The Formal Development Of Real-Time Systems: A Review

David Scholefield
Department Of Computer Science
University Of York
Heslington
York YO1 5DD

February 28, 1992

Abstract

As an increasing number of complex real-time systems are used to control safety critical operations, it becomes important to ensure the correctness of the software involved. Traditionally, *testing* has been the main method of program verification, but complete testing is not possible for large complex programs, and new verification methods must be found.

Formal development methods have the advantage of *proof techniques* which give confidence in program correctness to the software engineer. Until recently the time needed to perform rigorous proofs has been prohibitive, but now a number of formal systems have automated proof tools which dramatically reduce the development effort.

In this review we discuss a number of formal development languages for real-time systems. In particular, we categorise the languages depending on their underlying semantic models, and choose a representative sample from each category for detailed examination. Where automated support is provided we discuss the tools available , and where it is not, we suggest ways in which such support may be developed.

Contents

1 Introduction **5**
- 1.1 Scope And Aims Of The Review . . . 6
- 1.2 Formal Development Techniques . . . 6
- 1.3 The Formal Representation Of Real-Time Systems . . . 7
- 1.4 Technical Issues . . . 8
  - 1.4.1 Automated Support . . . 8
  - 1.4.2 Expressive Power Versus User-Friendliness . . . 9
  - 1.4.3 The Formal Representation Of Time . . . 10
- 1.5 The Structure Of The Review . . . 10

2 Real-Time Logics **11**
- 2.1 Introduction To Real-Time Logics . . . 11
- 2.2 Real-Time Temporal Logic . . . 12
  - 2.2.1 Overview Of RTTL . . . 12
  - 2.2.2 Discrete Event Processes . . . 12
  - 2.2.3 Extended State Machines . . . 12
  - 2.2.4 Real-Time Temporal Logic Syntax . . . 14
  - 2.2.5 Assessment Of RTTL . . . 14
- 2.3 TRIO . . . 14
  - 2.3.1 Overview Of TRIO . . . 14
  - 2.3.2 TRIO Syntax . . . 15
  - 2.3.3 Assessment Of TRIO . . . 16
- 2.4 RTL . . . 16
  - 2.4.1 Overview Of RTL . . . 16
  - 2.4.2 Actions . . . 16
  - 2.4.3 Timing Constraints . . . 17
  - 2.4.4 The RTL Logic Syntax . . . 17
  - 2.4.5 Events . . . 18
  - 2.4.6 State Predicates . . . 18
  - 2.4.7 System Specifications . . . 18
  - 2.4.8 Assessment Of RTL . . . 19
- 2.5 PCTL . . . 20
  - 2.5.1 Overview Of PCTL . . . 20
  - 2.5.2 PCTL Syntax . . . 20
  - 2.5.3 PCTL Semantics . . . 20
  - 2.5.4 Assessment Of PCTL . . . 21
- 2.6 TEMPURA . . . 21
  - 2.6.1 Overview . . . 21
  - 2.6.2 Interval Temporal Logic . . . 22
  - 2.6.3 The Language TEMPURA . . . 23
  - 2.6.4 Assessment Of TEMPURA . . . 23
- 2.7 Real-Time Logics – Comments . . . 23

3 Real-Time Process Algebra **24**
3.1 Introduction To Process Algebra . . . 24
3.2 An Introduction To OBJ1 . . . 24
3.3 Timed CCS . . . 24
3.3.1 Overview . . . 24
3.3.2 wTCCS Syntax . . . 25
3.3.3 wTCCS Semantics . . . 26
3.3.4 A Case Study In wTCCS . . . 27
3.3.5 Assessment Of wTCCS . . . 29
3.4 SCCS . . . 30
3.4.1 Overview . . . 30
3.4.2 SCCS Syntax . . . 30
3.4.3 SCCS Semantics . . . 31
3.4.4 Assessment Of SCCS . . . 32
3.5 Timed-CSP . . . 32
3.5.1 Overview Of Timed-CSP . . . 32
3.5.2 Timed-CSP Syntax . . . 32
3.5.3 Timed-CSP Semantics . . . 34
3.5.4 Assessment Of Timed-CSP . . . 34
3.6 CSR . . . 35
3.6.1 Overview Of CSR . . . 35
3.6.2 CSR Syntax . . . 35
3.6.3 CSR Semantics . . . 37
3.6.4 Assessment Of CSR . . . 37
3.7 Process Algebra – Comments . . . 38

