j\}$$

So, for example,

$$RAT \models (g_m < g_n)\{g_m \leftarrow i_R, g_n \leftarrow j_R\}$$

$$\Leftrightarrow$$

$$i_R < j_R$$

$$\Leftrightarrow$$

$$i\ (P\ +\ O\ +\ S\ +\ D)\ j$$

$$\Leftrightarrow$$

$$INT \models (p(e_m, e_n)\ \vee\ o(e_m, e_n)\ \vee\ s(e_m, e_n)\ \vee\ d(e_m, e_n))\{e_m \leftarrow i, e_n \leftarrow j\}$$

$$\Leftrightarrow$$

$$INT \models \phi^{\dagger}\{e_m \leftarrow i, e_n \leftarrow j\}$$

and similarly for the other five types of atomic formulae in the language of RAT.

Hence, providing that the assignments of i_L, i_R, j_L, and j_R to f_n, f_n, g_m, and g_n, and i, j to e_m, e_n are made in accordance with the convention used above, we may state a general lemma about the mutual satisfiability of ϕ and $\phi^{\dagger}$, whose proof follows immediately from the preceding considerations. We call such a pair of assignments *mutually acceptable*. If α is a mutually acceptable pair of assigments, let α_{RAT} be the part of α in the language of RAT, and α_{INT} be the corresponding part in the language of INT.

Lemma 8 *For every atomic formula ϕ of the language of RAT, for every mutually acceptable pair of assignments α,*

$$RAT \models \phi\{\alpha_{RAT}\} \quad \Leftrightarrow \quad INT \models \phi^{\dagger}\{\alpha_{INT}\}$$

Now we have defined the translation $\phi^{\dagger}$ of an atomic formula ϕ of the language of RAT, we extend this translation to all of the Boolean formulae of the language of RAT by the following clauses, which complete the inductive definition:

- If $\phi = (\neg\psi)$ then $\phi^{\dagger} = (\neg\psi^{\dagger})$.
- If $\phi = (\psi \wedge \rho)$ then $\phi^{\dagger} = (\psi^{\dagger} \wedge \rho^{\dagger})$.
- If $\phi = (\psi \vee \rho)$ then $\phi^{\dagger} = (\psi^{\dagger} \vee \rho^{\dagger})$.
- If $\phi = (\psi \rightarrow \rho)$ then $\phi^{\dagger} = (\psi^{\dagger} \rightarrow \rho^{\dagger})$.

It is easy to verify the following extension of the lemma above:

Lemma 9 *For every Boolean formula ϕ of the language of RAT, for every mutually acceptable pair of assignments α,*

$$RAT \models \phi\{\alpha_{RAT}\} \quad \Leftrightarrow \quad INT \models \phi^{\dagger}\{\alpha_{INT}\}$$

We shall not need to extend the translation to quantified formulae of RAT.

We define a *uniform disjunction* of atomic formulae in the language of INT to be a disjunction of atomic formulae of the language of INT involving the same two variables e_m, e_n occurring in the same order in each subformula. Examples of uniform disjunctions are the formulae $\phi^{\dagger}$ corresponding to the atomic formulae ϕ in the language of RAT.

In the case that ϕ is an order relation, it is apparent that $\phi^{\dagger}$ will be a conjunction of uniform disjunctions, and thus if ϕ is a disjunction of order relations, $\phi^{\dagger}$ will be a disjunction of conjunctions of uniform disjunctions.

3.3 The Algorithms

3.3.1 Quantifier-Elimination in $Th(INT)$

We have defined two translations, $(-^*)$ from formulas in the language of INT to formulas in the language of RAT, and $(-^\dagger)$ from Boolean formulas in the language of RAT to Boolean formulas in the language of INT, which preserve satisfiability - in fact, which preserve satisfaction by mutually acceptable assignments.

From these two translations, along with the quantifier-elimination procedure $\psi \longmapsto (\psi^\sharp)$ from $Th(RAT)$, we may define a quantifier-elimination procedure for $Th(INT)$. The procedure first translates ϕ in the language of INT into ϕ^* in the language of RAT, finds a formula $(\phi^*)^\sharp$ which is a disjunction of order relations equivalent to ϕ^* in $Th(RAT)$, and translates this formula back via the operation $-^\dagger$ to a formula in the language of INT, which is thus equivalent to ϕ in $Th(INT)$.

The only part of this procedure on which we have not yet commented is the translation $-^\sharp$ in $Th(RAT)$. The variables in the translations, and the assignments in the satisfaction relation, have been fairly carefully controlled to ensure the preservation of satisfaction as we move back and forth from INT to RAT. We need to ensure that the translation $-^\sharp$ in $Th(RAT)$ is equally careful. As noted before in the standard lemma introducing $-^\sharp$, we may choose $-^\sharp$ so that the free variables of $\phi^\sharp$ are a subset of the free variables of its input formula ϕ. This ensures that the satisfaction relation is preserved, since

$$RAT \models \phi \leftrightarrow \phi^\sharp$$

$$\Leftrightarrow$$

$$\textit{for any } \alpha, RAT \models (\phi \leftrightarrow \phi^\sharp)\{\alpha\}$$

$$\Leftrightarrow$$

$$(RAT \models \phi\{\alpha\} \quad \Leftrightarrow \quad RAT \models \phi^\sharp\{\alpha\})$$

Hence we have the following lemma establishing the validity of the quantifier-elimination algorithm. Note that the translation $((\phi^*)^\sharp)^\dagger$ of a formula ϕ is just some other formula of the language of INT, obtained in a computable manner from ϕ. The translations $-^*$, $-^\sharp$, and $-^\dagger$ have no meaning within $Th(INT)$ itself.

Lemma 10

$$INT \models \phi \leftrightarrow ((\phi^*)^\sharp)^\dagger$$

and thus

$$Th(INT) \vdash \phi \leftrightarrow ((\phi^*)^\sharp)^\dagger$$

Proof: For any assignment α in INT, there is a mutually acceptable pair of assignments β_α such that $\beta_{\alpha,INT} = \alpha$ and

$$INT \models \phi\{\beta_{\alpha,INT}\}$$

$$\Leftrightarrow$$

$$RAT \models \phi^*\{\beta_{\alpha,RAT}\}$$

Furthermore, all such choices of β_α agree on the assignments to the free variables of ϕ and ϕ^*. Pick any such β_α.

$$RAT \models \phi^*\{\beta_{\alpha,RAT}\}$$

$$\Leftrightarrow$$

$$RAT \models (\phi^*)^\sharp\{\beta_{\alpha,RAT}\}$$

since the free variables of $(\phi^*)^\sharp$ are a subset of those in ϕ^*.

$$RAT \models (\phi^*)^\sharp\{\beta_{\alpha,RAT}\}$$

$$\Leftrightarrow$$

$$INT \models ((\phi^*)^\sharp)^\dagger\{\beta_{\alpha,INT}\}$$

by a previous lemma.

So, chaining all these equivalences together, and noting that $\beta_{\alpha,INT} = \alpha$, we have for any assignment α

$$INT \models \phi\{\alpha\}$$

$$\Leftrightarrow$$

$$INT \models ((\phi^*)^\sharp)^\dagger\{\alpha\}$$

Therefore, for any assignment α

$$INT \models (\phi \leftrightarrow ((\phi^*)^\sharp)^\dagger)\{\alpha\}$$

and therefore, since $Th(INT)$ is complete,

$$Th(INT) \vdash (\phi \leftrightarrow ((\phi^*)^\sharp)^\dagger)$$

End of Proof.

The Quantifier-Elimination Algorithm:
Given a formula ϕ in the language of INT, compute $((\phi^*)^\sharp)^\dagger$.
End of Algorithm.

The above lemma guarantees the correctness of the algorithm. Obviously it always terminates.

Lemma 11 *For any formula ϕ in the language of INT, $((\phi^*)^\sharp)^\dagger$ is a disjunction of conjunctions of uniform disjunctions.*

Proof: Directly from the comments on uniform disjunctions at the end of the previous section.
End of Proof.

3.3.2 Consistency Algorithms

Allen's algorithm *[All83]* checked conjunctions of uniform disjunctions for consistency. We call a conjunction of uniform disjunctions an *Allen formula.* Let us call an algorithm for checking consistency of Allen formulae a *complete Allen algorithm.* It has been noted there are such algorithms in the literature. Our first consistency algorithm will use a complete Allen algorithm for checking arbitrary first-order constraints in $Th(INT)$ by combining it with the quantifier-elimination procedure.

The quantifier-elimination method takes as input a formula ϕ of the language of INT, and produces an equivalent quantifier-free formula $\psi_\phi = ((\phi^*)^\sharp)^\dagger$, which is a disjunction of Allen formulae. Given that Allen formulae may be checked for consistency, a disjunction of Allen formulae may be checked for consistency by checking each of them in parallel.

Consistency Algorithm I:
Given a formula ϕ of the language of INT
1) Find $\psi_\phi (= ((\phi^*)^\sharp)^\dagger)$.
2) ψ_ϕ is a disjunction of Allen formulae. Apply a complete Allen algorithm to each disjunct of ψ_ϕ.
End of Algorithm.

We assume that the complete Allen algorithm used is chosen to be correct and to terminate. Given this, and that there are only a finite number of disjuncts, the parallel application of Allen algorithms terminates. It has already been noted that the quantifier-elimination algorithm is correct, and terminates. Hence the decision algorithm terminates and is correct modulo an appropriate choice of complete Allen algorithm.

However, this is not necessarily the easiest way to check consistency, given the apparatus we are using. Given a formula ϕ in $Th(INT)$, the formula $(\phi^*)^\sharp)$ is a disjunction of order relations. Since the translation $\phi \longmapsto (\phi^*)^\sharp)$ preserves satisfaction, we may check consistency of ϕ by checking the consistency of $(\phi^*)^\sharp)$ in $Th(RAT)$. $(\phi^*)^\sharp)$ is a disjunction of order relations. We shall say an order relation O is *in* $(\phi^*)^\sharp)$ if and only if O is one of the disjuncts of $(\phi^*)^\sharp)$. $(\phi^*)^\sharp)$ is consistent if and only if some order relation in $(\phi^*)^\sharp)$ is satisfiable. The following lemma is immediate *[ChaKei73]*:

Lemma 12 *An order relation $u_{i_1} < u_{i_2} < ..... < u_{i_k}$ is satisfiable if and only if it contains only a single occurrence of each free variable u_{i_j} $(1 \leq j \leq k)$.*

Thus, we have the following algorithm for consistency of a formula of $Th(INT)$ in that theory:

Consistency Algorithm II:
Given a formula ϕ of $Th(INT)$,

1) Compute $(\phi^*)^\sharp)$;
2) Check each order relation in $(\phi^*)^\sharp)$ for consistency (this may be accomplished in parallel);
3) Return **consistent** if one order relation is consistent, **inconsistent** if they are all inconsistent.
End of Algorithm.

Since *Algorithm II* involves the same steps as *Algorithm I*, but omits the final translation $(-^\dagger)$ back into the language of INT, and substitutes a double-occurrence check on each order relation in the reduced formula in $Th(RAT)$, it should be clear that *Algorithm II* is at least as efficient as *Algorithm I*, from the practical point of view.

The algorithms check a formula ϕ of the language of INT for logical consistency with $Th(INT)$. There is a well-known relation between provability and consistency in complete theories - a formula without free variables is consistent if and only if it is provable, and a formula with free variables is provable if and only if its negation is inconsistent, and inconsistent if and only if its negation is provable. The final possibility is that a formula with free variables is satisfiable, and so is its negation, of course under different assignments. The decision procedure for first-order formulas of the interval theory is to check a formula and its negation as appropriate for consistency. This will determine which of the three classes it falls into.

3.3.3 A Satisfaction Procedure

The techniques used in *Consistency Algorithm I* and *Consistency Algorithm II* may be used to yield more information about the formula ϕ. Specifically, a consistent order relation $u_{i_1} < u_{i_2} < ..... < u_{i_k}$ may be satisfied by any increasing sequence of rational numbers, say the integers $1, ..., k$ Let an *integer assignment* be an assignment of increasing consecutive integers, starting with 1, to each variable to some order relation in $(\phi^*)^\sharp)$. These integers will then represent assignments of integers to variables representing the endpoints of interval variables in ϕ. Thus, intervals with these endpoint assignments can be assigned to the interval variables of $((\phi^*)^\sharp)^\dagger)$, and by the mutual satisfiability property of the translation $(-^\dagger)$, the conjunction of uniform disjunctions that corresponds to the satisfied order relation will be satisfied by these intervals, so $((\phi^*)^\sharp)^\dagger)$ will be satisfied, so ϕ will be satisfied by this assignment also. We can summarise this in the following satisfaction algorithm:

The Satisfaction Algorithm
Given ϕ in the language of INT,
1) Compute $(\phi^*)^\sharp)$.
2) Make an integer assignment to the first (either in terms of order, or in terms of time) consistent order relation in $(\phi^*)^\sharp)$. If there is no such relation, return **inconsistent**. Suppose this order relation is O.
3) Suppose f_n has been assigned integer p, and g_n does not appear in O. Assign $p+1$ to g_n. Similarly, suppose g_n has been assigned integer $p+1$, and f_n does not appear in O. Assign p to f_n. Do this for all such f_n and g_n.

4) For each interval variable e_n such that f_n or g_n appear in O, suppose f_n and g_n have been assigned integers p and q respectively. Assign the interval (p, q) to e_n.
5) Assign an arbitrary interval, say $(0, 1)$, to each free variable e_m of ϕ such that neither f_m nor g_m appear in O.
End of Algorithm.

The correctness of the algorithm is straightforward. We have noted already the purpose of steps **1** and **2**. Suppose ϕ is consistent with $Th(INT)$. Steps **3** and **4** construct a pair of mutually acceptable assignments. The purpose of step **3** is to make an assignment to the other endpoint of intervals which have had only one endpoint assigned. The choice of this other endpoint is arbitrary, consistent with the requirement that $i_L < i_R$ for each interval i. Step **4** makes the assignment of the appropriate pair to an interval variable. Finally, step **5** completes the assignment to all the free variables of ϕ. Since O and thus $((\phi^*)^\sharp)^\dagger)$ is satisfied by a subassignment of the appropriate assignment of this mutually acceptable pair, it follows from our previous work that ϕ is satisfied by the assignment constructed by the satisfaction procedure.

3.4 Summary

We have introduced two translations $-^*$ from the language of the interval theory to the language of the theory of unbounded dense linear order, and $-^\dagger$ in the other direction, which preserve satisfiability of formulas. By composing these with the quantifier-elimination method from *[ChaKei73]* for the theory of unbounded dense linear order, we have obtained for each first-order formula ϕ in the interval theory a quantifier-free formula $((\phi^*)^\sharp)^\dagger$ in the same language, which is a disjunction of Allen formulas. The formula $((\phi^*)^\sharp)^\dagger$ is computable from ϕ. This procedure constitutes a quantifier-elimination method for $Th(INT)$.

We then discussed consistency algorithms for formulas in $Th(INT)$. We noted that the formula $((\phi^*)^\sharp)^\dagger$, since it is a disjunction of Allen formulae, may be checked for consistency by using complete Allen algorithms in parallel on each disjunct. We also noted that the formula ϕ could be checked for consistency by checking the $RONF$ of the translation ϕ^* in the rational ordering.

Finally, we showed how a procedure for constructing a satisfying assignment for a consistent interval formula ϕ could be designed by combining the quantifier-elimination algorithm with a simple assignment procedure for consistent order relations in $Th(RAT)$, and finding a mutually acceptable pair of assignments from this.

We have noted that the contribution of this work is to extend satisfaction procedures to a much wider class of formulae in the interval theory than had been considered by others.

Chapter 4

The Convex *TUS*

4.1 Synopsis

In this chapter, we show that the *TUS* is a canonical model for the Interval Calculus. We accomplish this by showing, as a consequence of theorems in chapter 2, that the part of the TUS which refers to convex intervals, is a countable representation of the Interval Algebra. Since there is, up to isomorphism, only one countable representation of the Interval Algebra (or, countable model of the Interval Calculus) we can indeed conclude that the TUS is a canonical model for the Interval Calculus.

The result entails that all reasoning about the Interval Calculus may be accomplished using intervals in the TUS, rather than intervals represented as pairs of real numbers, or even pairs of rational numbers as in chapter 2.

Specifically, the TUS is complete for the Interval Calculus in the following sense: if a statement in the Interval Calculus is not a theorem of the Calculus, then there is a counterexample to that statement in the TUS, and if a statement in the language of the Interval Calculus is true for all the intervals in the TUS, then it is a theorem of the Interval Calculus.

The sequence representation of time intervals is richly structured, compared with a pairs-of-points representation. Now that we know that this representation is complete in the sense above, we can design algorithms for the time system which exploit this rich structure, which will also be general algorithms for the Interval Calculus. We have found that for many purposes it is computationally more efficient to use simple procedurally defined algorithms that use the sequence structure, rather than using more general constraint-satisfaction algorithms, especially for trivial tasks which are performed often. We can now conclude from the results presented here that these algorithms may be used as part of a general constraint-satisfier for the Interval Calculus, since the TUS is complete for the Calculus.

We note that the satisfaction procedure from chapter 3 may be used also as an algorithm for the convex *TUS*, by making a straightforward translation. However, it should be emphasised that no information about the durations of intervals is considered in this satisfaction procedure, as the interval calculus itself does not consider durations. Therefore, the translation of the satisfaction procedure just assigns lengthy durations. The

incorporation of durations into this algorithm is a subject for further research. In our extension of the interval calculus later in this work, we do include durations.

The TUS, and the Interval Calculus, have been implemented at Kestrel Institute, as part of the ongoing task of implementing and using the Extended Interval Calculus. Currently, the TUS is used as the base for the time package in a prototype project management system that has been developed as part of other research at Kestrel Institute.

Organisation and Style

In section 2, we introduce the time unit system from *[Lad86.2]*, and define the language we will use for the statement of the main result.

In section 3, we state the main result.

Section 4 is a summary of the work.

4.2 The Time Unit System

The system of time units, the TUS, was introduced in *[Lad86.2]* by giving examples. The system consists of *basic time units*, which are sequences of integers of a certain sort, along with an binary operator *convexify*, which enables us to build arbitrary convex intervals from the basic units. The TUS includes other structure needed for the Extended Interval Calculus *[Lad86.2]*, which need not concern us here. We are concerned only with that part of the TUS which deals with convex intervals. We introduce this part by example here, and later give a more formal definition via the language $\mathcal{I}$ and the associated theory $\mathcal{J}$. $\mathcal{J}$ is the convex part of the TUS defined by the axioms given in *[Lad86.2]*, or, alternatively, may be obtained from $\mathcal{I}$ by using the algebraic methods used in this paper.

The Convex Part of the TUS

We represent the *basic* time intervals in the TUS by sequences of integers. Each element of the sequence represents a particular *year, month, day, hour, minute, second ....*, The first element represents the *year*, the next is the *month* in that year, the third is the *day*, etc. We illustrate the system down to seconds, hence our sequences will have lengths of up to six elements. The system is clearly extendable to smaller units such as microseconds and picoseconds, indeed needs to be extendable to arbitrarily small units of time in order to prove our main theorem. It was indicated in *[Lad86.2]* that the system is adequate for defining other kinds of intervals that are not *basic* intervals such as *WEEKS* and *CENTURIES*, providing there is some logic programming ability in the computational system within which the time unit system is programmed.

Some example real-time intervals in this system are:

- **[1986]** representing the year 1986
- **[1986,3]** representing the month of March, 1986
- **[1986,3,21]** representing the day of 21st March, 1986

- **[1986,3,21,7]** representing the hour starting at 7am on 21st March, 1986
- **[1986,3,21,7,30]** representing the minute starting at 7:30am on 21st March, 1986
- **[1986,3,21,7,30,32]** representing the 33rd second of 7:30am on 21st March, 1986 (the first second starts at 0)

In addition to the basic sequences that represent fixed units of clock time, we need to be able to make arbitrary convex time intervals. The operator *convexify* accomplishes this.

convexify(i,j) = *the smallest convex interval of time that contains both of the intervals i and j*

So, for example:

- convexify([1987],[1999]) = the interval containing all of the years from 1987 to the end of the century
- convexify([1987,1], [1987,12]) = the interval from January 1987 through December 1987, i.e. the year 1987.
  Thus convexify([1987,1], [1987,12]) = [1987].
- convexify([1987,3,2,8], [1987,3,2,11]) = the (American) working morning of Monday March 2 1986, from 8am until noon (the hour of 11am extends up until noon).

Axioms for defining *convexify* precisely will be given in chapter 7 (see also *[Lad86.2]*). For the purposes of this chapter, the intuitive definition and illustrations given above should suffice.

4.3 The Main Result

In this section, we introduce the languages $\mathcal{I}$ and $\mathcal{I}_1$, and the associated theory $\mathcal{J}$, of the TUS. We thus present the TUS as a formal theory, but of course this theory is *about* time intervals. We choose this method of presentation because we wish to later map the TUS intervals into pairs-of-rational-numbers, for the main theorem. It is easiest to do this by presenting the theory as a language of terms, with an axiomatised equality on those terms (which we indicate first intuitively, and later give a more formal presentation), and then defining the mapping to the set of pairs-of-rational-numbers, by means of the language, in a way which preserves the equality on those terms. This is a standard technique from Universal Algebra, see e.g. *[BurSan81]*.

We in fact introduce a more general family of languages, $\mathcal{G}^{\mu}$, and present the results for these more general languages.

The result for $\mathcal{J}$ is an instance of the general result.

4.3.1 The Language $\mathcal{I}$

We need to distinguish between *actual* convex time intervals, and *terms* in the language for denoting time intervals. This is because, as we saw, we can have two different terms denoting the same time interval, e.g. convexify([1987,1], [1987,12]) and [1987].

To be more precise, our language $\mathcal{I}$ for time units has constants for some finite sequences of integers, (which we denote by e.g. **[a,b,c]** for the 3-element sequence whose first element is the integer a, whose second is the integer b, and whose third is the integer c); contains the binary operator symbol **conv** (to stand for the *convexify* operator); and contains a standard set of punctuation symbols for constructing terms from these constants and the operator (which we write prefix). The *terms* of this language are obtained in the standard manner from the sequence constants and **conv**.

The precise collection of finite sequences that we want to denote are just those that denote actual parts of years, months, days, etc. The technical definition is straightforward, but detailed:

- the first item is an arbitrary (positive or negative) integer, i.e. a member of **Z**, the set of integers.
- the second item is an integer between 1 and 12, inclusive,
- the third item depends on the first and second items, and is always an integer between 1 and 31 (we have to allow for 28, 29 and 30 day months).
- similarly for the hour elements, minute elements, ...

We shall use the convention that when we want to refer to a term, we write it in boldface, and when we use a term to refer to an interval, we write it in lightface. For those familiar with the use/mention distinction, this means that we *use* in lightface, and *mention* in boldface.

So *convexify([1987,1], [1987,12])* is the same interval as *[1987]*, but **conv([1987,1], [1987,12])** is of course not the same term as **[1987]**. $\mathcal{J}$ of course contains the statement (**conv([1987,1], [1987,12])** = **[1987]**) as a theorem, since the intervals denoted by these two terms are in fact the same interval.

$\mathcal{J}$ is just the theory in the language $\mathcal{I}$ which is obtained by considering which terms in $\mathcal{I}$ refer to the same interval. In other words, $\mathcal{J}$ is the theory of the equality relation obtained from considering the equations true of the operator *convexify*. This theory $\mathcal{J}$ will be axiomatised in chapter 7 (see also *[Lad86.2]*, p358).

The sublanguage of $\mathcal{I}$ which we are most interested in is the language $\mathcal{I}_1$ which consists of all the sequence constants from $\mathcal{I}$ along with those terms of the form **conv(i, j)**, where **i** and **j** are sequence constants (i.e. there are no iterated applications of **conv** in terms in $\mathcal{I}_1$).

The importance of the sublanguage $\mathcal{I}^1$ is that every convex interval in the TUS may be represented by a term in $\mathcal{I}^1$. In other words, iterated applications of the *convexify* operator may be replaced by a single application of *convexify* to *basic* intervals. This fact is stated precisely in the following lemma:

Lemma 13 *For every term $t \in \mathcal{I}$, there is a term $t_1 \in \mathcal{I}^1$, such that $\mathcal{J}$ contains the statement $t = t_1$ as a theorem.*

4.3.2 The Sets $\mathcal{G}^\mu$ of Sequences

We define the set of sequences $\mathcal{G}^\mu$, where μ is an infinite collection of functions $f_1, f_2, ...$ and f_k is of arity k, for each k. (For simplicity, we identify the arguments to f_k with a single sequence of length k, so we shall write $f_k(\alpha)$ where α is a sequence of length k)

We define $\mathcal{G}^\mu_k$, the set of sequences of length k in $\mathcal{G}^\mu$, by induction.

- $\mathcal{G}^\mu_0$ is the empty set.
  (i.e. $\mathcal{G}^\mu$ does not contain the empty sequence).
- $\mathcal{G}^\mu_1 = \{[n] : n \in \mathbf{Z}\}$
  (i.e. the sequences of length one have integer elements).
- $\mathcal{G}^\mu_{k+1} = \{\alpha \frown [\gamma] : \alpha \in \mathcal{G}^\mu_k \ \& \ \gamma \in \mathbf{Z}_p\}$
  where $\frown$ is concatenation of sequences, $p = f_k(\alpha)$,
  and $\mathbf{Z}_p$ is the ring of *integers mod p*, i.e. the set $\{0, 1, ..., (p-1)\}$.
- $\mathcal{G}^\mu = \bigcup_k \mathcal{G}^\mu_k$
  (i.e. the union of all the $\mathcal{G}^\mu_k$).

Intuitively, $\mathcal{G}^\mu$ is a collection of sequences of the form $[a_1, a_2, ..., a_n]$, where a_1 is an integer, $[a_1, a_2, ..., a_{n-1}]$ is also a sequence in $\mathcal{G}^\mu$, and a_n can be any integer from 0 to $(p-1)$, where the value of p depends on the values $a_1, a_2, ..., a_{n-1}$.

Example: μ is a collection of constant functions, say $f_k(\alpha) = p_k$ for each k. Then $\mathcal{G}^\mu$ is the collection of sequences of the form $[a_1, a_2, ..., a_n]$, where a_1 is an integer, and a_k is an *integer mod* p_k, i.e. $0 \leq a_k < p_k$, for each $1 < k \leq n$.

Example: μ is the collection of functions defining the calendar used for the time unit system, i.e.
f_1 is the constant function 12 (there are 12 months in any year);
$f_2([a, 1]) = 31$,
$f_2([a, 2]) = 28$ if either a is not divisible by 4, or $a \in \{100, 200, 300\}$,
$f_2([a, 2]) = 29$ in these other cases,
$f_2([a, 3]) = 31$,
$f_2([a, 4]) = 30$,
etc;
f_3 is the constant function 24 (there are 24 hours in any day);
etc.

$\mathcal{G}^\mu$ in this example is almost the collection of basic sequences in the language $\mathcal{I}$. The difference is that the sequence elements in $\mathcal{I}$, after the first, start at 1 and go up to some value, (say 1 to 31 for the days in the first month) whereas those in $\mathcal{G}^\mu$ start at 0 (and

would go from 0 to 30 in the case of the first month). The two can be mapped onto each other simply by performing the translation

$$[a_1, a_2, ..., a_n] \in \mathcal{G}^\mu \leftrightarrow [a_1, (a_2 + 1), ..., (a_n + 1)] \in \mathcal{I}$$

We assume that this translation is always implicitly performed, and refer to the basic units in $\mathcal{I}$ as being an example of a $\mathcal{G}^\mu$.

4.3.3 The Mapping of $\mathcal{G}^\mu$ into Pairs of Rational Numbers

In order to obtain the main result, we map $\mathcal{G}^\mu$ into the set of pairs of rational numbers $\mathbf{Q} \times \mathbf{Q} = \{(a, b) : a < b \ \& \ a, b \in \mathbf{Q}\}$ ($\mathbf{Q}$ is the set of rational numbers) in a canonical way.

We define the mapping $\Gamma^\mu : \mathcal{G}^\mu \to (\mathbf{Q} \times \mathbf{Q})$ by induction on the length of the sequence $[a_1, a_2, ..., a_n]$ as follows:

- $[n] \mapsto (n, n + 1)$
  i.e. single-element sequences are mapped to pairs of consecutive integers

- Suppose $[a_1, a_2, ..., a_n] \mapsto (a, b)$. Let $p = f_n([a_1, a_2, ..., a_n])$. Let $\delta = (b - a) \div p$.
  Then $[a_1, a_2, ..., a_n, 0] \mapsto (a, a + \delta)$,
  $[a_1, a_2, ..., a_n, 1] \mapsto (a + \delta, a + 2\delta)$,
  .....
  $[a_1, a_2, ..., a_n, (p - 1)] \mapsto (a + (p - 1) \times \delta, b)$.
  I.e. we divide the interval (a, b) into p equal parts, and then map the sequences extending $[a_1, a_2, ..., a_n]$ by one element, into the equal parts, consecutively.

By results in chapter 2, $\mathbf{Q} \times \mathbf{Q}$ is a representation of the Interval Algebra, and hence a model of the Interval Calculus. $\mathcal{G}^\mu$ can inherit the interval relations from $\mathbf{Q} \times \mathbf{Q}$ in the following way:

Let R be one of the interval relations, which we write infix. Then we can define the relation R on sequences in $\mathcal{G}^\mu$ from its values on elements of $\mathbf{Q} \times \mathbf{Q}$ by means of the equivalence

$$(\alpha \ R \ \beta) \ in \ \mathcal{G}^\mu \Leftrightarrow (\Gamma^\mu(\alpha) \ \ R \ \ \Gamma^\mu(\beta)) \ in \ \mathbf{Q} \times \mathbf{Q}$$

We call this the *structure inherited from* $\mathbf{Q} \times \mathbf{Q}$ *via* Γ^μ

This does not turn $\mathcal{G}^\mu$ into a representation of the Interval Algebra, since, by results in chapter 2, it is necessary that for each relation R of the Interval Calculus, and each object a in the representation, there must exist an object b such that aRb. And, it is easy to observe that there is no sequence $\beta \in \mathcal{G}^\mu$ such that, under the structure inherited from $\mathbf{Q} \times \mathbf{Q}$ via Γ^μ, ($[n]$ starts β), or ($[n]$ ends β), or ($[n]$ during β), for any sequence $[n]$. Hence we need to extend the system $\mathcal{G}^\mu$ to provide such β for the sequences of length 1.

4.3.4 The Extension of $\mathcal{G}^\mu$

Let Σ^μ be the set of all terms of the form **conv(i, j)** where **i, j** $\in \mathcal{G}^\mu$. These terms are intended to represent the *convexify* operator on intervals represented as sequences. We saw that many terms can represent the same convex interval, so we need a way of picking just one term that names a given convex interval. First, we define the mapping Ψ^μ, which associates a pair of rational numbers with each **term** in Σ^μ:

Let $i \mapsto (a_1, b_1)$ and $j \mapsto (a_2, b_2)$ under Γ^μ.
Let $a = min(a_1, a_2)$ and $b = max(b_1, b_2)$.
Then

Ψ^μ: **conv(i, j)** $\mapsto (a, b)$

i.e. Ψ^μ maps the term **conv(i, j)** to the pair (a, b)

We now use a standard technique from Universal Algebra, in order to get the domain, and the mapping, that we want. This technique may be found in a standard text, e.g. *[BurSan81]*. We consider the equivalence relation $\equiv^\mu_\Psi$ defined on terms t_1 and t_2 in Σ^μ as follows:

$$t_1 \equiv^\mu_\Psi t_2 \Leftrightarrow \Psi^\mu(t_1) = \Psi^\mu(t_2)$$

As is standard, we denote the collection of equivalence classes by $(\Sigma^\mu / \equiv^\mu_\Psi)$.
We define

$$\Delta^\mu = (\Sigma^\mu / \equiv^\mu_\Psi)$$

Δ^μ is known as the *kernel* of the mapping Ψ^μ.
We can now factor the map Ψ^μ through the equivalence relation $\equiv^\mu_\Psi$, in the standard way, so it becomes a map $\Theta^\mu : \Delta^\mu \mapsto \mathbf{Q} \times \mathbf{Q}$ by defining, for each $\gamma \in \Delta^\mu$:

$$\Theta^\mu(\gamma) = (a, b) \Leftrightarrow (\exists t \in \Sigma^\mu)(\Psi^\mu(t) = (a, b) \;\&\; t \in \gamma)$$

Note that $\Theta^\mu(\gamma)$ is well-defined, since for a given γ, for all $t \in \gamma$, $\Psi^\mu(t)$ has the same value, by definition of the equivalence $\equiv^\mu_\Psi$.
We now extend Γ^μ to a mapping $\mathcal{G}^\mu \cup \Delta^\mu \to \mathbf{Q} \times \mathbf{Q}$ by setting $\Gamma^\mu(\gamma) = \Theta^\mu(\gamma)$ if $\gamma \in \Delta^\mu$. $\mathcal{G}^\mu \cup \Delta^\mu$ inherits the relation structure from $\mathbf{Q} \times \mathbf{Q}$ in the same manner as $\mathcal{G}^\mu$ did, and we don't bother to repeat the definition for this case.

The technique of factoring a mapping through its kernel, while mathematically appropriate, doesn't necessarily give a good idea of what the kernel looks like, i.e. precisely when it is that two terms name the same convex interval. We have already addressed this issue for the case of $\mathcal{J}$, by referring to the axiomatisation in *[Lad86.2]*. The axioms in fact axiomatise the inherited structure as we present it here, and can be regarded as a first-order presentation of the same structure that we are here presenting algebraically.

Another way of accomplishing the same factorisation is just to pick a canonical representative of each equivalence class, and let Δ^μ instead be just the collection of canonical members. We now extend Γ^μ by setting $\Gamma^\mu(\gamma) = \Psi^\mu(\gamma)$ if $\gamma \in \Delta^\mu$, to obtain the same effect as we did with the equivalence classes. There are many ways of picking a canonical representative of each equivalence class, and in fact this is the technique we used in the implementation of the time unit system at Kestrel.

We can now state the **Main Theorem**:

Theorem 16 *$\mathcal{G}^\mu \cup \Delta^\mu$ is a countable representation of the Interval Algebra under the structure inherited from $\mathbf{Q} \times \mathbf{Q}$ via Γ^μ.*

Sketch of Proof: By results in chapter 2, and*[JonTar52, Mad78]*, it is sufficient to show

- the two sets

  $$\Pi_1 = \{a : (\exists \alpha \in \mathcal{G}^\mu \cup \Delta^\mu)(\exists b \in \mathbf{Q})(\Gamma^\mu(\alpha) = (a, b))\}$$

  and

  $$\Pi_2 = \{b : (\exists \alpha \in \mathcal{G}^\mu \cup \Delta^\mu)(\exists a \in \mathbf{Q})(\Gamma^\mu(\alpha) = (a, b))\}$$

  are both countable, dense, linear orderings under the standard order, and are thus each isomorphic to the whole of $\mathbf{Q}$;

- for each relation R of the Interval Calculus, and each $\alpha \in \mathcal{G}^\mu \cup \Delta^\mu$, there exists $\beta \in \mathcal{G}^\mu \cup \Delta^\mu$ such that aRb, under the structure inherited from $\mathbf{Q} \times \mathbf{Q}$ via Γ^μ.

It is tedious but straightforward to verify these properties.

End of Proof Sketch.

From now on, we shall use the notation $\mathcal{G}^\mu \cup \Delta^\mu$ ambiguously for both the set, and the representation of the Interval Calculus obtained by inheriting the structure from $\mathbf{Q} \times \mathbf{Q}$ via Γ^μ. There should be no confusion engendered by this systematic ambiguity.

Corollary 6 *$\mathcal{G}^\mu \cup \Delta^\mu$ is isomorphic to the pairs-of-rational-numbers representation of the Interval Calculus*

Proof: By results of chapter 2, all countable representations of the Interval Algebra are isomorphic.
End of Proof.

Corollary 7 *$\mathcal{J}$ is isomorphic to the pairs-of-rational-numbers representation of the Interval Calculus*

Proof: We use a systematic abuse of language in the statement of the Corollary. $\mathcal{J}$ is technically a theory, but may be considered as a structure by taking the equivalence classes of terms under the equality relation axiomatised by $\mathcal{J}$, and the inherited structure from the mapping into $\mathbf{Q} \times \mathbf{Q}$. We have noted in an earlier example that $\mathcal{I}_1$ is an instance of $\mathcal{G}^\mu \cup \Delta^\mu$, providing that the implicit translation is made for the sequence elements of $\mathcal{I}_1$. Hence the corollary follows. **End of Proof.**

4.3.5 Adaptation of the Satisfaction Procedure

Given an assignment of intervals to variables of a consistent formula in $Th(INT)$, an assignment of intervals in the *TUS* may be made straightforwardly. Recall that the assignment resulting from the satisfaction procedure included only intervals with integer endpoints.

Suppose ϕ is a consistent formula, with free variables e_{n_i} for $1 \leq i \leq m$. Suppose (p_i, q_i) is assigned to variable e_{n_i} by the satisfaction procedure for $1 \leq i \leq m$. Then the assignment **convexify**$([p_i], [q_i - 1])$ to e_{n_i} , $1 \leq i \leq m$, of objects of the convex *TUS* to the free variables of ϕ will satisfy the formula by objects of the convex *TUS*.

4.4 Summary

We have described the convex part of a time unit system, TUS, which has been implemented and used at Kestrel Institute. This system implements the Interval Calculus introduced by James Allen *[All83]*. We have presented a theorem which has as a consequence that the (convex part of the) TUS is a canonical model for the Interval Calculus, in the sense that if a statement in the Interval Calculus is not a theorem, then there is a counterexample to that statement in the TUS, and any statement true of all of the intervals in the convex part of the TUS is a theorem of the Interval Calculus. This theorem relies heavily on results of chapter 2 *[LadMad87.1]*. We noted that the theorem allows us to devise general algorithms for the Interval Calculus that nevertheless can exploit the specific structure of the convex part of the TUS, which is much richer than that of the previously-known representations of the Interval Calculus as pairs of real numbers *[All83]*, or pairs of rational numbers (chapter 2).

We showed how to adapt the general satisfaction procedure of chapter 3 to the *TUS*, if it is desired to do so.

Chapter 5

Models of the Allen-Hayes Axioms

5.1 Synopsis

Allen and Hayes *[AllHay85]* introduced the axioms **M1 - M5** as a first-order logical formulation of the Interval Calculus. We investigate the axioms in the slightly different form in which they are presented in *[AllHay87.1]*. We present a complete categorisation of the models of the Allen-Hayes axioms. This enables us, via results in *[LadMad87.1]*, to directly compare the strengths of the various first-order theories of intervals in *[vBen83, AllHay85, LadMad87.1]*. In this section, we state the technical results.

We consider both the full theory axiomatised by **M1 - M5**, which we call $\mathcal{I}_{\mathcal{AH}}$, and also the weaker theory obtained by omitting **M4**, which we call $\mathcal{I}_{SUB}$.

We characterise the models of both of these axiom sets by considering certain syntactic definitions and the consequences of these in the theory. We introduce 'points' as a definable equivalence relation on pairs of intervals (the term '*intervals*' just refers to objects in the model). (Rather than develop a theory of pairs within the axioms, we use a syntactically definable relation with four interval arguments to define the equivalence relation on pairs of intervals). We call the equivalence classes *pointclasses*.

We show that pointclasses are linearly ordered by a definable relation (which again has to be a relation on four intervals rather than on pairs of intervals), as a consequence of the axioms.

We associate to each interval two pointclasses, representing the 'ends' of the interval, and show these pointclasses are unique, for a given interval. We show that axiom **M4** guarantees also that there is a unique interval corresponding to a given pair of pointclasses.

We can now show that the pairs of (ordered) distinct elements from an arbitrary unbounded linear order S, a structure which we call *INT(S)*, forms a model of $\mathcal{I}_{\mathcal{AH}}$, and conversely that any model of $\mathcal{I}_{\mathcal{AH}}$ is of the form *INT(S)*, for some unbounded linear order S.

When the restriction of uniqueness ensured by axiom **M4** is dropped, there may be an arbitrary number of intervals with given *endpoint-classes*, and we show that the models of $\mathcal{I}_{SUB}$ are characterised by two parameters:

- the (unbounded) linear ordering of the pointclasses
- for each pair of pointclasses, the number of different intervals with that pair as the 'endpoints'.

Finally, we show how to complete the Allen-Hayes axioms so that they have *INT(Q)*, the rational intervals, as the only countable model up to isomorphism (see also the comment in *[AllHay87.1]*).

5.2 Summary of Related Work

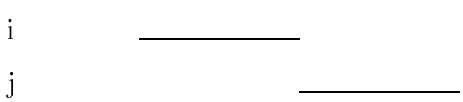
In the introduction, we compared the three first-order theories of time intervals VB of van Benthem, LM of Ladkin and Maddux, and $\mathcal{I}_{\mathcal{AH}}$ of Allen and Hayes. LM and VB are both axiomatisations of *Th(INT(Q)*, with different but intersecting sets of primitive relations, each set explicitly definable in the other. Both these theories are complete, countably categorical, and hence decidable. The Allen-Hayes theory $\mathcal{I}_{\mathcal{AH}}$ is a logically weaker theory of intervals, which has as models precisely the structures of intervals over an arbitrary unbounded linear order.

5.3 Technical Results

5.3.1 The Allen-Hayes Axioms

We consider the Allen-Hayes axioms defined in *[AllHay87.1]*. The axioms are motivated by considering intuitive properties of the relation *meets* over intervals from a linear order such as Q or Z.

The intuitive definition of *meets* is given by the picture below:



We give the formal definition in terms of intervals as pairs-of-points over some arbitrary linearly-ordered domain S.

- *(a,b)* is an *interval* if and only if $a < b$
- *(a,b) meets (c,d)* if and only if $b = c$
- *INT(S)* is the set of intervals on S, with the thirteen natural binary relations definable from the ordering on S

Note in particular that there is no question of intervals being *sets* of points, and therefore no issue as to whether they include endpoints or not. Intervals are just pairs of points, and an endpoint is just one of these points. It does turn out that the class of open, closed, and half-open (at either end) intervals on the rationals is also countably

categorical, and we can provide an extension of the Allen-Hayes axioms that have this structure as the only countable model, up to isomorphism *[Lad87.3]*.

The Allen-Hayes axioms are intended to axiomatise the concept of interval as defined above. We shall show that they have succeeded, in that all the models of their theory are isomorphic to *INT(S)* over some linearly ordered set S. Allen and Hayes require the objects referred to by the theory to be intervals, and there is no straightforward way to refer to points out of which intervals are defined. We shall show later how to define pointhood in a model.

We give the Allen-Hayes axioms without much commentary, and refer the interested reader to *[AllHay85, AllHay87.1, AllHay87.2]* for further motivation.

We use the symbol $\|$ for *meets*. Their axioms are:

M1 : $(\forall p,q,r,s)((p\|q)\ \&\ (p\|s)\ \&\ (r\|q) \rightarrow (r\|s))$

which is intended to make the 'meeting-places' unique

M2 : $(\forall p,q,r,s)((p\|q)\ \&\ (r\|s) \rightarrow (p\|s\ \otimes\ (\exists t)(p\|t\|s)\ \otimes\ (\exists t)(r\|t\|q)))$

where $\otimes$ is *exclusive or*, i.e. precisely one of the alternatives must hold.