4 Pogramming And Graphical Languages **39**
4.1 Introduction To Programming And Graphical Languages . . . 39
4.2 LUSTRE . . . 39
4.2.1 Overview . . . 39
4.2.2 Clocks and Sequence Variables . . . 39
4.2.3 LUSTRE Programs . . . 39
4.2.4 Compilation . . . 40
4.2.5 The Verification Of Properties . . . 40
4.2.6 Assessment Of LUSTRE . . . 40
4.3 RUTH . . . 41
4.3.1 Overview . . . 41
4.3.2 RUTH Programs . . . 41
4.3.3 RUTH Semantics . . . 42
4.3.4 Assessment Of RUTH . . . 43
4.4 RT-ASLAN . . . 43
4.4.1 Overview Of RT-ASLAN . . . 43
4.4.2 RT-ASLAN Specifications . . . 44
4.4.3 The Semantics Of RT-ASLAN . . . 45
4.4.4 Assessment Of RT-ASLAN . . . 45
4.5 Timed Petri-Nets . . . 45
4.5.1 Overview . . . 45

4.5.2 TB-Net syntax . . . 46
4.5.3 TB-Net Semantics . . . 46
4.5.4 Assessment Of TB-Nets . . . 46
4.6 Statecharts . . . 47
4.6.1 Overview . . . 47
4.6.2 Statecharts Syntax . . . 47
4.6.3 Statechart Semantics . . . 49
4.6.4 Assessment Of Statecharts . . . 49
4.7 Programming And Graphical Languages – Comments . . . 50

5 Current Research Projects **51**
5.1 Introduction . . . 51
5.2 SPEC . . . 51
5.3 Real-Time Computing: The Next Generation . . . 51
5.4 CoS . . . 51
5.5 TEMPURA . . . 52
5.6 ORE . . . 52
5.7 DEMON . . . 52
5.8 Edinburgh University . . . 52

6 Conclusions **53**
6.1 Summary Of Language Characteristics . . . 53
6.1.1 Real-Time Logic . . . 53
6.1.2 Real-Time Process Algebra . . . 54
6.1.3 Programming And Graphical Languages . . . 55
6.2 Problems With Existing Languages . . . 55
6.2.1 Methods . . . 55
6.2.2 Automated Support . . . 56
6.2.3 Explicit Support For Criticality And Fault Tolerance . . . 56
6.2.4 The Specification And Design Dichotomy . . . 57
6.2.5 A Summary Of Problems . . . 58
6.3 Expressibility . . . 58

7 Terminology **60**

8 Annotated Bibliography **64**
8.1 Introduction . . . 64
8.2 Bibliography . . . 64

1 Introduction

It would be over-enthusiastic to claim that formal development techniques could provide a panacea for the problems involved in developing useful computer systems. There are too many aspects which are not amenable to formal representation or reasoning. For example, it is hard to envisage a way in which the process of requirements elicitation could be totally formalised; it is true that some requirements can be denoted using a formal language, and possible inconsistencies found, but the *completeness* of the requirements with respect to the client's intentions cannot be formally proven. Also, there are too many problems inherent in using the formal techniques themselves; most formal languages have a considerably more complex notation than many programming languages, and we know that programmers often make syntactic and semantic mistakes – we should not expect software engineers and formalists to be any less prone to similar mistakes. It is also important to realise that mistakes in a specification are potentially more damaging than mistakes in an implementation, they 'filter down' the development process, potentially invalidating any work dependent upon the specification.

These problems may not be insurmountable, and persevering with the development of formal techniques could prove very rewarding. Once we have accepted that such techniques are simply *tools* (in the same way as data-flow diagrams, or entity relationship tables are) then we will have made significant progress. The idea propagated by some 'traditionalists' that the devotees of formal methods are really obsessed religious fakes, touting their all-encompassing doctrines of development, may be a damaging one for the future of software engineering. The situation is exacerbated by the tacit acceptance of many formalists of their mantle of mystique, and their reluctance to affirm that formal methods provide 'just another tool', with a necessarily limited scope of application.

To help with the problem of complexity perhaps we can learn from the experience of programming and develop automated tools which check for syntactic correctness and semantic consistency (a 'lint' for logics perhaps ?) Experience with such tools is growing and we will examine some of them in detail. Compositional proof systems also help with notational complexity by enabling the user to write specifications in a modular form and prove properties about those modules. This approach is known as *divide and conquer*.