This axiom is intended to linearly-order the meeting places

M3 : $(\forall p)(\exists q,r)(q\|p\|r)$

which is intended to ensure that the intervals are unbounded at either end of the time line

M4 : $(\forall p,q,r,s)(p\|q\|s\ \&\ p\|r\|s \rightarrow\ q=r)$

which is to ensure that there are unique intervals with particular given 'endpoints'

M5: Functional Form : $(\forall p,q)(p\|q\ \rightarrow (\exists r,s)(r\|p\|q\|s\ \ \&\ \ r\|(p+q)\|s))$

which is intended to guarantee the existence of a 'union' interval of two meeting intervals.

M5: Existential Form : $(\forall p,q)(p\|q\ \rightarrow (\exists r,s,t)(r\|p\|q\|s\ \ \&\ \ r\|t\|s))$

5.3.2 Existential M5 versus Functional M5

Allen and Hayes consider the axiom **M5** in the functional form. We prefer to include it in the existential form, and here we indicate our reasons.

The operator $+$ in **Functional M5** may be introduced by *Skolemisation* in any given model of the axioms with **Existential M5**, i.e. such a model may be augmented with the addition of a function so that it becomes a model of **Functional M5** *[ChaKei73]*. For our purposes, it is preferable to use **Existential M5**, since **Functional M5** implies nothing about the values of the operator $+$ on arguments that do not *meet*. A model of **Functional M5** will have to assign some definite value of the operator to every pair of arguments, whether they meet or not, and thus there will be a profusion of models

of **Functional M5** that differ only in their assigment of values to non-meeting interval arguments, while having the same values on meeting arguments. So in general there will be many non-isomorphic models of **Functional M5**, which are nevertheless identical on the structure that we care about (namely pairs of meeting intervals). Allen and Hayes needed $+$ to be only a partial operator, but in standard model theory, all operators must be total.

Existential M5 asserts the existence of a 'sum' interval for meeting pairs only. Hence we have no problem with the proliferation of non-isomorphic models of a partial operator, stemming from different totalisations of that operator. Since, as we have pointed out, models with $+$ may be obtained from any model of **Existential M5**, we consider that the axiomatisation with **Existential M5** is preferable, and henceforth we shall refer to this axiom simply as **M5**.

5.3.3 The Axiom M4

There are model theoretic techniques for obtaining models of $\mathcal{I}_{\mathcal{AH}}$ from models of $\mathcal{I}_{SUB}$ alone. The relation of *'having the same endpointclasses as'* is an equivalence relation that preserves the primitive relation *meets*, and therefore any model of $\mathcal{I}_{SUB}$ has a homomorphic image that is a model of $\mathcal{I}_{\mathcal{AH}}$, obtained by 'factoring through' the equivalence relation, i.e. by identifying objects iff they have the same equivalence class *[ChaKei73]*.

The models of $\mathcal{I}_{SUB}$ are not the intended models of the interval theory, however, since in general they may have many intervals with the same endpoints. Hence even though **M4** is dispensable from the point of view of model theory, we need it to pick out the intended models of the interval theory. This argues for its retention in the axiom system.

5.3.4 Analysing the Allen-Hayes Axioms

We present the definitions of 'points' in a model of the Allen-Hayes axioms, and analyse the models of the axioms.

The following theorem is due to Allen and Hayes:

Theorem 17 (Properties of $\|$) *The relation $\|$ is irreflexive, asymmetric and atransitive.*

The next theorem shows that the function notation introduced for the 'union' in **M5** is dispensable in the model theory.

Theorem 18 (Skolemisation) *: Every model of the axiom* **M5** *in the existential form may be extended (by adding a function) to a model of the axiom* **M5** *in the operator form.*

Proof and Comments: This is just the theorem of *Function Introduction* in *[Man-Wal85]*. This technique is known as *Skolemisation* to model theorists *[ChaKei73]*, as it was the form in which Skolem introduced his result. The more common use of the term *Skolemisation* in computer science is to refer to the dual technique introduced by Herbrand and used in theorem proving, which is where *universal* quantifiers are eliminated

in favor of function symbols.
End of Comments.

We now define the four-argument predicate that will generate the equivalence relation on pairs of meeting intervals.

Define *Equiv(p, q, r, s)* if and only if $p\|q$ & $r\|s$ & $p\|s$.

We shall use the notation *[p, q]* to refer to the pair of intervals p and q, and simultaneously implicitly assert that $p\|q$.

We shall write *Equiv(p, q, r, s)* as *[p, q]* $\sim$ *[r,s]*.

Using our notation, we could define *Equiv(p, q, r, s)* by the biconditional:
[p, q] $\sim$ *[r,s]* if and only if $p\|s$.

It should be understood that the notation [p, q] is only a convenience, and that assertions involving terms of this form, and $\sim$, technically should be rewritten using the 4-ary relation *Equiv* instead.

Theorem 19 ($\sim$ is an Equivalence Relation) *(a) [p, q]* $\sim$ *[p, q]*
(b) [p, q] $\sim$ *[r, s]* $\Leftrightarrow$ *[r, s]* $\sim$ *[p, q]*
(c) [p, q] $\sim$ *[r, s]* $\sim$ *[u,v]* $\Rightarrow$ *[p, q]* $\sim$ *[u, v]*

Clearly, this theorem can be rephrased entirely in terms of *Equiv* if need be. We call the equivalence classes *pointclasses*, and we denote the equivalence class of [p, q] by *[[p, q]]*. They will represent the 'points' in any model of $\mathcal{I}_{\mathcal{AH}}$.

Define the 4-ary relation *PointLess(p, q, r, s)* as follows:

PointLess(p, q, r, s) if and only if

$$(\exists u, v, w)([p,q] \sim [u,v] \;\&\; [r,s] \sim [v,w])$$

Remember this definition is short for a longer one involving assertions $p\|q$, etc.
PointLess is heterological; that is, it's not a pointless relation.
We denote *PointLess(p, q, r, s)* by the rather more perspicuous notation
[[p, q]] $\prec$ *[[r, s]]*.

Theorem 20 ($\prec$ is linear) $\prec$ *linearly orders the equivalence classes of* $\sim$

Sketch of Proof: This is straightforwardly deducible using **M1** and **M2**, remembering that the definitions of the properties have to be modified to reflect that we are ordering, not the pairs, but the equivalence classes under $\sim$, e.g. the *linearity* condition becomes

$$[[p,q]] < [[r,s]] \otimes [[r,s]] < [[p,q]] \otimes [r,s] \sim [p,q]$$

End of Sketch.

Theorem 21 (Models of the Axioms) *Given an arbitrary unbounded linear order $<$ on a set S, the intervals of S, INT(S), form a model of $\mathcal{I}_{\mathcal{AH}}$ under the definition of $\|$ given earlier. Furthermore, the ordering $\prec$ on equivalence classes of meeting intervals is isomorphic to the ordering $<$ on S.*

Sketch of Proof: If two intervals meet, they have a member of S in common. It's easy to check that the equivalence classes have the same member of S associated with each pair in the class, and that each member of S is associated with an equivalence class. To construct the required isomorphism, map [(a, b), (b, c)] to b. It is easy to see that $\prec$ on the classes is preserved as $<$ on S.
End of Sketch.

We shall show that these are the only models of $\mathcal{I}_{\mathcal{AH}}$. We accomplish this by characterising the models of $\mathcal{I}_{SUB}$, which we do in such a way that the models of **M4** are homomorphic images of these.

Theorem 22 (Endpointclasses) *For any p, there are unique equivalence classes P_1 and P_2 such that $(\exists q)([p,q] \in P_1)$ & $(\exists r)([r,p] \in P_2)$*

Proof : M3 ensures that there are such q and r, and hence that there exist such equivalence classes. Uniqueness follows from the observation that [p, q] $\sim$ [p, q'] and [r, p] $\sim$ [r', p] for any p, q, q', r, r'
End of Proof

Summarising what we have so far:

Associated with any object p in a model for $\mathcal{I}_{SUB}$ is a unique pair of equivalence classes. All intervals which meet p are included in some pair in one equivalence class, as are all intervals which are-met-by one of those. In the other are included in some pair all intervals which are-met-by p, and all intervals which meet one of those. The equivalence classes are linearly ordered.

Given any model **M** of $\mathcal{I}_{SUB}$, form the set *M'* of pairs of equivalence classes of meeting intervals under $\sim$, and using the linear order $\prec$, form the *intervals*, and the *meets* relation on these by using the standard definition for pairs from a linearly ordered set. Call the resulting model *INT(M)*, the *interval structure* of *M*.

We can now state and prove our main result categorising the models of $\mathcal{I}_{\mathcal{AH}}$. All of them are isomorphic to their interval structures.

Theorem 23 (Model Structure I) *INT(M) is a homomorphic image of M, and is a model of $\mathcal{I}_{\mathcal{AH}}$. Furthermore, if M is a model of $\mathcal{I}_{\mathcal{AH}}$, they are isomorphic.*

Sketch of Proof: The mapping is $p \mapsto ([[q,p]],[[p,r]])$ for any q, r that meet, respectively, are-met-by p. It's easy to check that the relation $\|$ is preserved by this mapping, and that the mapping is onto. Since this is the only primitive in the theory, this suffices for the homomorphism. To show isomorphism if **M4** is true in M, note that if

$p, p' \mapsto ([[q,p]],[[p,r]])$, then $q \| p'$ and $p' \| r$ and hence $p = p'$, so the map is one-to-one.
End of Sketch.

Since the interval structures *INT(M)* are homomorphic images of each model M of $\mathcal{I}_{SUB}$, it follows that to discover the structure of models of $\mathcal{I}_{SUB}$, it suffices to look at the kernel of the homomorphism, which in each case is the equivalence relation

$p \simeq q$ if and only if

$$(\exists r, r', s, s')(([[r,p]],[[p,r']]) = ([[s,q]],[[q,s']]))$$

This is the equivalence relation of *'having-the-same-endpoints-as'*, and it's easy to check that the same intervals meet p as meet q, and the same intervals are-met-by p as are-met-by q, when p $\simeq$ q. Hence the number of intervals in each $\simeq$ equivalence class may be chosen independently for each equivalence class. This may be stated more precisely in the following way:

Let *endpoints(p)* be the pair ([[r, p]], [[p, r']]).
Let *MULTI-INT(M)* consist of the pairs *(endpoints(p), p)*, with the relation of $\|$ defined as

(endpoints(p), p) $\|$ *(endpoints(q), q)* if and only if $p \| q$.

It's easy to check that $p \| q$ if and only if *endpoints(p)* $\|$ *endpoints(q)*.

Theorem 24 (Model Structure II) *MULTI-INT(M) is isomorphic to M.*

Proof: The isomorphism is $p \mapsto$ *(endpoints(p), p)*. It's clearly one-to-one since endpoints(p) is unique, given p, it's clearly onto by construction, and the $\|$ relation is clearly preserved by the definition of the relation for *MULTI-INT(M)*.
End of Proof.

Another way of constructing *MULTI-INT(M)* is simply by taking *INT(M)* and, for each $(a,b) \in$ *INT(M)*, adding an element *((a, b), p)* for each p such that *(a, b) = endpoints(p)*. This is summarised in the following theorem.

Theorem 25 (Model Structure III) *The models of $\mathcal{I}_{SUB}$ are completely characterised by*
(a) the linear ordering $\prec$ on the equivalence classes under $\sim$;
(b) the number of elements in each equivalence class of $\simeq$.

Proof: Given a model of the form *MULTI-INT(M)*, construct the model *M'* with the elements $((a,b),\beta)$ for each $\beta < \alpha$, where α is the cardinality (number) of the p such that *endpoints(p) = (a, b)*. Define $\|$ on this model the same way as in *MULTI-INT(M)*. The models are clearly isomorphic by simply putting together the one-to-one functions which map the p such that *endpoints(p) = (a, b)* onto the cardinals α. Conversely, given a structure with domain $\bigcup_{(a,b)\in INT(S)}\{((a,b),\beta) : \beta < \alpha_{(a,b)}\}$, where $\alpha_{(a,b)}$ is an arbitrary cardinal for each $(a,b) \in INT(S)$, with the definition of $\|$ as above, it is straightforward to check that this structure is a model of $\mathcal{I}_{SUB}$.
End of Proof.

Hence we have completely characterised the models of $\mathcal{I}_{SUB}$, and the models of $\mathcal{I}_{\mathcal{AH}}$. By the *Models of Linear Ordering* theorem, we have

Corollary 8 (Number of Countable Interval Models) *There are uncountably many countable models of the axioms $\mathcal{I}_{\mathcal{AH}}$.*

5.3.5 Extending the Theory

We show how to complete the Allen-Hayes axioms so as to obtain a complete and countably categorical theory whose countable models are all isomorphic to *INT(Q)*.

Consider the axiom

- **N1:** $(\forall p, q, r, s)$
  (*PointLess(p, q, r, s)* $\rightarrow$
  $(\exists x, y)$ (*PointLess(p, q, x, y)* & *PointLess(x, y, r, s)*))

  N1 expresses the density of the ordering $\prec$ on pointclasses. Translating it into the $\prec$ notation should make this clear.

Theorem 26 (Completion) *The theory axiomatised by* **M1 - M5, N1** *is countably categorical, with all countable models isomorphic to INT(Q), and hence is Th(INT(Q)).*

Proof: **N1** just expresses the density of $\prec$. Hence $\prec$ must be a dense, unbounded linear order, and by the *Categoricity of Ordering* Theorem, all such countable orderings are isomorphic to Q. Hence *INT(M)* is isomorphic to *INT(Q)* by the pairing function. Since any model M of **M1 - M4, N1** is isomorphic to *INT(M)*, we have that M is isomorphic to *INT(Q)* by composing isomorphisms.
End of Proof.

We note that since the theory axiomatised by **M1 - M5, N1** is countably categorical, and has the same countable model as the countably categorical theory LM of *[LadMad87.1]*, they are alternative axiomatisations of the 'same' theory, given that the other primitive relations in LM are explicitly syntactically definable from the single relation of $\|$, as shown in *[AllHay87.2, LadMad87.1]* (again, strictly speaking, the theories cannot be identical if they have different sets of primitives, but each is explicitly definable in the other).

5.4 Summary

We have discussed the theories of time intervals VB of Van Benthem, $\mathcal{I}_{\mathcal{AH}}$ of Allen and Hayes, and LM of Ladkin and Maddux. We have characterised the models of $\mathcal{I}_{\mathcal{AH}}$ as structures of intervals over an arbitrary unbounded linear order. The characterisation shows that the Allen-Hayes axioms serve the purposes for which they were introduced. Additionally, along with results in *[LadMad87.1]*, the characterisation has enabled a direct comparison of $\mathcal{I}_{\mathcal{AH}}$ with VB and LM. $\mathcal{I}_{\mathcal{AH}}$ is incomplete, which was intended, and is logically strictly weaker than LM/VB. We indicated how to complete the Allen-Hayes axioms so that the completion is countably categorical and defines the same theory as VB/LM.

Chapter 6

Union-of-Convex Intervals

6.1 Introduction

Convex intervals are intuitively those which have no gaps. Here, we investigate the binary relations that can hold between intervals which are unions of convex intervals. We call such relations *non-convex relations*. These intervals consist intuitively of some (maximal) convex subintervals with convex gaps in between them.

We first present an argument for the necessity of using union-of-convex intervals for specification of temporal dependencies. We discuss an example, the representation of time-sharing a single CPU between multiple processes, and two protocols for doing so. We conclude from the example that intervals of time are the appropriate objects with which to represent the temporal information in this example, and furthermore that intervals which are union-of-convex will be needed in many situations.

We then present a taxonomy of important binary relations between intervals which are unions of convex intervals, and we provide examples of these relations applied to the description of tasks and events. These relations appear to be necessary for such description.

Finally, we provide logical definitions of a taxonomy of general binary relations between non-convex intervals.

The combinatorial explosion of possible binary relations between unions-of-convex intervals is dampened by considering only a subset of all possible relations. However, results of the author and Roger Maddux show that there are infinitely many relations definable in the algebra generated by these intervals [*LadMad87.3*]. The notion of *convex interval* is definable in the algebra, as are the notions of *having exactly (greater than, less than) n maximal convex subintervals*, for each n [*LadMad87.3*].

6.2 The Specification of Time Dependencies

Consider the problem of characterising the properties of a single-CPU computer system. We wish to use a predicate running(P) to say that a process P has control of the CPU. For any process P, running(P) will be true at some times and false at others. The predicate does not yet contain sufficient information about the dependence of its truth value on the time at which it is evaluated. Without knowing anything about the structure of the time domain, we may try to include in the predicate a single parameter to express the dependency. The predicate becomes running(P,d), with the extra parameter d.

What should the values of d look like? The predicate running(P,d) is intended to have a single truth value for appropriate choices of the parameter d. In particular, for some choice d_1, running(P,d_1) will be true, and for some other choice d_2, running(P,d_2) will be false. In this case, we intuitively want to pick out the maximal d_1 such that running(P,d_1) is true over d_1, and hope that this will define the value of running(P,d) for all other d.

We consider two cases - a simple multiprogramming protocol, and a more complex one, to show how this can be done.

6.2.1 A Simple Case

The simplest kind of protocol is a simplified form of batch processing, where a job retains control of the CPU until it has terminated, i.e. all interrupts are disabled.

There will be a single continuous period of time when the job has control of the CPU. Hence running(P,d_1) will be true over a continuous period of time. d_1 is most naturally chosen to represent this period. We shall call d_1 a time interval. Since d_1 has no gaps in it, we call it a convex interval, borrowing a term from topology.

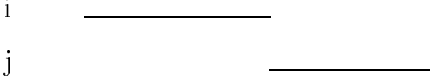
6.2.2 Convex Intervals

Allen's taxonomy was presented in the introduction. We give again some examples of his relations, and the names of the rest, as a reminder.

A convex interval looks like this (with time going from left to right):

Some examples of Allen's relations are:

- i meets j, which means i is before j, but there is no gap (non-empty interval) separating them



- i precedes j, which means i is before j, and there is a non-empty interval separating them

  i _____

  j _____

- i starts j, intuitively i has the same starting point as j, but is otherwise strictly contained in j (ends strictly earlier than j)

  i _____

  j _____

- i overlaps j, meaning i starts before j, and j ends after i

  i _____

  j _____

and there are nine others -

- equals, ends, during, and the converse relations to all these (except equals, which is symmetric)

The interval d_1 could be represented as a pair of points, representing the start and end of the interval. So we could, instead, represent the dependency of the predicate running on time, by including a pair of numbers t_1 and t_2, representing the endpoints of the convex interval d_1, giving us the predicate running(P,t_1,t_2). This form does not extend to the complex case. If we use the *interval* parameter, then when we pass to more complex relations, the model generalises directly.

Furthermore, Allen has argued persuasively that reasoning about the interval d_1 directly is adequate to express, and solve, questions about time dependencies involving convex intervals, without involving the representation of time as points [*All83, AllKau85*] . Ladkin and Maddux in [*LadMad87.1*] provide further evidence, by analysing the mathematical structure of Allen's algebra.

6.2.3 Extending the Interval Model

What should be the choice of the parameter d_2, in the case where running(P, d_2) is false? The natural choice would be the 'complement' of d_1. This time interval, however, is disconnected. It is formed of the collection of points before t_1, along with the collection of points after t_2, and thus looks like:

d_2 _____ _____

This is not a convex interval, an interval with no 'gaps'. It is instead a 'union' of two convex intervals. We shall call such a creature a **union-of-convex** interval, and we shall see that we need these for talking about more complicated protocols.

6.2.4 Representing Context Switching

Suppose we change to a more sophisticated model of the single-CPU system, in which some interrupts are enabled, and context switching occurs.

Again, what is the natural choice of an object d_1 for the predicate running(P,d_1)? Typically, a process will retain control of the CPU for a convex interval, and will then relinquish control when interrupted, regain control for a convex interval, again relinquish, etc. In this situation, the analogous object d_1 will be a union-of-convex interval, as will its 'complement' d_2.

Convex intervals are thus not enough for specification purposes, and we need to reason about union-of-convex intervals in order to represent the situation using a single time parameter.

The general union-of-convex interval looks something like this:

We do not assume any bound on the number of **maximal convex subintervals**. (A maximal convex subinterval can be thought of as a single whole line segment in the picture. We call them **maxconsubints** for short). The unions-of-convex interval calculus can, in fact, express that the interval is the union of **n** maxconsubints, for each natural number **n**, if this is needed.

Consider representing the context-switching situation using point-based time. We no longer can know how many convex periods of time the process will use while executing. Each convex interval may be represented as a pair of points, hence t_1, t_2 will represent the first convex interval, t_3, t_4 the second, etc, and our formula will look like running(P, t_1, t_2, t_3, t_4,....) with the number of parameters indeterminate. Unless we wish to work directly with variadic predicates with arbitrary numbers of arguments, we need to collect the parameters t_1, t_2, t_3, t_4,.... into a single parameter. We would form the list of parameters $<t_1, t_2, t_3, t_4, ...>$ and use this list as a single argument.

6.2.5 Intervals are the Appropriate Abstraction

We are able to reason directly about intervals, without having to represent them as lists of points. So a pair of time points, respectively a list of time points, are just one *representation* of the time **interval**. From Allen's and our work [*op. cit.*] we know that much of the information about the interval can be gleaned directly from the interval calculus. We do not have to define complicated combinatorial relations on pairs of points in order to reason about time. We regard time **intervals** as the appropriate abstraction with which to work.

One significant test of the correct abstraction of a dependency is whether it is extensible to new situations without change. In the example above, the syntax with which

we represented the dependency, running(P,d), did not change as a result of passing to the more complex model. If we had originally modelled the time dependency in the traditional fashion with points, although we started out with two point parameters, we would have been forced to converge to the interval syntax in the more complex case, with an interval interpreted as a list of points. So the interval syntax is eventually forced upon us anyway.

By using intervals, we have allowed the specification to be extensible, although of course the amount of knowledge we shall have to use about time may increase substantially. But this is as it should be. The time dependent information has been factored out of the problem, and now can reside in a time expert, which contains information about dependencies on time, no matter how complex we should choose to get.

6.3 Relation Primitives for Unions of Convex Intervals

Intervals which are unions of convex intervals occur naturally. For example, any recurring time period can be represented: we can regard the period *MONDAYS* as being composed of each individual Monday, *LABOR-DAYS* is likewise the union of convex intervals, consisting of each individual Labor day, the period of the regular weekly meeting with the boss is also a union of convex intervals, each of them the period of single meeting. These kinds of intervals seem to be among the most useful of the non-convex cases, and since we have reason to hope that knowledge gleaned from considering the convex case will transfer in part, we consider unions of convex intervals in detail.

An interval which is a union of convex intervals looks like this

This interval i has three "parts", i.e. maximal convex subintervals, which we call *maxconsubints.*

6.3.1 The Choice of Relation Primitives

There are too many discrete ways that unions of convex intervals may be related to each other. An exhaustive enumeration is infeasible, because

Lemma 14 *The number of relations between unions of convex intervals is at least exponential in the number of maxconsubints.*

In fact, a much sharper result is true (see [*LadMad87.3*]), but we intend only to establish infeasibility here.

Proof:

We prove the theorem by enumerating the relations between two intervals with n maxconsubints and using an inductive argument.

Consider two intervals which are the unions of 2 convex subintervals each. Suppose the first subinterval of each is entirely disjoint from the second subinterval of the other. Then

each first maxconsubint *precedes* the second maxconsubint of the other. The intervals are related in 13^2 ways, including equality, since the first maxconsubints can be related in 13 ways, including equality, and so can the second maxconsubints. When we consider that the first maxconsubint of each may be related by other than *precedence* to the second maxconsubint of the other, e.g. they may *overlap*, or *meet*, we see there are more than 13^2 relations overall.

Now consider two intervals with $n+1$ maxconsubints, such that the first n maxconsubints of each interval all *precede* the final maxconsubint of the other. By the inductive hypothesis, there are more than 13^n ways the subintervals consisting of the first n maxconsubints may be related. The final two maxconsubints may be related in 13 ways, and therefore the total number of possible relations is more than 13^{n+1}. Again, when we consider that the final two maxconsubints may be related by *overlaps* or *meets* to the penultimate maxconsubint of the other, we notice there are many more relations than just those we enumerated in the proof.

Hence we have established the base and the inductive steps, and we draw the conclusion of the induction.

End of Proof.

To avoid the combinatorial explosion implied by the theorem, our basic relations don't depend on the number of maxconsubints. It is intuitively plausible that we don't need relations that depend on the number of maxconsubints for expressing properties of time periods associated with actions, tasks, events or propositions.

However, the relation algebra generated by the relations we consider is still infinite, and still enables us to define the class of intervals with exactly n subintervals, for each n [*LadMad87.3*].

The approach we take will generalise the convex relations, by introducing *functors* that generate non-convex relations from convex relations, by enumerating new subclassifications of relations that weren't there in the convex case, and by enumerating the various relations that arise from considering just the first and last maxconsubints.

Additionally there is one relation, *bars*, which is not obtained by generalising the convex case in some way.

6.3.2 A List of the Relations

The explanations and pictures of the relations obtained cover several pages, in the next section. We list the names here, for easier reference after a first reading. The reader may prefer to omit this section, and proceed to the explanations. The relations on union-of-convex intervals are:

- those generated by the functors *mostly, always, partially, sometimes*, and *disjunction* from convex relations (*always* may be defined in terms of *mostly*)
- *contains*
- *disjoint from*, which splits into:

  - *precedes* and *follows*
  - *meets* and *is met by*
  - *intermingles with*, which splits further into:
    * *disjointly-contains* and *disjointly-contained by*
    * *disjointly-overlaps* and *disjointly-overlapped by*
    * *begins preceding* and *begins following*
    * *ends preceding* and *ends following*
    * *surrounds* and *is surrounded by*

- *strictly intersects*, splitting into
  - *begins after* which splits into
    * *begins in*
    * *begins with*
    * *begins following*
  - *ends before* which splits into
    * *ends in*
    * *ends with*
    * *ends preceding*
  - *begins before* and *ends after*, with the corresponding case splits
  - *begins at* and *ends at*

- *bars*, which is a new relation not generalised from convex relations

6.4 The Definition of the Relation Primitives

In this section, we give the definitions of the relations, and in the section after we follow with examples to show they are naturally occurring. We conclude that these relations are therefore necessary for expressiveness in a calculus of intervals which are unions of convex intervals.

6.4.1 The Intended Calculus

We intend that these relations will be manipulated algebraically, that is, by considering only derived relations defined in the relation algebra generated by these relations [*JoTa52, Mad78, LadMad87.3*]. We believe there might be considerable payoff in the simpler structure of an algebraic theory, although work on this is continuing. We are not concerned here with first-order definability, but elsewhere we are investigating the formulation of the theory of finite-union-of-convex intervals as a first-order theory *[Lad87.5]*. We do not yet have a provably complete axiomatisation of the theory, so we do not discuss it here. However, we have a partial axiomatisation, and all of the relations below have a

first-order formulation in this axiomatisation. In particular, there is a distinguished sort for convex intervals, and axioms guaranteeing that in any countable model this sort will be interpreted by *INT(Q)*. Our investigation of these issues is in *[Lad87.5]*.

The relations are not atomic (not disjoint as sets of interval pairs), and some of them are definable from others. For example, *surrounds* is the conjunction of the relations *begins before* and *ends after*, i.e. *begins before* $\wedge$ *ends after*, where $(i\ (R \wedge S)\ j) \equiv ((iRj) \wedge (iSj))$

Define a *component* of a non-convex interval to be a maxconsubint.

$R^{\smile}$ is the converse relation to R; i.e. $(i\ R^{\smile}\ j)$ iff $(j\ R\ i)$.

We draw example intervals i and j for each relation. We represent them on different lines, but they are intended to be intervals on the same time line. It would be better to use two colors.

The *relation functors* are:

- **mostly**: *i mostly R j*, where R is a convex relation, if, for every component of j, there is a component of i that is R to it. This allows the possibility that there are other components of i, but not of j.

  E.g. *i mostly meets j*

  i ___ _____ _____

  j _____ ___

- **always**: *i always R j*, where R is a convex relation, iff every component of i R some component of j, and every component of j $R^{\smile}$ some component of i. *Always* is definable from *mostly*: *i always R j* iff *i mostly R j* and *j mostly* $R^{\smile}$ *i*, where $R^{\smile}$ is the converse relation to R.

  E.g. *i always meets j*

  i ___ _____ _____

  j _____ ___ _____

- **partially**: *i partially R j*, where R is a convex relation, iff some pairs of components of i and j are R to each other, and all others are disjoint. This allows the possibility that the disjoint intervals may *meet*

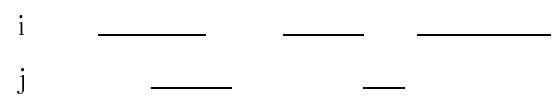
  E.g. *i partially meets j*

  i ___ _____ _____

  j _____

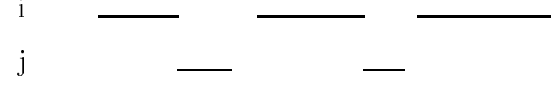
- **sometimes**: *i sometimes R j*, where *R* is a convex relation, iff some pairs (at least one pair) of components of *i* and *j* are *R* to each other

  E.g. *i sometimes meets j*

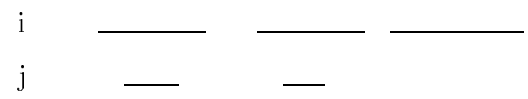
- **disjunction**: *i R* $\vee$.....$\vee$ *Q j* iff every pair of components is related by *R* or .... or *Q* or *precedes* or *follows*

  E.g. *i meets* $\vee$ *contains j*

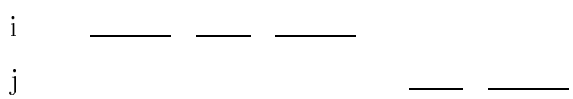
Many of the convex classifications generalise directly to the non-convex case. However, some of the convex classifications get new subclassifications in the non-convex case, notably *disjoint from*, which obtains a new category of *intermingles*, which itself has subclassifications, and *strictly intersects*, which obtains many new subcategories. Some of the new subcategories are valid for both *intermingles*, which is a category of *disjoint*, and *strictly intersects*, which is a different category.

- **contains**; unchanged from the convex case; *i contains j* iff every component of *j* is contained by some component of *i*

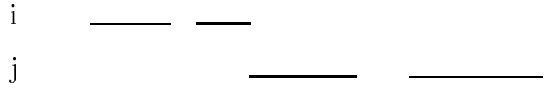
  

- **disjoint from**, which is a symmetric relation, and can be classified into:

  - **precedes**, as in the convex case; precedes is antisymmetric, and *i precedes j* iff all subintervals of *i* precede all subintervals of *j*

  - **follows**, the inverse of *precedes*
  - **meets**, antisymmetric; *i meets j* iff the final component of *i* meets the first component of *j*

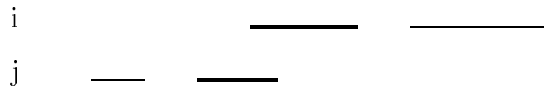
  - **is met by**, the inverse of *meets*
  - **intermingles with**; new in the non-convex case, symmetric, and itself has subclassifications enumerated below

- **strictly intersects**, which has new subclassifications generated by the *relation functors*, as well as the new subcategories enumerated below

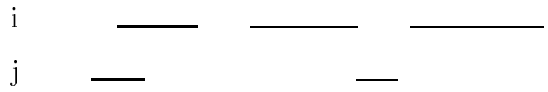
We now enumerate the subclassifications of *strictly intersects*.

- i **begins after** j; which is split into the mutually exclusive cases:

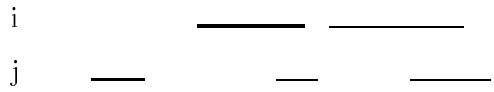
  - i **begins in** j; the leftmost component of i is overlapped by a component of j.

  - i **begins with** j; the leftmost component of i is overlapped by the leftmost component of j

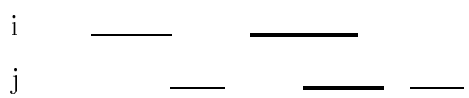
    

  - i **begins following** j; some component of j *precedes* all components of i

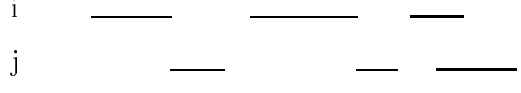
    

- i **ends before** j; split into the mutually exclusive cases:

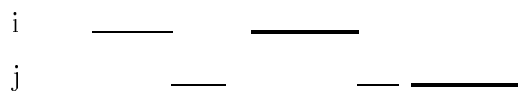
  - i **ends in** j; the rightmost component of i overlaps a component of j

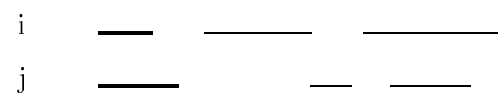
  - i **ends with** j; the rightmost component of i overlaps the rightmost component of j

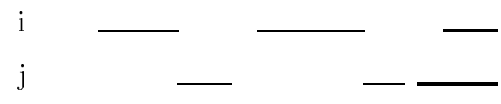
– *i* **ends preceding** *j*; some component of *j* *succeeds* all components of *i*.



- the converses **begins before** and **ends after**, which are also split into cases, giving **begins preceding** and **ends following**, and the converses of the other four relations. We can't find any useful names for these other four at present
- *i* **begins at** *j*; the leftmost component of *i* starts the leftmost component of *j*

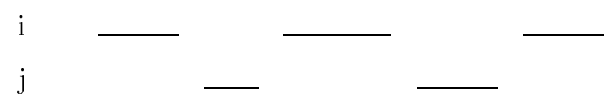


- *i* **ends at** *j*; the rightmost component of *i* finishes the rightmost component of *j*



We enumerate the subclassifications of *intermingles with*:

- *i* **disjointly-contains** *j*; equivalent to *i is disjoint from and surrounds j*



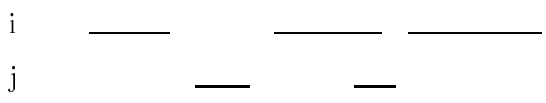
- *i is* **disjointly-contained** *in j*; the converse of *disjointly contains*
- *i* **disjointly-overlaps** *j*; equivalent to *disjoint and begins preceding and ends preceding*

- *i is* **disjointly-overlapped** *by j*; the converse of *disjointly-overlaps*

Finally, we note there are certain classifications that are valid in both the *intermingling* and the *intersecting* cases:

- **begins preceding**, **begins following**, **ends preceding** and **ends following**, with the definitions given in the *intersecting* case valid also for the *intermingling* case

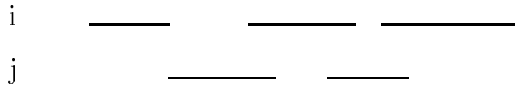
- *i* **surrounds** *j*; equivalent to *i begins before and ends after j* in the *intersecting* case, and *disjointly-contains* in the disjoint case. We illustrate the *intersecting* case:



- *i is* **surrounded by** *j*; the converse of *surrounds*

Additionally, there is a polyadic relation that is of some importance. We illustrate it for two intervals, and it should be clear how to generalise it to many. In our calculus, we only consider the binary case [*LadMad87.3*].

- *i* **bars** *j*; the union of *i* and *j* is convex. *Bars* is a symmetric relation, and commutative in the general case



6.5 Examples of Relations Between Non-Convex Intervals

We illustrate the relations by examples drawn from general processes, procedures, tasks and occurrences. We conjecture that the most useful applications of the calculus will be in the areas of task description and management, action theory and process theory.

The reader may observe that many of the relations above are converses of relations already included in the enumeration. We include enough examples to show that the relations are useful and natural for descriptive purposes. Each example has the form of expressing a relation between two tasks, events or actions. Suppose P is such a creature. Then we attach to P the interval *int(P)* (see chapter 7). Then, for two tasks, P and Q, we can consider the interval relation *R(int(P), int(Q))* between the associated intervals. This relation R appears above each example.

- *mostly meets*:
  - after the committee has reviewed the market on Monday, the brokers may act to buy the desired stock.
  - when designs are finalised, the programmers assigned may be immediately available for the implementation task
- *always meets*:
  - after the committee has reviewed the market on Monday, the brokers always act to buy the desired stock.

  - when designs are finalised, the programmers assigned should be immediately available for the implementation task

- *always overlaps*:

  - investigation of the system crash starts before system service is restored, and continues afterwards.
  - preparations for performing the task should be initiated while the design team is finishing the detailed description

- *(overlaps* $\vee$ *contains)*:

  - investigation of the system crash starts before system service is restored, and sometimes continues afterwards.
  - these tasks are always concurrent:
    work on the processor configuration and the distributed system design task

- *partially contains*:

  - if you need to cross the road, you do so while there's no traffic.
  - task 34 should only be worked on when Fred and Mary are available for it

- *partially meets*

  - when system service degrades, sometimes we need to reboot
  - the distributed system design task may need to be followed by a feasibility study

- *begins in*:

  - the emergency procedures were introduced and used during the company reorganisation
  - the implementation should be commenced while the system is being configured

- *ends in*:

  - the final reorganisation before dissolution occurred during last year's first financial crisis
  - the implementation should be completed while the system is in the test stage

- *begins with*:
  regular system backups were started during the first time the system lost a drive because of a head crash

- *ends with*:
  communication with other machines ceased with the last failure of the main processor of the gateway

- *begins at*:
  tasks A and B are independent starting tasks for the project

- *ends at*:
  finishing a project at the deadline date

6.6 Relations for General Non-Convex Intervals

General non-convex intervals correspond to arbitrary sets of time-points, in a points-based model of time. In this section, we enquire how the relations enumerated for union-of-convex intervals may also be defined for general non-convex intervals

6.6.1 The Choice of Primitives

To define the relations logically, we can either assume that certain relations are primitive, that a notion of *subobject* is primitive, or that *part of* is primitive (subobjects are *parts of* their superobjects, and *contains* and subobjects are interdefinable also, as indicated below).

The notion of *subobject* may be introduced by definition also from interval *operators*. For example, if we have a notion of *intersection* of intervals, which is natural in certain representations such as that of a set of points, we may stipulate that the result of any *intersection* of intervals is a subobject of all those intervals. For another example, the argument intervals of a *union* operation on two intervals are subobjects of the result of the union. We do not consider operators on intervals in this work, and prefer to avoid them where possible, since the addition of operators vastly complicates the algebra, which would be no longer simply a relation algebra in the sense of Tarski.

6.6.2 A List of the General Non-Convex Relations

We list the general non-convex relations for easier reference, and we follow in the next part with the informal definitions. Again, a reader may wish to omit this list, and proceed to the definitions. We note that the list is similar to the union-of-convex list, but the reader should note that for the general non-convex case we have no relations generated by functors from the convex relations, there is no *surrounds* relation, and there are no subdivisions of *begins after*, *ends before*, *begins before*, and *ends after*. The relations on general non-convex intervals are:

- *contains*

- *disjoint from*, which splits into:

  - *precedes* and *follows*

  - *meets* and *is met by*
  - *intermingles with*, which splits further into:
    * *disjointly-contains* and *disjointly-contained by*
    * *disjointly-overlaps* and *disjointly-overlapped by*
    * *begins preceding* and *begins following*
    * *ends preceding* and *ends following*

- *strictly intersects*, splitting into

  - *begins after*
  - *ends before*
  - *begins before* and *ends after*, with the corresponding case splits
  - *begins at* and *ends at*

- *bars*

6.6.3 The Informal Definitions of the Relations

We discuss in this part the definitions of the relations on general non-convex intervals. It should be easy to see how to convert the informal definitions into formal first-order definitions in a first-order language with the specified primitives. Firstly, we provide some background discussion of the primitive notions that we need in order to provide the first-order definitions of the relations.

We shall refer to the basic object of time (point, interval or whatever) as a *time object*. For example, if you prefer points-based time notions, then you will represent intervals as sets of points, and your basic time object will be a set of points. We assume at present that there is no *null* object.

Allen and Hayes [*AllHay85, Hay87*] are able to define the convex relations from one primitive, in a first-order manner. Van Benthem [*vBen83*] uses two. We have more, to enable us to keep the logical form of the definitions to a statement with at most two bounded quantifiers. We give English descriptions of the definitions, but it should be obvious how to translate them into a first-order logical language with the declared primitives.

Given a basic time object, we define its *subobjects* as those objects which *are contained in* it. So, for example, for the set-of-points notion, the subobjects of i are precisely the subsets of i. For the convex interval notion, subobjects are convex subintervals, and for the unions-of-convex-intervals notion, subobjects are unions of convex subintervals. We assume that in a given ontology of intervals, either the subobjects are precisely defined, or the notion of *containment* is primitive. They are interdefinable, as indicated below.

If *subobject* is a primitive notion, or, alternatively, *containment* is a primitive relation, and *precedes* and *meets* are also primitive relations, we can give the following definitions of the relations. We assume that the conditions in the subclassifications are conjoined

with the conditions in the apropriate superclassification, so that we may avoid repeating parts of definitions; e.g. *i disjointly contains j* is to be read as *(i is disjoint from j)* $\wedge$ *some subobject of i is .....*

i contains j: j is a subobject of i

i precedes j: a primitive relation

i meets j: also primitive

i is disjoint from j: i and j have no common subobjects.

We note that this definition is adequate only because there is no *null* object, which would have to be a subobject of every object.

i disjointly contains j: some subobject of i *precedes* all subobjects of j and some subobject of i *follows* all subobjects of j

i disjointly overlaps j: some subobject of i *precedes* all subobjects of j and some subobject of j *follows* all subobjects of i

i begins preceding j: some subobject of i *precedes* all subobjects of j

i begins following j: some subobject of i *follows* all subobjects of j

i ends preceding j: some subobject of j *follows* all subobjects of i

i ends following j: some subobject of i *follows* all subobjects of j

i strictly intersects j: (also known as i *overlaps* j) i and j have a common subobject, and neither i *contains* j nor conversely.

Notice this relation is symmetric. We can turn this relation into an antisymmetric relation, as *overlaps* is for Allen, with our primitives, by asserting that the subclassifications below are mutually exclusive and exhaustive of the *strictly intersects* relation.

However, we would consider this move to be a claim about the structure of a particular interval model. For example, if one were to consider axiomatising the structure of closed sets of real numbers, one might want to assert that *meets* is the empty relation (since two closed sets can only meet by intersecting at a point, which is a closed set); that *precedes* is a dense (partial) order (any two non-intersecting closed sets may be separated by a closed set); etc.

We prefer to leave the assertion of exhaustiveness as a structure axiom if it is needed. Thus we must allow *strictly intersects* the luxury of symmetry.

i begins after j: some subobject of j *precedes* all subobjects of i

i ends before j: some subobject of j *follows* all subobjects of i

i begins before j: some subobject of i *precedes* all subobjects of j

i ends after j: some subobject of i *follows* all subobjects of j

i begins at j: there is no subobject of i that precedes all subobjects of j, and symmetrically for j and i

i ends at j: there is no subobject of i that follows all subobjects of j, and symmetrically for j and i

For the case of *bars*, we did not include a monadic predicate in our language for selecting *convex* intervals. It is obvious that we would need such a predicate, whether primitive or defined, in order to define *bars*, however one should exercise some care in the choice of definition.

For example, if we were to provide a predicate for *convexity*, we might try to define:

i bars j: there is a convex object k such that the subobjects of k are exactly the subobjects of i and j

Such an attempt doesn't work: consider arbitrary sets of real numbers, with subobjects being subsets, and convex objects being convex sets.

Let $A = [a, b)$, and $B = [b, c]$, where $a < b < c$. Then intuitively $A\ bars\ B$ since $A \cup B$ is $[a, c]$ which is convex. However, let x, y be such that $a < x < b < y < c$. $[x, y]$ is a subobject of $A \cup B$, but isn't a subobject either of A or of B.

The following definition, due to Stuart Russell, avoids this problem *[Rus87]*:

i bars j: there is a convex object k such that ((all the subobjects of i and j are subobjects of k) and (no subobjects of k are disjoint from i and j))

It's probably most convenient to include *bars* as a primitive relation.