The problems that we have mentioned above apply to the development of all classes of systems. When we come to consider the development of real-time systems in particular, we find that the number and importance of these problems are dramatically increased. Real-time systems involve the extra complexity of *temporal correctness* (i.e. the correctness of a real-time system is not only dependent upon its functionality, but its timeliness as well). This means that any formal development technique for real-time systems must be able to describe and reason about the relationship between time and computational events. This relationship may be reasoned about abstractly (in isolation from implementation details) in terms of specificational consistency, but design verification will always need to deal with implementational details such as processor speed, and scheduler policy, etc. This contradicts of one of the tenets of structured development, the separation of the implementation from any other aspect of the development process.

The problem of complexity is often highlighted by the fact that formal language developers rarely provide any *method* [1] with which to utilise their language. In a survey of formal 'methods' (of

[1] A method is a guide to using a formal language in the development of systems, it may be defined in terms of

general applicability), Strutt [Str89] asserts that this lack of method is one of the prime reasons for hostility towards formal techniques:

> "Most methods are not methods at all, but merely formal languages or notations.... The provision of a method (perhaps by integrating existing structured methods with formal languages) is essential."

In this review we discuss a number of formal languages designed to aid in the development of real-time systems. In particular, we discuss the ways in which these languages address the problems of scope, complexity, and method.

In the remainder of this section we discuss the aims and scope of the review, systems development methods in general, real-time issues such as the hardness of deadlines, safety-criticality and fault-tolerance, and the particular problems in their formal representation. We also discuss relevant theoretical issues in using these formalisms, such as expressibility, completeness, automation, and temporal models. Finally, we outline the scope and structure of the remainder of the review.

1.1 Scope And Aims Of The Review

Although we have consistently used the term 'system' in the introduction, this review is concerned with software development, not hardware. It is difficult to define where software ends and hardware begins, and with real-time systems (especially embedded control systems) complete separation is harder still. We shall not be looking at hardware at the physical level. We need to consider input and output devices but only from the point of view of their software interface characteristics.

We will be investigating existing formal specification and design languages which apply to real-time systems. We do so with a critical eye, as any potential user would, and hope that this helps accurately to summarise the usefulness of each language. We are also interested in what methods are available for each language, or how easy their development would be.

1.2 Formal Development Techniques

There is growing disagreement on what constitutes a specification or a design. In traditional development methods (such as the life cycle model suggested by Shooman [Sho83]) a specification describes *what* a system is to achieve, and a design *how* the system is to achieve it. In contrast, there are a number of formal languages, for example Morgan's programming language [Mor90], which are capable of describing a continuous range of programs, from the most abstract (what), through to executable implementations (how). Morgan, and others [Mil89] suggest that we should not make such a definite distinction between the specification and design, instead we should classify all software descriptions as programs which reside on a scale between completely abstract, and completely concrete (i.e. actual code).

The advantage of a single language to represent the program at all levels of abstraction is one of formal coherence. The correctness of a transformation between a highly abstract program and a

hueristics, or a collection of specific techniques.

functionally equivalent concrete program can be proven within the same semantic model. This reduces complexity in correctness proofs and may help with their automation.

The traditional view of separate specifications and designs can also be supported by formal languages and methods. Correctness proofs between specification and design may be more complex but separating these aspects may encourage a more disciplined approach to software production.

Because of this dichotomy, we will examine both specification and design languages.

1.3 The Formal Representation Of Real-Time Systems

The class of systems that may be called 'real-time' is a large one. It has been argued that *all* computer systems are in some way real-time: we are always concerned that results are produced, or interactions occur, within a 'reasonable' time span. Young [You82] reinforces this rather general definition with the following much quoted assertion:

> (... a real-time system is ...) any processing activity or system which has to respond to externally generated input stimuli within a finite and specified time.

The constraint "specified time" narrows the class if we consider implicit temporal constraints to be unspecified ones. For example, with a word processing program we would expect the paragraph reformat command to always take less than three seconds, is this a specified time ?

We can narrow the class by requiring that a real-time system be defined always in conjunction with a 'cost function' which relates the time of the occurrence of events with specific costs (or values) to denote the importance of meeting deadlines. Burns [Bur90] discusses these functions and presents some ideas on formally representing and reasoning about them using an event logic first suggested in [Sch90]. In many real-time systems the cost function is implicit (only influential in the formulation of requirements) but it remains an integral part of the system.

Therefore, in formally representing any real-time system, we have to consider two major issues. Firstly, we have to decide upon what structure best captures our intuition of time, and fulfills our requirements in reasoning about it. Secondly, we need to clarify what we mean by the term *computation* e.g. what are events, actions, and deadlines ? Many of the formal languages we discuss treat these issues very differently.