We note that primitives for interval-based notions of time are discussed extensively in [vBen83] and [Hum79]. We note further that our proposed classification is much finer-grained than the primitives in these works, but that all are first-order definable from the primitives used therein. We have mentioned before that our motives for such profusion are algebraic. We note that [vBen83] also provides an extensive discussion of the types of interval structure that may be obtained in different domains of application.

Chapter 7

Synthesis of Concurrent Processes

7.1 Synopsis

In order to justify our claims for the Extended Interval Calculus as a specification language, we consider here the problem of synthesising code solving a simple concurrency problem, from high-level specifications in the Extended Interval Calculus.

The problem we consider is a simple readers/writers problem, which we illustrate with two readers and two writers for clarity. The technique we exhibit works for mutual exclusion problems similar to this, and also for limited-resource problems.

7.2 Some Relations on Unions-Of-Convex Intervals

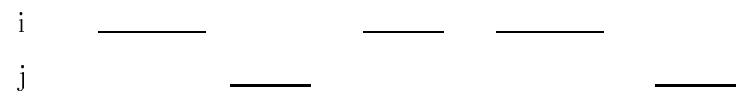
In this section, we re-introduce the relations on union-of-convex intervals that we shall use in this example.

- i always-meets j. Any maxconsubint of i has to meet a maxconsubint of j, and every maxconsubint of j is met by some maxconsubint of i.

- i always-(precedes-or-meets) j. Every maxconsubint of i either precedes or meets some maxconsubint of j, and every maxconsubint of j is preceded by or met by some maxconsubint of i.

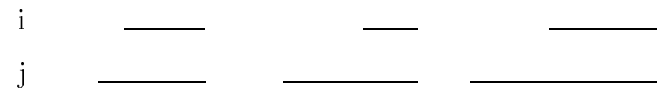
- i always-(follows-or-met-by) j. The converse of the relation always-(precedes-or-meets) above. The **converse** relation is obtained by just exchanging the order of the arguments, i.e. S is the converse of R just in case the following condition is satisfied: (i R j) if and only if (j S i).

- i disjoint-from j. i and j have no subintervals in common. (All of the relations illustrated above are subrelations of disjoint-from).

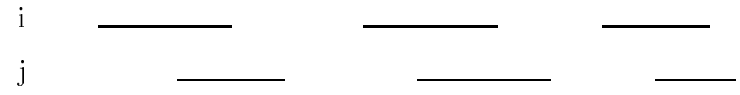


- i always-starts j. Every maxconsubint of i starts some maxconsubint of j, and every maxconsubint of j is-started-by some maxconsubint of i

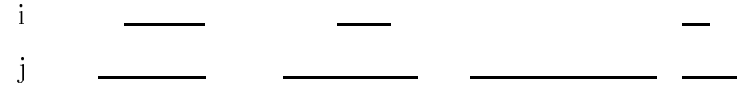
- i always-ends j. Every maxconsubint of i ends some maxconsubint of j, and every maxconsubint of j is-ended-by some maxconsubint of i. (Ends, in Allen's calculus, is similar to starts, but with the ends of the intervals coinciding, instead of the beginnings coinciding).



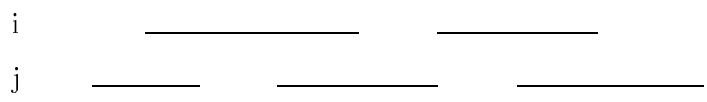
- i always-overlaps j. Any maxconsubint of i has to overlap a maxconsubint of j, and every maxconsubint of j is overlapped by some maxconsubint of i.



- i contained-in j. Every subinterval of i is contained in some subinterval of j. Contained-in includes always-starts and always-ends as subrelations, and is definitionally equivalent to always-(starts-or-ends-or-during)

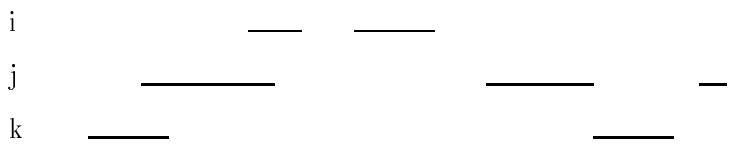


- i bars j. The 'union' of i and j is a convex interval.



Additionally, we shall need the operator combine. There are no operators in the pure interval calculus, which is just an algebra of relations on union-of-convex intervals. The operator combine takes as argument a **set of intervals**, and returns the single union-of-convex interval which is the 'union' of every interval in the set [*Lad86.2*]. For example, the relation (i bars j) says the same as the assertion (combine({ i , j }) is convex). The property of **convexity** is definable in the calculus, as a specific subrelation of the identity relation.

Suppose the set of intervals is A = { i , j , k } , where i, j and k are



Then combine({i , j , k }) will be the interval

(These two diagrams are aligned)

7.3 Synthesis of Concurrent Process Specifications

Our technique can produce executable synchronisation skeletons for a large class of mutual exclusion problems. We can refine specifications of concurrent systems of the following kinds:

- limited-resource contexts, where there are a fixed number of servers of the same type that an arbitrary number of processes may access, for example, disks.
- contexts of exclusive access to a single resource, where the access is of different kinds for different types of process, for example the readers/writers problem.

We assume familiarity with the readers/writers example, which has the attraction that there are mutual exclusions of two different kinds:

- Readers do not exclude other readers, but they must exclude writers
- Writers exclude everyone

Thus the readers/writers problem is more complex than a simple mutually-exclusive-access problem. Two examples of readers/writers situations:

- A stock broker's information system. An individual buffer represents the current price of a single stock. The buffer is written by a central unit at the exchange. The querying units are individual agent's workstations.

- A communication system implemented as a non-broadcast message system. Messages are written by a distributed multi-processor system, into a buffer read by many personal portable units in the field. These units may also update the contents of messages for others to read.

7.3.1 The Readers/Writers Specification

We specify the problem informally, then we give the interval calculus specification, to show how close to the original informal specification it is. We regard it as a major advantage of the interval-based approach that it lends itself naturally to paraphrasing informal specifications.

The informal specification is:

- There is a reusable buffer containing time-sensitive information.

- The buffer is updated by many writers. However, only one writer at a time may update, and only when no readers are reading.

- All readers may read the buffer concurrently, but only when no writers are writing.

7.3.2 Specification in the Interval Calculus

We only require three statements in the calculus to specify the readers/writers situation. The formal specification is quite natural, and very close to the informal specification. We include the requirement that reader processes are different from writer processes, for convenience only.

We associate an interval with each process, and with some predicates also. We represent the time interval associated with a process or a predicate as interval-of(P). Most of the specification which follows is written in terms of constraints upon the intervals.

Read-ing is a predicate that is true whenever a reader is reading. We add many such predicates in the refinement technique. Read-ing(Y) is a predicate that is true whenever the particular process Y is reading. Similarly for write-ing and write-ing(Y).

The specification of the readers/writers configuration is in Horn-clause form, using relations in the interval calculus:

- interval-of(read-ing) disjoint-from interval-of(write-ing)

  specifying the exclusion of the class of read-ers from the class of write-ers

- write-er(X) $\wedge$ write-er(Y) $\wedge$ X $\neq$ Y $\Rightarrow$
  interval-of(write-ing(X)) disjoint-from interval-of(write-ing(Y))

  specifying the mutual exclusion of write-ers

- We shall also require for convenience, that readers are disjoint from writers, i.e.

  read-er(X) $\wedge$ write-er(Y) $\Rightarrow$ X $\neq$ Y

Our object is to produce automatically, from such a specification, *synchronisation skeletons* for processes, that include statements incorporating the synchronisation primitives, along with assertions that are to be maintained as true throughout execution. We discuss the target language below.

7.4 Refinement of The Specification

7.4.1 The Target Language

Our target language for the refinement is very rich. It consists of a subset of *Concurrent* $REFINE^{TM}$ [1], along with certain constructs from the language of the *TUS* time system. $REFINE^{TM}$ [2] allows assertional as well as procedural specification, and allows compilation of combined specifications which include assertions about predicates as well as procedural definitions of functions, into running code. Our refinement technique was developed to illustrate how very-high-level specifications of concurrent systems could be transformed into specifications that could in principle be accomodated in an existing system. We shall need to build a sequencer for our target language that satisfies the assumptions discussed later.

The assertional part of the language has the ability to maintain conditions specified in the interval calculus. The procedural part uses *guarded commands* as synchronisation primitives.

Although the target language has the capabilites mentioned, it does not know about mutual exclusion, or indeed about any concurrency protocols. (Of course the sequencer uses some mutex primitive). The major contribution of our refinement technique is to explicate the knowledge about the concurrent protocols implied by the very-high-level specification, so that a language with
truth-maintenance capability and the ability to execute conditional statements with delay is sufficient to implement the protocol, and to do this all automatically.

7.4.2 Guarded Commands

The procedural part of the processes which we synthesise will employ guarded commands of the following form:

- P!foo
  Wait until predicate P is true, then execute procedure foo.

[1] *Concurrent* $REFINE^{TM}$ is a trademark of Reasoning Systems Inc.
[2] $REFINE^{TM}$ is a trademark of Reasoning Systems Inc.

7.4.3 The General Strategy

We know:

- The predicate P in a guarded command P!foo should be true precisely when it is safe to execute foo
- The procedures foo for which we need guards are the critical sections in the process

The refinement proceeds by

- using a precise formulation of the notion of *critical section*, in the interval calculus.
- synthesising precise constraints for the guard predicate P, in the interval calculus

The semantics of a guard is such that a process executing a critical-section procedure foo will first test the guard and wait for it to become true. We call this the request-foo part of critical section execution. When the guard becomes true, then foo may be executed by the process, and we call this the execute-foo stage. The exclusion constraints clearly should only apply to the execute-foo stage. We call the guard predicate for the critical section enable-foo.

When we have identified critical sections and synthesised assertions for the guard predicates, we have to assemble the synchronisation primitives into the skeleton for a process. In some scenarios, such as a message-send / confirmation-receive situation, it will be implied by the specification that the message send must occur before the receive of confirmation for that message, in order for the system to operate correctly. Hence the synchronisation primitives associated with a process will have some implied order. We comment on how to do this in the next subsection.

In our readers/writers example, there will be only one synchronisation statement in each process, so the question of identifying order will not arise.

Our technique consists in the following steps:

- Introducing all the auxiliary predicates we need, such as the -ing predicates
- For a given procedure foo, constructing the set of procedure executions with which it is mutually exclusive. If this set is non-empty, then foo will be a **critical section** and we shall need to include it as a guarded command in the skeleton for its calling process
- For all critical sections foo, we formally decompose foo into request-foo and execute-foo parts
- We construct all the conditions which assert that the guard predicate enable-foo is true when and only when it is safe to execute foo
- Finally, we order the guarded statements within the calling process as needed

7.4.4 Formalisation of Exclusion and Ordering

Procedures which are mutually exclusive may be specified using the relation disjoint-from in the interval calculus. If the two procedures may not execute concurrently, then the interval in which one is executing will be disjoint-from an interval in which the other is executing. Thus critical sections may be identified by constructing for each procedure the set of procedures provably disjoint from it. We call the set of this set the disable-set of a critical section.

We have reduced the problem of identifying critical sections not explicit in the specification to the problem of deducing the domain of the particular relation of disjoint-from from a given specification in the interval-calculus.

We may order critical sections within a process by identifying, for each critical section, the set of critical sections in the process that always-(meet-or-precede) it. We may then use standard sorting algorithms (the specification lends itself naturally to the formulation of Quicksort in [*ManWald*], but any method may do) to determine the implied order in which these critical sections shall appear in the process skeleton.

7.4.5 A Small Readers/Writers Example

In the general case, the example includes variables for processes, as in the specification above, and the statements resulting from the refinement are all prefixed with conditions of the form 'read-er(X) $\Longrightarrow$ .....', etc, in all cases. For clarity, we work with a specific number of processes, to avoid all the conditionals.

We suppose four processes: two readers R_1 and R_2, and two writers W_1 and W_2. We use the contraction int for interval-of to make the result easier to read.

7.4.6 Refining the Assertions

- The specification.

  int(read-ing) disjoint-from int(write-ing)

  write-er(X) $\wedge$ write-er(Y) $\Rightarrow$
  int(write(X)) disjoint-from int(write(Y))

  read-er(X) $\wedge$ write-er(Y) $\Rightarrow$ X $\neq$ Y

- We set up the -ing predicates. The conditions ensure that, for example, the read-ing predicate is true when and only when some read-ing(X) predicate is true.

  Combine({int(read-ing(R_1)), int(read-ing(R_2))}) = int(read-ing)

  Combine({int(write-ing(W_1)), int(write-ing(W_2))}) = int(write-ing)

- We form the disable-sets. These are the sets of all predicates which are *provably disjoint* from a given procedure. We have predicates associated with each procedure, so if we are able to determine in the interval calculus that two procedures have to occur disjointly, each will be in the disable-set of the other. This identifies the critical sections of a process, by identifying the procedures that execution of a section

may not proceed concurrently with. If we take the intervals associated with these predicates, then this is the collection of time intervals on which the section may not proceed. If there is more than one interval in this collection, we can then turn the collection into a single interval by using the *combine* operator. We call the result the disabling-int of a section.

These statements all follow from the single disjointness condition in the specification.

disable-set(read-ing(R_1)) = { writ-ing }

disabling-int(read-ing(R_1)) = { int(writ-ing) }

disable-set(read-ing(R_2)) = { writ-ing }

disabling-int(read-ing(R_2)) = { int(writ-ing) }

disable-set(write-ing(W_1)) = { write-ing(W_2), read-ing }

disabling-int(write-ing(W_1)) = combine({ int(write-ing(W_2)), int(read-ing) })

disable-set(write-ing(W_2)) = { write-ing(W_1), read-ing }

disabling-int(write-ing(W_2)) = combine({ int(write-ing(W_1)), int(read-ing) })

- Now we have the critical sections identified, we decompose them into request and execute intervals. The critical sections are read-ing(R_1), read-ing(R_2), write-ing(W_1), write-ing(W_2). We also replace the disjointness conditions on a critical section by the identical conditions on its execute-* part. There are no disjointness conditions on the request-* part, since a process may request entry into a critical section at any time.

  int(request-read-ing(R_1)) always-starts int(read-ing(R_1))

  int(request-read-ing(R_2)) always-starts int(read-ing(R_2))

  int(request-write-ing(W_1)) always-starts int(write-ing(W_1))

  int(request-write-ing(W_2)) always-starts int(write-ing(W_2))

  int(execute-read-ing(R_1)) always-ends int(read-ing(R_1))

  int(execute-read-ing(R_2)) always-ends int(read-ing(R_2))

  int(execute-write-ing(W_1)) always-ends int(write-ing(W_1))

  int(execute-write-ing(W_2)) always-ends int(write-ing(W_2))

  int(request-read-ing(R_1)) always-meets int(execute-read-ing(R_1))

  int(request-read-ing(R_2)) always-meets int(execute-read-ing(R_2))

  int(request-write-ing(W_1)) always-meets int(execute-write-ing(W_1))

  int(request-write-ing(W_2)) always-meets int(execute-write-ing(W_2))

  int(execute-read-ing(R_1)) disjoint-from int(execute-write-ing(W_1))

  int(execute-read-ing(R_2)) disjoint-from int(execute-write-ing(W_1))

  int(execute-read-ing(R_1)) disjoint-from int(execute-write-ing(W_2))

  int(execute-read-ing(R_2)) disjoint-from int(execute-write-ing(W_2))

  int(execute-write-ing(W_1)) disjoint-from int(execute-write-ing(W_2))

- A process may enter its critical section when none of the predicates in the disable-set of that process is true. We define the enable-* predicate for each critical section, so that it is constrained to be true precisely when no predicate in the disable-set is true. This amounts to saying that the enable-* predicate should be true on precisely the *'complement'* interval of the *disabling-int.* We have no operators, but only binary relations on intervals, so we cannot just use a *complement* operator directly. However, we can achieve the effect of specifying it by saying that the enable-* both bars and is disjoint-from the *disabling-int.*

  int(enable-read(R_1)) bars disabling-int(read-ing(R_1))

  int(enable-read(R_1)) disjoint-from disabling-int(read-ing(R_1))

  int(enable-read(R_2)) bars disabling-int(read-ing(R_2))

  int(enable-read(R_2)) disjoint-from disabling-int(read-ing(R_2))

  int(enable-write(W_1)) bars disabling-int(write-ing(W_1))

  int(enable-write(W_1)) disjoint-from disabling-int(write-ing(W_1))

  int(enable-write(W_2)) bars disabling-int(write-ing(W_2))

  int(enable-write(W_2)) disjoint-from disabling-int(write-ing(W_2))

- We ensure that the enable-foo(X) predicates can only be true after a request-foo-ing predicate has become true, and that the enable-foo(X) predicate becomes false after X enters the execute-foo(X) procedure, and before termination of that procedure. This is necessary, because the enable-foo predicate becomes false when some other process excludes all of its disable-set. Clearly, we should not have execute-foo starting at this point, so the intervals associated with enable-foo and execute-foo need to overlap and not just meet. Similarly with request and enable.

  int(request-read-ing(R_1)) always-overlaps int(enable-read(R_1))

  int(request-read-ing(R_2)) always-overlaps int(enable-read(R_2))

  int(request-write-ing(W_1)) always-overlaps int(enable-write(W_1))

  int(request-write-ing(W_2)) always-overlaps int(enable-write(W_2))

  int(enable-read(R_1)) always-overlaps int(execute-read-ing(R_1))

  int(enable-read(R_2)) always-overlaps int(execute-read-ing(R_2))

  int(enable-write(W_1)) always-overlaps int(execute-write-ing(W_1))

  int(enable-write(W_2)) always-overlaps int(execute-write-ing(W_2))

We now have a complete set of assertions guaranteeing that the enable-read and enable-write predicates do the right thing. For example, the constraints on enable-foo(X) allow us to prove the following statement of logic, for all time periods

enable-foo(X) $\Longrightarrow \neg A_1 \wedge \neg A_2 \wedge .. \wedge \neg A_n$

where $A_1..A_n$ are the predicates in disable-set(foo(X)). So, intuitively, enable-foo(X) can only hold when none of its disabling conditions is true.

7.4.7 Generating Process Skeletons

The synchronisation skeletons in this example are trivial, since each process has only one synchronisation statement in its skeleton, which is either

enable(read)!execute-read

or

enable(write)!execute-write

Most of the work of ensuring that the guards are correct has been offloaded to the assertions. If the assertions are maintained, then the process skeletons will behave as required.

7.4.8 Execution of The Lower-Level Specification

The very-high-level specification is not obviously executable as is. This is part of the motivation for the refinement. We justify our claim that the resulting specification is executable in principle by indicating how the lower-level refined specification will be executed by a smart compiler. We indicate assumptions we make on the sequencer as we go. We enumerate the assumptions in a later subsection.

We have generated a set of assertions that specify how the truth value of certain predicates must be related. For example, if a process X executes procedure foo, request-foo-ing(X) is immediately true. Suppose enable-foo(X) (the guard predicate) is false. Then X remains in the (no-op) procedure request-foo since the procedural specification of X does not allow execute-foo(X) until enable-foo(X) is true, by the semantics of the guarded command enable-foo!execute-foo.

Enable-foo(X) can be false only because some other predicate, incompatible with enable-foo(X) according to the assertions, is true. This predicate will of necessity be execute-bar-ing(Y) for some other critical section bar(Y) of a process Y, by our characterisation of critical sections, and our refinement of these into request and execute parts. Y may be the same as X here, because the actual execution time of the execute-foo(X) procedure may be shorter than the length of time that execute-foo-ing(X) is held true, because of inefficiencies in the sequencer - see below.

By our assumption of finite termination of critical sections, execute-bar(Y) terminates, and the sequencer makes the predicate execute-bar-ing(Y) false. By our assumption on arbitration, the sequencer sets some mutually compatible set of guard predicates to true. This triggers the processes, which have been waiting on the truth of the guards in the guarded commands.

In particular, enable-foo(X) may be one of those predicates made true by the sequencer. Enable-foo(X) is the guard for execute-foo(X), and hence the process X starts execute-foo(X), and simultaneously the predicate execute-foo-ing(X) is made true, so that all other critical sections that are exclusive of execute-foo(X) will be inhibited from executing until execute-foo-ing(X) becomes false again, i.e. when execute-foo(X) terminates. If enable-foo(X) is not made true this time around, then it will be on some subsequent round by the assumption of fairness.

Suppose there is no inhibition on the truth of enable-foo(X) when X enters request-foo(X). Then the sequencer makes
enable-foo(X) true immediately, by assumption, (or rather, a short time afterwards, by the assertions) and X thus enters execute-foo(X) immediately. A short time afterwards, and before execute-foo(X) terminates, enable-foo(X) is made false again by the sequencer (again forced by the assertions).

If the sequencer does not make enable-foo(X) false before the task execute-foo(X) runs through its code, then execute-foo-ing(X) still has to be kept true by the sequencer, and thus other incompatible critical sections inhibited, until the sequencer makes enable-foo(X) false, when it may subsequently (and, one hopes immediately) make execute-foo-ing(X) false and allow other sections to continue.

7.4.9 What We Have Accomplished

We have started with a specification of a synchronisation problem which is very close to the crude informal description of the problem. We have refined this specification into a more detailed, lower-level specification.

We have shown how to accomplish automatic identification of critical sections, and how to automatically order the resulting synchronisation primitives within each process. We have done this by formalising in the interval calculus.

The refinement shows how very-high-level specifications may be reduced to lower-level specifications.

We have shown how the resulting refined specification can be executed.

We now address the sequencing assumptions.

7.4.10 Sequencing Assumptions

We make the following four assumptions:

- The knowledge required by each process of the global state of the concurrent system is at least the current truth value of all its guard predicates.
- When a process requests entry into a critical section, it may not terminate until it has entered and exited its critical section.
- When a process X requests entry into a critical section foo, if the truth of enable-foo(X) is consistent with the predicates that are currently true, the sequencer makes enable-foo(X) true.
- Arbitration of the following kind:
  - Suppose guard predicate P_1 is true, and excludes the two other guard predicates S_1 and S_2. Also suppose that $S_1 \Longrightarrow \neg S_2$, i.e. these two guards are also mutually exclusive. For example, P_1, S_1 and S_2 are mutex guards for different processes on a single resource. At most one may be true at any given time.

We therefore require the following kind of arbitration:

When the procedure guarded by P_1 is finished, the sequencer has to choose one of S_1 or S_2 to enable, and leave the other one false.

Arbitration generalises to multiple guards. Suppose we have a collection of guard predicates all incompatible with the current state, and of which some are incompatible with others. When the current state changes, the arbiter picks a consistent subset of them to make true, leaving all others false. (The solution where only one is picked is sufficient, but inefficient).

Additionally, we need to assume that arbitration is accomplished in a **fair** manner, i.e. , if a process is waiting for a guard to become true, it will not be locked out for ever by the sequencer if it is at all possible for the guard to become true. This property of solutions to concurrent problems is known as **liveness**. **Liveness** properties are usually guaranteed under the assumption of a fair sequencer, as we have above, and as in [*ManWol84*].

7.4.11 Completeness of The Interval Calculus

We have indicated that we can discover critical sections automatically by calculating in the interval calculus.

For this to work, we need a *completeness theorem* to the effect that all semantic consequences of appropriate sorts of specifications are actually derivable in the interval calculus. We have not yet provided a formal proof of this property for all types of specifications, although for quantifier-free specifications, it follows from the completeness theorem for equational logic (the logic in which the interval calculus is represented).

We refer the interested reader to [*LadMad87.1*].

7.4.12 Generalising the Method

In the course of our example transformation, we have indicated that the general approach may be used to refine a number of similar types of very-high-level synchronisation specifications. We do not have space to explicate the steps of the general method, but suffice it to say that all components of the method have been used in the example, with the exception of the ordering of the synchronisation primitives within the process skeleton. We have already commented on how this ordering is accomplished.

7.4.13 Limitations of the Method

It should be obvious that we cannot reduce all synchronisation problems by our method. For example, applying the steps above to an arbitrary specification of the dining philosophers problem would not be guaranteed to solve the livelock issue for that problem. In general, the problem of refining a specification to eliminate all potential livelock seems to be very hard, and we believe it's probably recursively undecidable. Justification of such a claim will have to await precise formulation of the nature of livelock in general.

7.5 Summary

We have introduced the calculus of unions-of-convex intervals, as defined in [*Lad86.1, Lad86.2*] as a specification language for concurrent systems.

We have argued that intervals are the correct abstraction for reasoning about time dependencies, allowing flexible, extensible specification of these dependencies,

We noted that the unions-of-convex interval calculus is more expressive than temporal logic, since temporal logic may be translated into it, and the calculus provides real-time specification capabilites that are absent from temporal logic.

We have used the readers/writers example to indicate a refinement method for a large class of concurrency specifications. Our refinement technique consists of making explicit the protocols implied by the very-high-level specification, using the capabilities of the interval calculus.

We have shown how to reduce the two problems of automatically identifying critical sections, and automatically ordering synchronisation statements in the process skeleton, to derivation in the interval calculus, via formal specification in the calculus.

We have indicated how the lower-level specification resulting from our refinement may be executed.

We hope to have established that the interval calculus is a natural and powerful tool both for specification and refinement of concurrent process configurations.

Chapter 8

The Full *TUS* and Its Implementation

We define the Full *TUS*, which includes operators necessary for implementing the Extended Interval Calculus, within a system which incorporates a wide-spectrum specification language, and declarative forms of specification. We assume some logic programming capability, namely the ability to compile Horn clause specifications. We are able to axiomatise the operators by means of Horn clause axioms. The high-level data structure that we assume is just some finite set theory. All this is available in the *REFINE* system in which we implemented a prototype of the *TUS*.

In addition to the pure interval concepts, we also incorporate the notion of *duration* into the *TUS*. *Duration* is given in terms of length units such as days, although we discuss it here in the abstract, namely by means of axioms that will be true no matter what length units are used.

8.1 Notation

We assume the reader is familiar with standard logical notation, in particular the connectives and quantifiers.

- $\wedge \ \vee \ \neg \ \Rightarrow \ \forall \ \exists$

- We refer to the relations in [*All83*] as *convex* relations. All convex relations are irreflexive and antisymmetric, except for equality. *Non-convex* intervals or relations are those for unions of convex intervals in chapter 5. A picture of such a beast will look like a sequence of lines with gaps, when drawn in one dimension. The lines are the *maximal convex subintervals*, or *maxconsubint*s.

- $\|$ is the convex relation *meets*.
  $(i \| j)$ iff i is before j with no interval occurring between them

- $\ll$ is the convex relation *contained-in*
  $(i \ll j)$ iff i is a strict subinterval of j, i.e.
  i *starts* $j \vee i$ *during* $j \vee i$ *ends* j

- $\prec$ is the convex relation *precedes*
  $(i \prec j)$ iff i is before j, with some other interval occurring between

- $\oslash$ is the convex relation *overlaps*
  $(i \oslash j)$ iff, intuitively, i starts before j, and finishes before j.

- $(R_1 \vee R_2 \vee ...)$ is the non-convex relation of *disjunction*, where the R_i are convex relations.
  $(i\,(R_1 \vee R_2 \vee ...)\,j)$ iff corresponding maxconsubints of i and j are one of the R_i to each other (different subintervals may be related by different R_i).
  We assume a 1-1 correspondence between the maximal convex subintervals is available.

- $(i$ *always-*R $j)$, where R is a convex relation, iff maxconsubints of i are in each case R to the corresponding maxconsubints of j

- $(i$ *sometimes-*R $j)$, where R is a convex relation, iff some maxconsubint of i is R to the corresponding maxconsubint of j

8.2 Operators for Intervals

We attach intervals to actions, tasks, events and assertions, representing the periods of time over which an action takes place, a task is performed, an event happens, or an assertion is true. We note here that certain supposed problems with the definition of truth-on-an-interval have been adequately answered in [*Hum79*].
We also provide the correspondence between an interval and a set of convex intervals, a means of measuring the *duration* of an interval, and the length of time over which it happens, here called the *diameter*. *Duration* and *diameter* are, of course, the same for convex intervals.
In mathematics, duration is usually called *measure*, and *diameter* is terminology from topology.

8.2.1 The operators

We use the word *type* to indicate a domain of objects of the same sort. There is no implied type theory in the use of this terminology.

interval-of(P): *P is of type task / action / event / assertion*
returns i of *type interval* such that *occupies(P,i)* (see below)
Since we reason assertionally about time intervals, this gives us a way of passing between the task domain and the time domain.

dissect(i): the *set* of maximal convex subintervals of i *Dissect* is somewhat like a selector for the data domain of non-convex intervals even though it returns a *set*, not an interval.

combine(S): S is a *set* of intervals
makes an interval out of S, rather like a *union* operator. In general, this interval will be non-convex. All the intervals in S are subintervals of *combine(S)*. *Combine* is the constructor for the data domain of non-convex intervals.

duration(i): *type real*, the measure of i

convexify(startint,endint): *type interval*, the smallest convex interval containing *startint* and *endint*. Note that it follows that *convexify* is commutative and associative.

alltime: *type interval*, the global time interval that includes all time.

Note that we have omitted from the list of primitives the function

diameter(i): *type real*, the largest distance between two subintervals of i (including the duration of the subintervals)

Diameter may be defined in terms of *duration* by the equation

- *diameter(i) = duration(convexify(i,i))*

We do not include *diameter* as a primitive, since it is definable, but it is a basic part of the constraint-expression language.

Similarly, *duration* need only be defined for *convex* intervals, since its extension to non-convex intervals follows from the additive property below.

8.2.2 Axioms

In this section, we give the axioms that specify the operators described above. We note that the only unbounded quantifiers appearing in the axioms are universal, and that the bounded existential quantifiers that appear are restricted to range over sets or intervals that are parameters in the formula. We envisage that these objects will be finitely bounded in most applications, in such a way that the quantifiers may be realised by an enumeration. This is indeed the case for $REFINE^{TM}$.

Note *(i convex)* and *(convex i)* are both shorthand for *i bars i* (chapter 5).

- $(\forall i)(i \ll alltime \vee i = alltime)$ characterises *alltime* as the global time interval. All intervals are *contained-in* or *equal-to alltime*

The next three axioms characterise *dissect(i)*.

- $(\forall\, convex\ j \ll i)(\exists k \in dissect(i))(j \ll k \vee j = k)$
- $(\forall j \in dissect(i))(j\ convex \wedge (j \ll i \vee j = i))$
- $(\forall j)(\forall k \in dissect(i))(j \ll k \Rightarrow \neg\ j \in dissect(i))$

The first axiom states that all convex subintervals of i are contained in some interval k in *dissect(i)*, or are equal to such an interval. The second ensures that *dissect(i)* contains only convex subintervals of i, which in the presence of the first ensures that *dissect(i)* contains at least the maxconsubints of i. The third axiom is not in positive Horn clause form, since the consequent is negated. It may easily be turned into the right form by observing that the negation of the antecedent is equivalent to a positive disjunction of the other twelve interval relations enumerated by Allen, who gave the exhaustive list of possible relations between convex intervals. This observation allows us to take the contrapositive statement for our axiom. This has the correct form, even though its intent is more obscure. Our relations include those that are the disjunction of convex interval relations, so this disjunction reduces to a single predication in the consequent.

- $(\forall j)(\forall k \in dissect(i))((j \in dissect(i)) \Rightarrow (j\ (\| \vee \oslash \vee .....)\ k))$

In the presence of the first two axioms for *dissect*, the third *dissect* axiom ensures that *dissect(i)* contains *only* maxconsubints of i.

The next three axioms characterise *combine*. The first says that if you *combine* a *dissect*ed interval, that you get back the original interval under all circumstances. The second asserts that if the set S consists of disjoint, non-*meet*ing convex intervals, then *combine* is indeed the full inverse of dissect. Note that *dissect(i)* consists only of disjoint, non-*meet*ing intervals, i.e. this is derivable from the *dissect* axioms. The third *combine* axiom ensures that *combine*-ing a set of intervals which *overlap* or *meet* does not give more than you want in the resulting interval. If there were more, then the extra would form an interval that was a subinterval of *combine(S)*, but which was disjoint from anything in S. The axiom rules this out. An alternative way of axiomatising this property is

- $(\forall i \in dissect(combine(S)))((\exists j \in S)((j\ starts\ i)\ \vee$
  $(j = i))\ \wedge\ (\exists k \in S)((k\ ends\ i)))$

This property ensures that there are no "little bits hanging off the end" of an interval in *combine(S)* that aren't there in some interval in S. Notice that every convex interval in *dissect(combine(S))* is at least as big as any convex interval in S, thus reducing the number of cases we have to worry about stating in the axiom.

The cleanest way to implement the correct *combine* function is probably to iteratively *convexify* intervals in S which *overlap* or *meet*. Call the resulting set S_1. All the intervals in S_1 are disjoint, and non-meeting, and hence *dissect(combine(*S_1*))* $= S_1$. Furthermore, *combine(*S_1*)* $=$ *combine(S)* by the axioms. However, we are only concerned here with a correct assertional specification of the functions in a limited logical language, and we have given this for *combine*. The assertions may be compiled any appropriate way.

- $combine(dissect(i)) = i$
- $(\forall i \in S)(\forall j \in S)((i \prec j \ \vee \ j \prec i \ \vee \ i = j) \wedge$
  $(i\ convex \ \wedge \ j\ convex))$
  $\Longrightarrow dissect(combine(S)) = S$
- $(\forall j)(((j \oslash combine(S)) \Rightarrow (\exists i \in S)(j \oslash i)) \ \wedge$
  $((combine(S) \oslash j) \Rightarrow (\exists i \in S)(i \oslash j)))$

Next, we have the axioms for *duration*, which specify only that duration is a fully additive function. Giving values of *duration* on the convex intervals will then specify *duration* completely. The purpose of the special axiom for the case of *meet*ing is to be able to derive different time units. For example, one can specify that there are seven *days* in a *week*, by adding a condition that seven *day*-type intervals which meet consecutively form a *week*-type interval. One can then count in *week* units or *day* units merely by adding axioms of the form

- *(i* $\in$ *DAYS* $\Longrightarrow$ *duration(i) = 1)*
- *(i* $\in$ *WEEKS* $\Longrightarrow$ *duration(i) = 1)*

as appropriate. Given the *days* definition, the specification of *week*, and the axiom, will then guarantee that the *duration* of a *week* is seven *day* units.

- *duration(i)* $= \sum$ *duration(dissect(i))*

  The sum is taken over all members of *dissect(i)*.
- $(i \parallel j) \Rightarrow duration(combine(dissect(i) \cup dissect(j))) =$
  $duration(i) + duration(j)$

Finally, the axioms for *convexify* specify that *convexify(i,j)* is the smallest convex interval containing i and j. *Convexify* is a total, and thus a commutative operation. It is also associative, and we prefer to include both these properties explicitly, even though they follow from the minimal property.

- *i (begins-at* $\vee$ *ends-at) convexify(i,j)* $\wedge$
  j (begins-at $\vee$ *ends-at) convexify(i,j)*
- *convexify(i,j) convex*
- $(\forall k\ convex)(\forall i)(\forall j)((i \ll k \wedge j \ll k) \Rightarrow$
  $(convexify(i,j) \ll k \ \vee \ convexify(i,j) = k))$
- $convexify(i,j) = convexify(j,i)$
- $convexify(i, convexify(j,k)) = convexify(convexify(i,j), k)$

  Note that the associative and commutative properties of *convexify* actually follow from its minimal property. However, any reasonable theorem prover would probably prefer to know this explicitly (as ours does).

8.2.3 Additional Noteworthy Properties

The following properties are all consequences of the axioms:

- $i\ convex \Rightarrow duration(i) = diameter(i)$

  this follows from the definition of *diameter* and the property:

- $i\ convex \Rightarrow convexify(i,i) = i$

- $i \ll j \Rightarrow duration(i) < duration(j)$

  which follows from the additive property of *duration*, given enough subintervals in the universe.

8.3 Predicates

The predicates we wish to use on intervals are specified in chapter 5.

interval relations: We include all the relations between intervals defined in chapter 5.

8.3.1 Axioms

- We include the relation product table, and the other axioms needed for specifying the algebra of relations in chapter 5. It should be noted that this table has not yet been completely worked out for the Extended Interval Calculus.

8.4 Additional Defined Entities

We need the basic (but not primitive) function *diameter* to adequately express properties of, and constraints on, non-convex intervals. We repeat here the definition and properties of *diameter*

diameter(i): *type real*, the largest distance between two subintervals of i (including the duration of the subintervals)

- $diameter(i) = duration(convexify(i,i))$

We also need, for purposes of specification, the predicates

occupies(P,i): *P is type task / action / event / assertion*
i is type interval
i is the exact interval over which *P takes place / holds / occurs / is true*

occurs-in(P,i): *true* of all i such that $i \ll interval\text{-}of(P)$

which have the properties

- *occupies(P,interval-of(P))*
- $occupies(P,i) \Longrightarrow occurs\text{-}in(P,i)$
- $occurs\text{-}in(P,i) \Longrightarrow (\exists j \ll i)(occupies(P,j))$

These predicates are provided by, and their properties follow from, the definitions

- $occupies(P,i) \Leftrightarrow i = interval\text{-}of(P)$
- $occurs\text{-}in(P,i) \Leftrightarrow interval\text{-}of(P) \ll i$

8.5 Temporal Logic

8.5.1 Interval Constants

We introduce three interval constants, which correspond to McTaggart's A-series notion of time [*McT08, Lad85*], and the standard syntax of temporal logic. The A-series notion conceives of time as consisting of the moving, changing present, the past and the future. This corresponds to the interpretation of the temporal operators in classical tense logic, except that 'present' is implicit. The semantics of tense logic, however, is similar to McTaggart's B-series, which consists of immutable points of time, like timestamps, at which there is no change. Change is represented in the B-series by moving from one point to another. The evaluation of a tense-logical formula relative to a point, which is the standard semantical definition, is similar to connecting the A-series and the B-series notions. We show how to capture the A-series notion in interval calculus. The standard time-of-day clock functions as an A-series to B-series converter.

The constants are:

now: intuitively, an interval of smallest granularity. In any practical domain of application, intervals will not be infinitely divisible. If they are, there is still no logical contradiction in the axioms presented, as can be shown by a compactness argument from model theory. In this case, *now* would function like an interval of measure 0.

future: intuitively, for those who like their intervals to contain points, the interval (now, ∞); all future time

past: intuitively, the interval $(-\infty, now)$; all past time

8.5.2 Axioms

- *now convex*
- *future convex*
- *past convex*

- $past \parallel now \parallel future$
- $(\forall\, convex\ i)(\neg\, now \ll i \Longrightarrow ((i \ll past) \vee (i \ll future)))$
- $(\forall i)(i \prec now \Longrightarrow i \ll past)$
- $(\forall i)(now \prec i \Longrightarrow i \ll future)$
- $(\forall\, convex\ i)((now \ll i) \vee (now \prec i) \vee (i \prec now) \vee$
  $(i \parallel now) \vee (now \parallel i) \vee (i = now))$

These axioms characterise the constants. They state that the three convex intervals meet in the right ways, that they include all time, and that time is linearly ordered with respect to the *now* interval.

8.5.3 The Temporal Operators

We provide here definitions of the two temporal operators $\Box$ and $\Diamond$.

The formula $\Box P$ is true if and only if P is true everywhere in the future. Similarly, $\Diamond P$ is true if and only if P is true at some evaluation object in the future. We deliberately avoid the word "point" since in our formulations, the evaluation objects are intervals, which may or may not have point-like properties depending on the application. [*Hum79*] shows how truth may be defined for tense-logical formulae evaluated on convex intervals.

The definitions of $\Box P$ and $\Diamond P$ can be

- $\Box P \equiv (int(P) = future \vee future \ll int(P))$
- $\Diamond P \equiv$
  $(int(P)\ sometimes\text{-}(overlaps \vee contained\text{-}in)\ future)$

By observing that $\Diamond P \equiv \neg\Box\neg P$, we can obtain alternative definitions of either within the interval framework.

8.5.4 Conversion Functions

Standard temporal logic has a syntax that corresponds to McTaggart's A-series time, and a semantics that corresponds to his B-series notion of time (roughly, timestamps).

We indicate that conversion by noting that it is provided already in the standard facilities available on most AI workstations, as a real-time clock, which converts *now* into a timestamp. We also need to construct *past* and *future* from the timestamp. We assume that the clock runs to a certain granularity, say microseconds, and point out that the clock does, in fact, specify a time interval, whose duration is one microsecond in this case. Essentially, the clock tells you which interval the query interrupt is contained in. In this context, there are *next* and *previous* operators, which return the next and the previous timestamps. They may be implemented by *increment* and *decrement* respectively. The representation is therefore just

timenow(): the call to the clock implements the conversion of *now* to its B-series timestamp.

If there are *infinity* intervals at both ends of the time structure, (∞ and $-\infty$ for want of better names) then *past* and *future* may both be represented by

past: *convexify($-\infty$, previous(now))*

future: *convexify(next(now), ∞)*

It is consistent to add such *infinity* intervals. If you don't want them, the properties of the past and future B-series intervals may then be inferred from the axioms alone.

8.6 Defining Time Units

We need to reason about years, months, days, minutes and microseconds. We introduce a standard form for an interval which represents an instance of one of these units. All the units will be convex intervals, and we then show how to develop the types of units from these standard forms.

8.6.1 Standard Time Units

We repeat, for the reader's convenience, our example of how we use sequences of integers to represent our standard units. We use a linear hierarchy of standard units, **year, month, day, hour, minute, second**, arranged as a sequence. We illustrate its use down to seconds, hence our sequences will have lengths of up to six elements. It should be clear that the hierarchy is easily extendable to smaller units such as microseconds. We illustrate the meanings of sequences of integers, rather than giving an obvious definition.

- **[1986]** represents the year 1986
- **[1986,3]** represents the month of March, 1986
- **[1986,3,21]** represents the day of 21st March, 1986
- **[1986,3,21,7]** represents the hour starting at 7am on 21st March, 1986
- **[1986,3,21,7,30]** represents the minute starting at 7:30am on 21st March, 1986
- **[1986,3,21,7,30,32]** represents the 33rd second of 7:30am on 21st March, 1986 (the first second starts at 0)

8.6.2 Axioms for Units

Certain relations hold between these intervals. All of these intervals are convex, and hence the vocabulary of relations is Allen's [*All85*].

We give examples only of the axioms, since the nature of the rest may easily be inferred from the examples.

It is obvious that not all integer sequences of appropriate lengths are going to name units in our formalism. We shall not bother with checking bounds on elements of a sequence, since this is a detail of no theoretical interest. We shall assume bounds are checked somehow.

We shall use **x, y, z,**..... for integer variables, and α, β,.... for sequence variables. Concatenation of sequences is denoted by $\frown$. All the axioms are quantifier-free statements.

- ([**x,1**] *starts* [**x**]) $\wedge$ ([**x,12**] *ends* [**x**])
  January is the first month and December is the last month of the year

- [**x,y,1**] *starts* [**x,y**]
  All months begin with day 1

- [**x,1,31**] *ends* [**x,1**]
  January ends on the 31st

- **x** divisible by 4 $\Rightarrow$ ([**x,2,29**] *ends* [**x,2**]) *else* ([**x,2,28**] *ends* [**x,2**])
  February has 28, or sometimes 29 days

- appropriate cases for the other months

- ([**x,y,z,0**] *begins* [**x,y,z**]) $\wedge$ ([**x,y,z,23**] *ends* [**x,y,z**])
  constraints for hours in a day

- appropriate cases for minutes and seconds

- for length(α) = 0 to 5 and **x** less than the ending numbers for the appropriate length,
  ($\alpha \frown$[**x**]) $\parallel$ ($\alpha \frown$[**x+1**])
  the **x**th year/month/.. *meets* the **x+1**st year/month/..