Some real-time systems are also fault tolerant. These systems are intended to be robust – they should recover from some faults within the system. Faulty behaviour should be considered at the specification stage of development, not left to the design stage. We need to be able to reason about the interaction between faults and normal behaviour before a design is developed.

Some real-time systems are also safety-critical so that any malfunction (caused by functional or timing errors) may injure people or damage equipment. Safety-critical systems may contain non critical components, and the ability to distinguish between levels of criticality is an important factor in finding a suitable formal representation. The cost function is certainly one way of representing criticality, and in some formal languages it is suggested that criticality may correspond to *probability* (we will examine this in more depth later).

Some real-time systems are also distributed. These systems are more complex for a number of reasons – they are necessarily concurrent, they may utilise a number of different physical or logical clocks, and they need communication mechanisms. Concurrency is perhaps the smallest problem – the formal representation of concurrency is now a well studied area [Mil80], [Hoa78]. Multiple clocks remains an area of controversy and there are a number of theoretical issues which affect the intuitive choice of temporal model for such systems. Communications mechanisms are most commonly used in concurrent systems, they have been studied in some depth and can be formally represented.

Because the temporal behaviour of a system is dependent upon the hardware utilised by the system, we need to be able to reason about resources and resource contention. In a some formal languages it is assumed that there are infinitely fast resources (processors, input and output devices etc), this is termed the *maximal parallelism* hypothesis. In real-time systems this hypothesis is not supportable.

We can therefore list a number of criteria with which we may compare formal development languages for real-time systems:

i) Can we reason about fault tolerance ?

ii) Can we reason about levels of criticality ?

iii) Can we represent concurrency ?

iv) Can we reason about communication ?

v) Can we define multiple time bases ?

vi) Can we reason about resource contention ?

vii) Can we reason about temporal behaviour ?

We will not consider *vii*) explicitly, if a language cannot reason about time in a quantitative manner, then it will not be included.

1.4 Technical Issues

1.4.1 Automated Support

In the introduction we suggested that a major problem with formal development languages was that of notational complexity. It is easy to make both syntactic and semantic mistakes when using a complex notation, and such mistakes can be catastrophic. Automated checking tools may be used but there are a number of theoretical problems which have to be resolved before these tools may be built.

Assuming that a language based on logic is used, correctness may be defined in terms of consistency, i.e. a term is correct only if it is either true or false, but not both. However, most logical systems are *undecidable* i.e. there exists no algorithm for deciding the truth of any formulae in finite time. This means that no general semantic correctness algorithm may be devised. There are currently two solutions to this problem, either the logic is restricted to a decidable subset [GMM89], or human intervention is requested by the decision algorithm [Ak86]. Semantic equivalence checking for non-logic-based languages may be an easier task. In CCS [Mil80,89] the proofs of bisimulation and divergence may easily be automated [CPS89].

Although a parser can be constructed to find syntactic mistakes, there is a potential problem with the lack of support for symbolic notations on standard computer keyboards and displays. Notations which rely on special characters or character positioning need to be translated into simpler terms, and mistakes may be made in the process. For example, TEMPURA [Mos86] has some problems with the temporal operators, which are all translated into English terms; formulae become very long and difficult to read. One solution is to define a notational standard for all symbolic representation in standard character sets for the purpose of machine entry. In electronic mail systems this is already achieved with the TEX notation [SMS90].

These problems give rise to two more criteria for comparing formal development languages:

viii) Is there support for automated syntax checking ?

ix) Is there an automated tool for semantic correctness checking, or can one be produced ?

1.4.2 Expressive Power Versus User-Friendliness

The criteria we have presented so far do not take account of the 'user-friendliness' of the formalism. This is the most difficult criteria to judge, and the final assessment of the ease of use of any language must be a subjective one.

There are a number of foundational theories, along with their notations, which are certainly powerful enough to deal with real-time system specification and design. Category Theory [Mac71] [BW90b] [RB88] provides a basis for describing logical systems, algebraic systems, domain theory, set theory and many other relevant mathematical theories. Indeed, Category Theory has been proposed as a language in which to formally define Mathematics itself! However, it is not realistic to expect software engineers to write specifications or designs in a categorical framework. The notation is extremely obscure and requires a *constructive* view of mathematics (which is untypical). Category theory is more suited to defining the theories in which specifications may be written, and in providing a framework in which to express correctness proofs of translations between theories.

We will make some comments on the user-friendliness of each formal language or method, but they should be not be considered overly important.