- for length(α) ≤ 5,
  ($\alpha \frown$[**x**]) $\ll \alpha$

We also need to be able to coalesce representations, to gather months into years, and seconds into minutes. The axiom for this is

- ($\alpha \frown \delta$ *starts* α) $\wedge$ ($\alpha \frown \gamma$ *ends* α) $\Rightarrow$
  convexify($\alpha \frown \delta$, $\alpha \frown \gamma$) = α

8.6.3 Interval Types Definable From Unit Types

We can now define classes of intervals, based upon the units.

- YEARS = { α | length(α) = 1 }
- MONTHS = { α | length(α) = 2 }
- DAYS = { α | length(α) = 3 }
- Similarly for HOURS, MINUTES, SECONDS,.....

We may also define units which are not in the list of basic units. Firstly, let us assume that all variables and sequences range over the set DAYS. This will simplify notation for our examples. We define

- $\phi_0(\alpha, \beta) \equiv (\alpha \parallel \beta)$
- $\phi_{i+1}(\alpha, \beta) \equiv (\exists\gamma)(\phi_i(\alpha, \gamma) \ \wedge \ (\gamma \parallel \beta))$ for $0 \leq i$
- ϕ^* is the symmetric, transitive closure of ϕ, for any binary relation ϕ

The ϕ_i are the iterated *meet* relations for DAYS. Note that, as we have defined them, a given α in DAYS *meets* exactly one β in DAYS, and *is-met-by* exactly one γ in DAYS.

- WEEKS =
  { *convexify*(α, β) | $\phi_5(\alpha, \beta)$ }
  defines all 7 day intervals as weeks
- MONDAYS =
  { α | $(\phi_6)^*$([**1986,3,31**], α) }
  Needless to say, [**1986,3,31**] is, in fact, a Monday

We may define other days of the week in a similar way to MONDAYS, or we may choose to use an implicit definition, such as

- (SUNDAYS $\subset$ DAYS) $\wedge$ (*combine*(SUNDAYS)
  always-meets combine(MONDAYS))

Our definitions of the interval types show that we need to maintain the distinction between a non-convex interval I and the set of its maximal convex subintervals *dissect(I)*. All of the unit classes YEARS, MONTHS, ..., as well as some of the defined classes such as WEEKS, satisfy the condition

- *combine(S) = alltime*

and cannot thus be distinguished purely as interval objects. One of the major reasons for introducing sets into the time structure must be to distinguish between the different classes of time units. Since the set theory is needed, we see no reason not to make cautious use of it, and we may then avoid the need for a proliferation of interval operations, since we may use *dissect* on an interval, perform set theoretic operations, and then use *combine* to create the new interval.

Chapter 9

A Sample Implementation

9.1 Summary

We include a sample of code written in *REFINE*, that implements part of the *TUS*. Additionally, we introduce the concept of *weighted interval*, which we are intending to use in investigation of synthesis of scheduling algorithms. In this example we indicate how weighted intervals may be used.

The sample code is being used in essentially this question/answer form as a training exercise for novice users of the *REFINE* system.

We emphasise that the solutions have been compiled in *REFINE* and checked for correctness. They are not merely paper solutions.

9.1.1 *REFINE* syntax

The syntax of the definitions we use in *REFINE* is fairly straightforward. We explain sufficient for the reader to understand the code samples. We do not use formal syntax for the definition, since it is not our intent to give a *REFINE* language definition. The manual contains the full language and system definition *[Ref86]*.

REFINE syntax incorporates some similarities to *Lisp*, in particular the Cambridge Polish style of writing functions applied to arguments, with the wrinkle that the arguments are separated by commas.

As in a *Lisp* system, when a *REFINE* object (function, type, whatever) is compiled, it stays in the system, and may be called at any time. Thus complicated objects may be built up from a series of smaller objects, as we do in the example below.

A *REFINE* function is defined by a statement of the form:

`(Defobject` **name** `#t vfunction` (**argument-types**):**value-type body**)

The **name** is the name chosen for the function.

The **argument-types** have the form

parameter : type

where **type** is one of the basic data types, or is a user-defined abbreviation for one of these.

There are as many argument types, separated by commas, as there are arguments to the function.

The **parameter** may be used in the **body**, and is understood to be local to the body.

The **value-type** is of form **type**.

The **body** is of two different forms in our functions. There is a standard form, which gives the value directly in Cambridge Polish syntax, except that the arguments (but not the function name) are separated by commas from each other; and a `computed-using` form.

The `computed-using` form contains the keyword `computed-using`, followed by an arbitrary number of clauses of the form

hypothesis => name = expression

where **hypothesis** may be any legitimate predicate, **name** is the name of the function being defined, and **expression** is some expression denoting a value, which may be a *REFINE* function applied to an argument (in Cambridge Polish), or may be a parameter defined elsewhere in the `Defobject`. The clauses are separated by commas in the body.

The hypotheses are tested for truth, and the function is assigned the value on the right-hand side of the => corresponding to the correct hypothesis.

A *REFINE* user-defined type may be defined by a statement of the form

`(Defobject` **name** `#t type` = **type-former**)

The **type-former** parameter allows the user to define a shorthand form **name** for the generally much more complex form **type-former**.

An example of this is

```
(Defobject doodle #t type =
  (set integer)    )
```

which defines the type `doodle` to be the type of sets of integers.

For another example,

```
(Defobject dawdle #t type =
  (mapping integer -> boolean)    )
```

defines the type `dawdle` to be the type of mappings from integers to booleans. The type of `mappings` in *REFINE* is the same as the type of sets of ordered-pairs (`cart-prods`) of argument-values of the appropriate types. Thus the type system of *REFINE* is extensional in this sense.

We present an example of this ability to define a mapping by argument-value pairs, using a generic object called a `v-form`. It is unimportant for our purposes to know what a `v-form` is supposed to be. We illustrate also by this example the ability to initialise an object, to assign it certain values, explicitly in the definition. We define first a map from integers to sets of integers, and initialise it to the `emptymap` We do this for the object `gloo` by means of the following kind of statement

```
(defobject gloo #t var:  (mapping integer -> (set integer)
                = emptymap)
```

We may now give this mapping a set of argument-value pairs by means of the following statement, which defines the value of `gloo` on the integer `1` to be the set { `2, 3, 4` }, and on the integer `3` to be the set { `1, 3, 5` } . We give this setting-object the name of `glee` - everything has a name.

```
(defobject glee #t v-form
   (gloo 1) { 2, 3, 4 }
   (gloo 3) { 1, 3, 5 }
)
```

The *REFINE* compiler type-checks all expressions. This allows certain forms such as the `computed-using` far more useful than it appears in the example above. The type checker catches most of the errors this user seems to make. Most of the functions work correctly when they eventually pass the type checker.

Consider the following sample *REFINE* definition. We reserve discussion of the data types in *REFINE* for below, so the reader may ignore for this example what operators make sense for the `sequence` data type that the argument of the function refers to.

```
(Defobject glob #t vfunction (a:  (seq integer)):integer
   ( if (size a) ≥ 3 then (a 3) )
)
```

This defines a function named `glob`, which takes a single argument of type `(seq integer)` (which is the type of a sequence of integers), and returns a value of type `integer`. The argument is referred to by the name `a` in the body. The argument has an attribute of `size` (the length of the sequence). If the `size` of `a` is greater than or equal to 3, then the value of `glob` is `(a 3)`, which is the third element of the sequence `a`, since sequences in *REFINE* behave the same as finite functions from positive integers to the domain type.

REFINE also uses special symbols for logic and set theoretic operators. For example, conjunction of predicates or sentences is represented by $\wedge$, disjunction by $\vee$. *REFINE* uses the relations $\leq$ and $\geq$, as well as $<$ and $>$. In the `set` data type, `union` is $\cup$, `intersection` is $\cap$, `membership` is $\in$, and `inclusion` is $\subseteq$.

The Pattern Matcher

The pattern-matching feature of *REFINE* allows natural mathematical-type definitions, and we use this feature in some of our code. An example should make it clear how the pattern-matcher works.

Consider the following definition:

```
(Defobject year-of #t vfunction (a:(seq integer)):integer
  computed-using
     (size a) ≤ 3 ∧ a = [year, ..]  => (year-of a) = year
)
```

This defines a function called `year-of` that takes a single argument of type `time-unit` (which is a user-defined type) and returns an `integer`. The argument to the function is denoted by `a` in the body. `a` is, in fact, a sequence, since `time-unit`s are sequences. Sequences have a `size`, that is, number of elements. If the `size` of `a` is ≤ 3, and `a` starts with some integer (the type is inferred) to which we give the name `year`, and we don't care about the rest (the '..' matches an arbitrary pattern within the sequence, i.e. the rest of the elements of the sequence), then the value of `year-of` is defined to be this integer named by `year`.

As an example of the further capabilities of the pattern-matcher, consider the definition

```
(Defobject frump #t vfunction
          (a:(seq integer), b:(seq integer)):boolean
  computed-using
     a = [$x, va, ..]  ∧ b = [$x, vb, ..]  ∧ va ≠ vb
        => (frump a, b) = (va < vb)
)
```

This function `frump` takes two integer sequence arguments, and returns a `boolean` value. Hence, it is a binary predicate of integer sequences. The words $\$x$ match an arbitrary part of the sequences `a` and `b`, but the arbitrary part has to be the same in both instances. The next element in the sequence is `va`, respectively `vb`, and it is part of the hypothesis that `va` $\neq$ `vb`. So the first two conjuncts in the hypothesis say:

- Suppose `a` and `b` start off with the same initial segment and then differ at some position, and call the values at this position `va`, respectively `vb`.

Then the conclusion says that the value of `frump` is the same as the truth value of the sentence `(va` $<$ `vb)`, i.e. `true` if `va` $<$ `vb`, and `false` if not. In other words, `frump` looks down its two arguments until the first place they differ, and then returns `true` iff the first argument is $<$ the second at this position. Thus `frump` is the *lexicographic ordering* of its arguments.

These two examples demonstrate all that we shall use from the pattern language capability of *REFINE*.

Data Types

REFINE allows users to build their own data types, from a basic collection that includes `set` and `sequence`, as well as the more standard types of `integer`, `boolean`, `real`.

The `set` data type includes the standard operators, with the standard notation, as mentioned previously; i.e. `union` is $\cup$, `intersection` is $\cap$, `membership` is $\in$, and `inclusion` is $\subseteq$. `Set`s may be explicitly formulated, as in mathematics, by { `3, 2, 1` } , which gives the set of the integers 1, 2, and 3; or by the usual set abstraction operator, with the usual syntax, except that the names that are bound within the operator are explicitly noted, in parens, immediately after the semi-colon, e.g. { `x : (x) x` $\in$ `Y` $\wedge$ `x` $\geq$ `3` } , which returns the set of things in the `set y` that are ≥ 3. The extra syntax in this expression is the word '`(x)`', which is the binding indicator mentioned.

Predicates are simply `vfunction`s with values of type `boolean`.

The type `sequence` has the explicit form `[3, 2, 1]`, for the sequence of integers of length 3 whose first element is 3, second is 2, and last is 1. There are abstract sequence formers, similar to set abstraction above, but since we don't use it below, we shall not give the syntax. The sequence type has a function `size` returning the length of the sequence, and the sequence may be regarded as a function from a finite initial segment of the positive integers to values from a single data type. So the third element of the sequence `a` would be denoted by `(a 3)`, and the first element by `(a 1)`.

The data type of sequences from type `tipe` is denoted by `(seq tipe)`, and the data type of sets over tipe is denoted by `(set tipe)`. We also have the type of mappings from `tipe` to `teep`, which is written as `(mapping tipe -> teep)`, as we have seen. The mappings are extensional in the sense that they are the same type as the set type of their argument-value pairs.

We shall also need the `cart-prod` type. A `cart-prod` is the cartesian product type of two possibly different data types. Except for the heterogeneity, it would be similar to an ordered pair, or a sequence of length 2.

The definition

```
(Defobject weight #t type =
   (cart-prod the-unit # time-unit, the-weight # integer)
)
```

defines a `cart-prod` whose first component is a `time-unit` (that is, a sequence of `integer`, as we shall see), and whose second component is an `integer`.

Suppose `aa` was an element of this type. Then the `time-unit` coordinate would be referred to by the expression `aa↑the-unit`, and the `integer` element by the expression `aa↑the-weight`.

Supposing `blob` were a `time-unit`, and `blip` were an `integer`, then the object which was of the `cart-prod` type with these two component types would be denoted by

$$\langle \texttt{blob}, \texttt{blip} \rangle$$

9.2 The Code Examples

9.2.1 Definition of the Time Units.

We define a basic time unit to be a sequence of integers, of length one, two, or three, representing

```
[Year, Month, Day]
```

so, for example, the full year 1986 will be represented as the sequence

```
[1986]
```

the month of March, 1986 will be represented as

```
[1986, 3]
```

and the 14th of March, 1986 will be represented as

```
[1986, 3, 14]
```

NOTE: For the sake of this series of exercises, we assume all twelve months have exactly 30 days in them.

9.2.2 *Time-Unit* as a Type

We define and compile a type `time-unit` that will let us represent units as sequences of integers.

```
(Defobject time-unit #t type = (seq integer) )
```

We need functions `year-of`, `month-of`, and `day-of` that apply to sequences of appropriate lengths that tell you what the year, month, and day are of a sequence.

It doesn't matter that the operations apply to inappropriate sequences. However, the solutions should not return a defined value on such sequences.

We may define `year-of` as:

```
(Defobject year-of #t vfunction (a:time-unit):integer
  (if (size a) ≤ 3 ) then (a 1)
)
```

There is another way of defining functions by conditions, which is more declarative, and possibly more elegant. We give an example of the same function `year-of` written using this declarative style:

```
(Defobject year-of #t vfunction (a:time-unit):integer
  computed-using
    (size a) ≥ 1 ∧ (size a) ≤ 3 => (year-of a) = (a 1)
)
```

Using this declarative style, there is also a more picturesque way of defining the same function, using the pattern language:

```
(Defobject year-of #t vfunction (a:time-unit):integer
  computed-using
    (size a) ≤ 3 ∧ a = [year, ..]  => (year-of a) = year
)
```

The full collection of definitions is:

```
(Defobject year-of #t vfunction (a:time-unit):integer
  ( if (size a) ≥ 1 ∧ (size a) ≤ 3 then (a 1) )
)

(Defobject year-of #t vfunction (a:time-unit):integer
  computed-using
    (size a) ≥ 1 ∧ (size a) ≤ 3 => (year-of a) = (a 1)
)

(Defobject year-of #t vfunction (a:time-unit):integer
  computed-using
    (size a) ≤ 3 ∧ a = [year, ..]  => (year-of a) = year
)

(Defobject month-of #t vfunction (a:time-unit):integer
  ( if (size a) ≥ 1 ∧ (size a) ≤ 3 then (a 2) )
)
```

```
(Defobject month-of #t vfunction (a:time-unit):integer
  computed-using
     (size a) ≥ 1 ∧ (size a) ≤ 3 => (month-of a) = (a 2)
)

(Defobject month-of #t vfunction (a:time-unit):integer
  computed-using
     (size a) ≤ 3 ∧ a = [year,month, ..]
          => (month-of a) = month
)

(Defobject day-of #t vfunction (a:time-unit):integer
  computed-using
     (size a) ≥ 1 ∧ (size a) ≤ 3 => (day-of a) = (a 3)
)

(Defobject day-of #t vfunction (a:time-unit):integer
   ( if (size a) ≥ 1 ∧ (size a) ≤ 3 then (a 3) )
)

(Defobject day-of #t vfunction (a:time-unit):integer
  computed-using
     a = [year,month,day] => (day-of a) = day
)
```

9.2.3 Legitimate Time Periods

We define a predicate called `legitp` that takes a sequence of integers, and returns `true` if the sequence represents some actual unit of time between the years 1000 and 2000 inclusive, and false otherwise.

For example,

`(legitp [1983, 2, 13])` returns `true`
`(legitp [1983, 2, 37])` returns `false`
`(legitp [2001, 2, 13])` returns `false`
`(legitp [1983, 22, 13])` returns `false`

Our convention is that all months have exactly 30 days in them, so

`(legitp [1983, 2, 30])` returns `true`, and
`(legitp [1983, 1, 31])` returns `false`

We find it convenient to use logic formulas in the definition of `legitp`, since the value of `legitp` is boolean. An example of a formula that we might use is

```
(year-of a) ∈ [1000 ..  2000]
```

which is true when the first component of the sequence is in the correct range. If we define formulas for the month and day components similarly, and conjoin them, the resulting formula will be true only on legitimate sequences, providing they are of the appropriate size. The straightforward solution is for `legitp` to case split on the ranges of the components.

```
(Defobject legitp #t vfunction (a:time-unit):boolean
  (size a) ≤ 3 ∧
  (year-of a) ∈ [1000 ..  2000] ∧
  (month-of a) ∈ [1 ..  12] ∧
  (day-of a) ∈ [1 ..  30]
)
```

9.2.4 Defining The Ends of Periods

We define a function `beginning-of` that takes a time unit (a legitimate sequence of integers) and returns the first day in that time unit.

We also define a function `ending-of` that takes a time unit (a legitimate sequence of integers) and returns the last day in that time unit.

```
(Defobject beginning-of #t vfunction (a:time-unit):time-unit
  computed-using
     a = [year] => (beginning-of a) = [year, 1, 1],
     a = [year, month] => (beginning-of a) = [year, month, 1],
     a = [year, month, day] => (beginning-of a) = a
)

(Defobject ending-of #t vfunction (a:time-unit):time-unit
  computed-using
     a = [year] => (ending-of a) = [year, 12, 30],
     a = [year, month] => (ending-of a) = [year, month, 30],
     a = [year, month, day] => (ending-of a) = a
)
```

9.2.5 Ordering the Units

We shall need to define an ordering on sequences, so that we know which days start before which other days. We adopt the convention that, in the case that the time units given are months or years, we compare the first days.

We define the ordering predicate `unit-less-thanp` on sequences.

`unit-less-thanp` should look at two sequences to find the first component on which they are different, and then order the sequences depending on the order of these two different components.

e.g.

- `[1986,2]` is `unit-less-thanp [1987,1]` because they differ on the first component, and 1986 is less than 1987;
- `[1986, 2]` is `unit-less-thanp [1986, 4]` because they differ on the second component, and 2 is less than 4;
- `[1986, 2]` is `unit-less-thanp [1986, 3, 22]` because they differ on the second component and 2 is less than 3;
- `[1986, 2]` is `unit-less-thanp [1986, 2, 22]` because the first day of `[1986, 2]` is `[1986,2,1]`, which is earlier than `[1986, 2, 22]`.
- `[1986, 3]` is not `unit-less-thanp [1986, 2, 22]` because the first day of `[1986, 3]` is `[1986,3,1]`, which is later than `[1986, 2, 22]`.

It is easier to define a predicate `day-less-thanp` first, which just compares units which are both days, and then define `unit-less-thanp` from this.

```
(Defobject day-less-thanp #t vfunction
          (a:time-unit, b:time-unit):boolean
  computed-using
     a = [year1, month1, day1] ∧ b = [year2, month2, day2]
       => (day-less-thanp a, b) =
            ( (year1 < year2) ∨
            ( (year1 = year2) ∧ (month1 < month2) ) ∨
            ( (year1 = year2) ∧ (month1 = month2)
                 ∧ (day1 < day2) ) )
)

(Defobject unit-less-thanp #t vfunction
          (a:time-unit, b:time-unit):boolean
  (day-less-thanp (beginning-of a), (beginning-of b) )
)
```

The functions can be defined much more easily using the pattern language of *REFINE*, which we introduced earlier. The function definition using the pattern language is

```
(Defobject alt-unit-less-thanp #t vfunction
          (a:(seq integer), b:(seq integer)):boolean
  computed-using
     a = [$x, va, ..]  ∧ b = [$x, vb, ..]  ∧ va ≠ vb
       => (alt-unit-less-thanp a, b) = (va < vb)
)
```

We call the predicate `alt-unit-less-thanp` to distinguish it from the predicate we have just defined. It does the same job.

9.3 Comparing Units in the Order

We define the predicate `unit-lesseqp` which tests if a unit is `unit-less-thanp` or `equal` to another unit.

```
(Defobject unit-lesseqp #t vfunction
                  (a:time-unit, b:time-unit):boolean
        a = b ∨ (unit-less-thanp a, b)
)
```

We define the functions `first-of` and `last-of`, which take two time unit arguments, and return the unit that starts first, respectively, ends later.

```
(Defobject first-of #t vfunction
                  (a:time-unit, b:time-unit):time-unit
   ( if (unit-less-thanp (beginning-of a), (beginning-of b))
     then a else b)
)

(Defobject last-of #t vfunction
          (a:time-unit, b:time-unit):time-unit
   ( if (unit-less-thanp (ending-of a), (ending-of b))
     then b else a)
)
```

9.4 Definition of Convex Time Periods

We define a period of time to be the whole interval of time from one time unit to another, and we want to represent these by pairs of time units, one representing the beginning of the time period, and one representing the end of the time period.

We define a type named `period` that will be the type of periods. The first component of a `period` should be `first-unit` and the last component should be called `last-unit`. The obvious type to use is a `cart-prod`.

```
(Defobject period #t type =
   (cart-prod first-unit # time-unit, last-unit # time-unit)
)
```

We define a function `periodify` that takes two arbitrary time units and returns the period that is spanned by those two time units. The period will be a cartesian product of the first day unit and the last day unit in the period.

For example, `(periodify [1986, 3, 2], [1986, 4, 1])` returns

$$\langle$$ `[1986, 3, 2], [1986, 4, 1]` $\rangle$

`(periodify [1986], [1987, 4, 30])` returns

$$\langle$$ `[1986, 1, 1], [1987, 4, 30]` $\rangle$

`(periodify [1987, 4, 30], [1986])` also returns

$$\langle$$ `[1986, 1, 1], [1987, 4, 30]` $\rangle$

`(periodify [1986], [1986, 4, 30])` returns

$$\langle$$ `[1986, 1, 1], [1986, 12, 30]` $\rangle$

because `[1986]` includes `[1986,4,30]` as a part of it.

The function definition is

```
(Defobject periodify #t vfunction
          (a:time-unit, b:time-unit):period
  computed-using
   (periodify a,b) =
    < (beginning-of (first-of a,b)), (ending-of (last-of a,b)) >
)
```

Notice that the `period` (periodify [1986,2], [1986,2]) is the same as other periods such as (periodify [1986,2,1], [1986,2,30]) and (periodify [1986,2], [1986,2,20]). Thus periods can be the same, even though the time units they were defined from are different.

It's also true that the *period* `(periodify [1986,2], [1986,2])` and the *unit* `[1986,2]` are different objects, because they have different types. We shall be working with periods rather than with units for much of the remainder of this example.

We define the functions `starting-unit-of` and `ending-unit-of` that return the value of the first day-unit that starts a period, and the last day-unit that ends the period, respectively.

```
(Defobject starting-unit-of #t vfunction
          (a:period):time-unit
  (beginning-of (first-of a↑first-unit, a↑last-unit))
)

(Defobject ending-unit-of #t vfunction
          (a:period):time-unit
  (ending-of (last-of a↑first-unit, a↑last-unit))
)
```

We define the functions `start-of` and `end-of` for periods, which do the same for periods as `beginning-of` and `ending-of` do for units. The functions for periods should return periods.

We can't tell, merely from the order of arguments of a period, which argument starts first. For example

(periodify [1986,2,1], [1986,2,30])

and

(periodify [1986,2,30], [1986,2,1])

are the same object of type `time-unit`.

The function definitions are

```
(Defobject start-of #t vfunction
          (a:period):period
  (periodify (starting-unit-of a), (starting-unit-of a))
)

(Defobject end-of #t vfunction
          (a:period):period
  (periodify (ending-unit-of a), (ending-unit-of a))
)
```

9.4.1 Overlapping Time Units

We define the predicate `overlapsp`, the Allen relation of *overlaps*, which is true of two periods P and Q just in case P starts before Q, but doesn't end until Q has already started, and Q ends after P has finished.

Note that if P ends on the same day that Q starts, the predicate should be true of P and Q, since they have this day in common.

The predicate should return `true` on the pairs

```
(periodify [1986], [1986])
     with
(periodify [1986,3], [1987]) ,

(periodify [1986,2], [1986,3])
     with
(periodify [1986,3], [1986,4,22])

  and
(periodify [1986,2,1], [1986,3,22])
     with
(periodify [1986,3,22], [1986,4,22]) ;
```

and it should return `false` on the pairs

```
(periodify [1986,2], [1986,3])
     with
(periodify [1986,4], [1986,5,22]) ,


(periodify [1986,2], [1986,2])
     with
(periodify [1986,2,22], [1986,2,30]) ,


(periodify [1986,2], [1986,2])
     with
(periodify [1986,2,1], [1986,2,22])

   and
(periodify [1986,3], [1986,4,22])
     with
(periodify [1986,2], [1986,3]) .
```

The function definition is

```
(Defobject overlapsp #t vfunction
          (a:period, b:period) :boolean
 ( (unit-less-thanp (starting-unit-of a) (starting-unit-of b)) ∧
     (unit-lesseqp (starting-unit-of b) (ending-unit-of a)) ∧
     (unit-less-thanp (ending-unit-of a) (ending-unit-of b)) )
)
```

9.4.2 Definition of Weighted-Intervals

We now want to use periods to define a more general data type which can be used for project management.

The idea is that with a period we can associate a number. If the period represents the availability of an individual to work on a task, then the number might represent the fraction of time that the person is available.

For example, Bill might be available for the month of March 1986, to work 50 percent time. We can associate the number 0.5 with the period `(periodify [1986,3], [1986,3])` to represent this fact.

However, Bill might not be available for this amount of time for the month of April. Say he can work 75 percent time for April. Then we can associate the number 0.75 with the period `(periodify [1986,4], [1986,4])` to represent the time Bill can work in April.

How do we represent the total amount of time that Bill can work?
We have two periods, with two different weights.
We also have two options as to how we define this object.

Option (1) : We can define a type `weighted-period`, which associates a real number with a period. We can represent this association as an ordered pair - a cart-prod - of the period with the number.

We can then define the type `weighted-period-set` as a `set` of weighted-periods. Then Bill's availability would be represented by the set

$$\{< March 1986, 0.5 >, < April 1986, 0.75 >\}$$

The type definitions are

```
(Defobject weighted-period #t type =
          (cart-prod argument # period, weight # real)
)

(Defobject weighted-period-set #t type = (set weighted-period) )
```

We define two weighted periods that represent the two components of Bill's map.

```
(defobject BILLS-FIRST #t var:weighted-period
  = < (periodify [1986,3], [1986,3]), 0.5 > )

(defobject BILLS-SECOND #t var:weighted-period
  = < (periodify [1986,4], [1986,4]), 0.75 > )
```

We now define the weighted-period-set that consists of the two components of Bill that we have just defined.

```
(defobject BILL-SET #t var:weighted-period-set
  = { bills-first, bills-second } )
```

Now we pursue the second option.

Option (2) : We can define `weighted-interval` as being a `map` from periods to real numbers.

```
(Defobject weighted-interval #t type = (mapping period -> real) )
```

We define a map that represents Bill's availability. We have to define the map, and make the assignments in two separate `Defobjects`.

```
(defobject BILL #t var:  weighted-interval = emptymap)

(defobject SET-BILL-MAP #t v-form
  (Bill (periodify [1986,3], [1986,3])) 0.5;
  (Bill (periodify [1986,4], [1986,4])) 0.75)
```

We now have Bill's availability represented in two ways - one as a set of pairs, and one as a mapping. There is a correspondence between the two. We may define a mapping from a set of pairs, providing the set is 'functional', that is, there are no pairs in the set with the same first elements but different second elements.

In *REFINE*, we can define the mapping from a functional relation using the following form, which is quite similar to mathematical notation, with the exception of the word `mapsto` instead of the symbol $\mapsto$.

```
{ a mapsto b:  (a,b) <a,b> ∈ Q }
```

We define the mapping that corresponds to the pairs in the weighted-period-set we defined earlier. We use the general map former as above.

```
(Defobject BILL-ALTERNATE #t var:weighted-interval
  = { a mapsto b:  (a,b) <a,b> ∈ bill-set } )
```

We can check that the definitions of BILL and BILL-ALTERNATE give exactly the same function, by evaluating

```
(bill = bill-alternate)
```

9.4.3 Cleaning Up the Definitions

For convenience, we need a function `make-period` so that

`(make-period a) = (periodify a,a).`

The following function will suffice:

```
(Defobject make-period #t vfunction (a:time-unit):period
  (periodify a,a)
)
```

In order to accomplish some of the schedule specification tasks, we need some auxiliary functions.

The first two are `predecessor` and `successor` functions on time units. Suppose we have a time unit, and we want to find the day BEFORE it starts, and the day AFTER it ends. We call the former the `predecessor`, and the latter the `successor` of the unit.

Some examples:

- the predecessor of `[1986,3,4]` is `[1986,3,3]`,
- the predecessor of `[1986,3,1]` is `[1986,2,30]`,
- the predecessor of `[1986,1,1]` is `[1985,12,30]`,
- the predecessor of `[1986]` is `[1985,12,30]`,
- the predecessor of `[1986,2]` is `[1985,1,30]`,
- the successor of `[1986]` is `[1987,1,1]`,
- the successor of `[1986,2]` is `[1986,3,1]`,
- the successor of `[1986,2,4]` is `[1986,2,5]`

```
(Defobject successor #t vfunction (a:time-unit):time-unit
  computed-using
     (size a) < 3 => (successor a) = (successor (ending-of a)),
     a = [year,12,30] => (successor a) = [year+1,1,1],
     a = [year,month,30] ∧ month ∈ [1 ..  11]
       => (successor a) = [year,month+1,1],
     a = [year, month, day] ∧ day ∈ [1 ..  29]
       => (successor a) = [year,month,day+1]
)

(Defobject predecessor #t vfunction (a:time-unit):time-unit
  computed-using
     (size a) < 3
       => (predecessor a) = (predecessor (beginning-of a)),
     a = [year,1,1] => (predecessor a) = [(year - 1),12,30],
     a = [year,month,1] ∧ month ∈ [2 ..  12]
       => (predecessor a) = [year,(month - 1),30],
     a = [year, month, day] ∧ day ∈ [2 ..  30]
       => (predecessor a) = [year,month,(day - 1)]
)
```

Now we have defined `predecessor` and `successor` for time units, we may use them to define and compile `next-period` and `previous-period` for periods.

```
(Defobject next-period #t vfunction (a:period):period
  (make-period (successor (ending-unit-of a)))
)

(Defobject previous-period #t vfunction (a:period):period
  (make-period (predecessor (starting-unit-of a)))
)
```

We shall need an operator that takes two overlapping periods, and splits them into three - the starting part, that belongs to the first interval, the overlapping part that belongs to both, and the final part that belongs only to the second interval.

So, if we split up the periods

```
(periodify [1986,2,2], [1986,3,24]) and
  (periodify [1986,3,12], [1986,4,2]),
```

we get the three periods

```
  (periodify [1986,2,2], [1986,3,11]),
(periodify [1986,3,12], [1986,3,24]) and
  (periodify [1986,3,25], [1986,4,2]).
```

We define a function `combine-periods` that takes two periods, and returns the set of periods consisting of the split illustrated above. If the periods do not overlap, then the function should return the empty-set.

`combine-periods` should work no matter which order the arguments are given to it (i.e. it should be *commutative*):

```
(combine-periods a,b) = (combine-periods b,a)
```

The function is

```
(Defobject combine-periods #t vfunction
             (a:period, b:period):(set period)
  computed-using
     (overlapsp b,a)
          => (combine-periods a,b) = (combine-periods b,a),
     ¬ ( (overlapsp b,a) ∨ (overlapsp a,b) )
          => (combine-periods a,b) = {},
     (overlapsp a,b) =>
       (combine-periods a,b) =
{
(periodify (starting-unit-of a), (predecessor (starting-unit-of b))),
   (periodify (starting-unit-of b), (ending-unit-of a)) ,
   (periodify (successor (ending-unit-of a)), (ending-unit-of b))
}
)
```

Notice the use of the `set` parentheses in the body of `combine-periods`.

We need to extend the operation of the function `combine-periods` so that it will perform the same splitting on `weighted-periods`, but with a twist. We want the weight of the overlapping part to be the *sum of* the weights on the original periods.

For example, `combine-weighted-periods` applied to

```
< (periodify [1986,2,2], [1986,3,24]) , 0.25 >
                    and
< (periodify [1986,3,12], [1986,4,2]) , 0.3 >
```

gives the set of periods

```
{

  < (periodify [1986,2,2], [1986,3,11]) , 0.25 >,
  < (periodify [1986,3,12], [1986,3,24]) , 0.55 >,
  < (periodify [1986,3,25], [1986,4,2]) , 0.3 >

}
```

Notice the weight on the overlapping part, 0.55 is the sum of the weights on the argument periods, which are 0.25 and 0.3 .

We define the function `combine-weighted-periods`.

```
(Defobject combine-weighted-periods #t vfunction
  (a:weighted-period, b:weighted-period):(set weighted-period)
  (if (overlapsp b↑argument, a↑argument )
    then (combine-weighted-periods b,a)
    else
  (if ¬ ( (overlapsp b↑argument , a↑argument) ∨
          (overlapsp a↑argument , b↑argument) )
    then {}
    else
  (if (overlapsp a↑argument , b↑argument) then
    {
      < (periodify (starting-unit-of a↑argument ),
        (predecessor (starting-unit-of b↑argument))),
          (a↑weight) >,
      < (periodify (starting-unit-of b↑argument),
        (ending-unit-of a↑argument)) ,
          (a↑weight) + (b↑weight) >,
      < (periodify (successor (ending-unit-of a↑argument)),
        (ending-unit-of b↑argument)) ,
          (b↑weight) >
    }))))
)
```

We finish the code examples here, but we indicate how to proceed to complete a simple schedule consistency checker.

The next step is to define a function `conglomerate`, which takes a set of weighted-periods, and performs the `combine-weighted-periods` operation on each pair of intervals in the set that overlap, until there are no more intervals in the set that overlap.

The `conglomerate` function may be used for consistency-checking in project management. For example, suppose that with each task and person is assigned a weighted-period-set which represents the portions of time assigned by that person to that task. Suppose also with each task is assigned a weighted-period-set which represents when the task is worked on, and the number of full-time-equivalent personnel working on it in each period. (The weights will in general be greater than **1** for this set).

We may now check the consistency of personnel assignments by taking the union of all the person-task sets associated with, say, Bill, and performing a conglomeration of those, and checking that no resulting weight on any interval is greater than **1**. A weight greater than **1** would indicate that Bill had been assigned more than 100% time over this interval.

We may check the consistency of task assignments similarly by taking the union of all the person-task sets for a fixed task, and `conglomerate`ing it. The resulting set should be exactly the weighted-interval-set associated with the task (or should be a portion of it, for a partial assignment).

Bibliography

All83 : Allen, J.F., *Maintaining Knowledge about Temporal Intervals*, Comm. A.C.M. 26 (11), November 1983, 832-843.

All84 : Allen, J.F., *Towards a General Theory of Action and Time*, Artificial Intelligence 23 (2), July 1984, 123-154.

AllKau85 : Allen, J.F. and Kautz, H., *A Model of Naive Temporal Reasoning*, in Hobbs, J.R. and Moore, R.C., editors, *Formal Theories of the Commonsense World*, Ablex 1985.

AllHay85 : Allen J.F. and Hayes, P. J., *A Commonsense Theory of Time*, Proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, California, Morgan Kaufmann 1985, 528-531.

AllHay87.1 : Allen J.F. and Hayes, P. J., *Short Time Periods*, Proceedings of the 10th International Joint Conference on Artificial Intelligence, Milano, Italy, Morgan Kaufmann 1987, 981-983.

AllHay87.2 : Allen J.F. and Hayes, P. J., *A Commonsense Theory of Time: The Longer Paper*, Technical Report, Dept. of Computer Science, University of Rochester, to appear.

Bel87 : Bell, C.E., *Representing And Reasoning With Disjunctive Temporal Constraints In A Point-Based Model*, preprint, University of Iowa, Department of Management Sciences.

BurSan81 : Burris, S., and Sankappanavar, H.P., *A Course in Universal Algebra*, Springer Verlag, 1981.

ChaKei73 : Chang, C.C., and Keisler, H.J., *Model Theory*, North-Holland, 1973.

ChiTar51 : Chin, L.H., and Tarski, A., *Distributive and Modular Laws in the Arithmetic of Relation Algebras*, Univ. Calif. Publications in Mathematics, New Series, vol 1, no. 9, 1951, 341-384.

Com84 : Comer, S.D. *Combinatorial Aspects of Relations*, Algebra Universalis 18, 1984, 77-94.

ClaEmSis86 : Clarke, E.M., Emerson, E.A. and Sistla, A.P., *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems (8), 244-263, 1986.

Dav87 : Davis, E., *Constraint Propagation With Interval Labels*, Artificial Intelligence 32, 281-331, 1987.

Dow79 : Dowty, D.R. *Word Meaning and Montague Grammar*, Reidel, 1979.

EmeHal83 : Emerson, E.A. and Halpern, J.Y., *'Sometimes' and 'Not Always' Revisited: On Branching Time Versus Linear Time*, Proceedings of the 10th ACM Principles of Programming Languages Conference, Austin, Tx, 1983.

Fra86 : Francez, N., *Fairness*, Springer Verlag 1986.

Hal74 : Halmos, P.K., *Lectures on Boolean Algebras*, Springer Verlag, 1974.

HalSho86 : Halpern, J.Y. and Shoham, Y., *A Propositional Modal Logic of Time Intervals*, Proceedings of the Symposium on Logic in Computer Science 1986, 279-292, IEEE Computer Society Press, 1986.

Ham71 : Hamblin, C.L., *Instants and Intervals*, Studium Generale (27), 1971, 127-134.

Hum79 : Humberstone, I.L., *Interval Semantics for Tense Logic: Some Remarks*, J. Philosophical Logic 8, 1979, 171-196.

ImiLip84 : Imielinski, T., and Lipski, W., Jr., *The Relational Model of Data and Cylindric Algebras*, Jour. of Computer and System Sciences 28, 1984, 80-102.

JonTar48 : Jonsson, B. and Tarski, A., *Representation Problems for Relation Algebras*, Bull. Amer. Math. Soc. 54 (1, part1), p80, Abstract 89T, 1948.

JonTar51 : Jonsson, B. and Tarski, A., *Boolean Algebras with Operators I*, American J. Mathematics (73), 1951.

JonTar52 : Jonsson, B. and Tarski, A., *Boolean Algebras with Operators II*, American J. Mathematics (74), 1952, 127-162.

Lad86.1 : Ladkin, P.B., *Time Representation: A Taxonomy of Interval Relations*, Proceedings of AAAI-86, 360-366, Morgan Kaufmann, 1986.

Lad86.2 : Ladkin, P.B., *Primitives and Units for Time Specification*, Proceedings of AAAI-86, 354-359, Morgan Kaufmann, 1986.

Lad86.3 : Ladkin, P.B., *Two Papers on Time Representation*, Kestrel Institute Research Report KES.U.86.5, 1986.

Lad87.1 : Ladkin, P.B., *Specification of Time Dependencies and Synthesis of Concurrent Processes*, Proceedings of the 9th International Conference on Software Engineering, Monterey, Ca, IEEE 1987, also available as Kestrel Institute Technical Report KES.U.87.1.

Lad87.2 : Ladkin, P.B., *The Completeness of a Natural System for Reasoning with Time Intervals*, Proceedings of the 10th International Joint Conference on Artificial Intelligence, Milano, Italy, Morgan Kaufmann 1987, 462-467, also available as Kestrel Institute Technical Report KES.U.87.5.

Lad87.3 : Ladkin, P.B., *Models of Axioms for Time Intervals*, Proceedings of AAAI-87, Morgan Kaufmann 1987, also available in a longer version as Kestrel Institute Technical Report KES.U.87.4.

Lad87.4 : Ladkin, P.B., *Deciding First-Order Statements About Time Intervals: Preliminary Report*, Kestrel Institute Technical Report KES.U.87.7.

Lad87.5 : Ladkin, P.B., *Constraint Satisfaction in Time Interval Structures I: Convex Intervals*, forthcoming Kestrel Institute Technical Report, 1987.

LadMad87.1 : Ladkin, P.B. and Maddux, R.D., *The Algebra of Convex Time Intervals*, Kestrel Institute Technical Report KES.U.87.2.

Lam80 : Lamport, L., *'Sometimes' is sometimes 'Not Never'*, Proceedings of the 7th ACM Principles of Programming Languages Conference, Las Vegas, Nv, 1980.

Lam86 : Lamport, L., *The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication* Journal of the A.C.M (33) 1986, 313-326.

Lyn50 : Lyndon, R.C., *The Representation of Relational Algebras*, Annals of Math. Ser. 2, vol 51, 1950, 707-729.

MacFre85 : Mackworth, A.K., and Freuder, E.C., *The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems*, Artificial Intelligence 25, 65-74, 1985.

Mad78 : Maddux, R.D., *Topics in Relation Algebras*, Ph. D. Thesis, University of California at Berkeley, 1978.

ManPnu81 : Manna, Z., and Pnueli, A., *Verification of Concurrent Programs: The Temporal Framework*, in Boyer, R. S., and Moore, J. S., The Correctness Problem in Computer Science, Academic Press 1981.

ManWal85 : Manna, Z. and Waldinger, R., *The Logical Basis for Computer Programming: Vol 1: Deductive Reasoning*, Addison-Wesley, 1985.

ManWol84 : Manna, Z. and Wolper, P. *Synthesis of Communicating Processes from Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems (6), 68-93, 1984.

McD82 : McDermott, D., *A Temporal Logic for Reasoning about Actions and Plans*, Cognitive Science 6 (2), April-June 1982, 71-109.

McD85 : McDermott, D., *Reasoning about Plans*, in Hobbs, J.R. and Moore, R.C., editors, *Formal Theories of the Commonsense World*, Ablex 1985.

McT08 : McTaggart, J.M.E., *The Unreality of Time*, Mind, 1908, 457-474.

Mon64 : Monk, J. D., *On Representable Relation Algebras*, Michigan Math. J. 11, 1964, 207-210.

New80 : Newton-Smith, W.H., *The Structure of Time*, Routledge Kegan Paul, 1980.

PelAll86 : Pelavin, R., and Allen, J.F., *A Formal Logic of Plans in Temporally Rich Domains*, Proceedings of the IEEE 74 (10), Oct 1986, 1364-1382.

Ref86 : *REFINE User's Manual*, Reasoning Systems Incorporated, 1986.

Rop79 : Roper, P., *Intervals and Tenses*, Journal of Philosophical Logic 9, 1980.

Rus87 : Russell, S.J., private communication.

Sik69 : Sikorski, R., *Boolean Algebras*, Springer Verlag, 1969.

Tar55 : Tarski, A., *Contributions to the Theory of Models III*, Nederl. Akad. Wetensch. Proc. Ser. A, 58 (Indag. Math. 17) 1955, 56-64.

TarGiv87 : Tarski, A., and Givant, S.R., *A Formalisation of Set Theory without Variables*, Colloquium Publications in Math 41, Amer. Math. Soc. 1987.

vBen83 : van Benthem, J.F.A.K., *The Logic of Time*, Reidel 1983.

VilKau86 : Vilain, M., and Kautz, H., *Constraint Propagation Algorithms for Temporal Reasoning*, Proceedings of AAAI-86, 377-382, Morgan Kaufmann, 1986.