1.4.3 The Formal Representation Of Time

There are at least three orthogonal considerations in temporal models, linearity, discreteness, and bounds. Linear models may be represented as a single timeline, with a strict ordering between any two times; branching times may have different possible future times. Bounded times may have a first time, a final time, or both.

The choice of temporal model must be guided by the application that we are reasoning about but, there are a number of theoretical issues that may affect our model, and we discuss these below.

There is no complete axiomatisation of integers [Gab89], therefore if we wish to have a temporal model with the characteristics of the integers (sparse and unbounded) then we will have to compromise. One compromise would be to define a lower bound, and in computer systems this is not a problem – all computer programs will have a first execution step.

If multiple clocks are sparse then they should all run at fixed speeds which are integer multiples of each other. Otherwise, if a 'snapshot' is taken of the system on the tick of any one clock, then a number of other clocks may be between ticks, and the system is no longer temporally coherent.

Lamport [Lam80] examined branching time models in an attempt to establish their relevance to concurrent system specification. He decided that because the axiom that states "p is either never true, or is true in every possible future" is not valid in branching time models, such models have limited application. Lamport was later shown to be incorrect in one of his assumptions that led to this claim, but the proof that this axiom was indeed valid in branching time models demonstrated their inherent complexity. Branching time models are rarely used, and we will not review any here.

1.5 The Structure Of The Review

We look at three classes of formal development language: real-time logics, real-time process algebras, and formal programming languages and graphical systems. We examine a representative sample in each class, and compare them using the criteria we have defined above. We will also pay particular attention to the availability of methods.

A list of current research projects involved in the design of formal development languages for real-time systems is provided.

Conclusions include a table of the characteristics of the formal development languages surveyed.

A section on terminology is provided to clarify a number of terms which are used inconsistently within the literature.

Finally an annotated bibliography is presented.

2 Real-Time Logics

2.1 Introduction To Real-Time Logics

Most specification languages for real-time systems are based on logic. Real-time logics are defined either by including a temporal domain into a version of predicate logic, or adding real-time operators to a propositional logic. Ee examine five different real-time logics, RTTL, TRIO, RTL, PCTL, and TEMPURA, each of which demonstrates a different trend in the semantic model.

In RTTL [Ost89] there is both a distinguished temporal domain *and* temporal operators. This leads to a complex proof system, but also provides a very expressive syntax.

The real-time logic TRIO [GMM89] allows a choice of temporal models.

RTL [JM86] is a two layered action/event model. The distinction between instantaneous events and temporally extended actions is well defined and important.

PCTL [GL89] enables the user to reason about probability, and therefore make some distinction between soft and hard deadlines. The semantics for PCTL are defined using Markov chains.

TEMPURA [Mos86] is executable and has interval semantics (formulae are evaluated over intervals of sparse time).

2.2 Real-Time Temporal Logic

2.2.1 Overview Of RTTL

RTTL (real-time temporal logic) [Ost89] is a logic for the development of embedded control systems. Such control systems are examples of *discrete event processes*, and it is for this type of system that the method for RTTL has been designed. It extends Manna and Pnueli's temporal logic [MP83] to allow assertions about transitions in extended state machines. The logic uses *both* a temporal domain, and modal operators. The semantics of RTTL are expressed in *trajectories*, which are paths through extended state machines (ESMs). Trajectories may have fairness and justice constraints.

2.2.2 Discrete Event Processes

A discrete event process has states and events (which cause state changes). Transitions are instantaneous, events occur at discrete (sparse) time values, processes are event driven, may be non-deterministic, communicate with the environment, operate concurrently, and may have hard deadlines to meet.

Discrete event processes are widely used in the design of manufacturing control systems and they often include real-time constraints. The RTTL development method is specific to such processes and to the development of the software in the feedback loop of control systems. Ostroff claims that the method is not suitable for developing real-time scheduling software, operating systems, or communication architectures. However, there is no theoretical reason against amending the method and expanding its scope to a wider range of systems.

2.2.3 Extended State Machines

ESMs extend finite state machines, and overcome the problems of non-compositionality, sequentiality, and exponential increase in state size. ESMs can be concurrently composed, have communicating transitions, and may store data.

The following three ESMs model the behaviour of a level crossing and gate. The transitions marked by the m-label communicate a value into the variable 'x' (the communication is signified by the m-channel between the train ESM and the controller ESM). The value is a boolean flag which signifies that a train is approaching the crossing; the c-channel is used for the controller to signify

that the gate should be raised.

The three ESMs may perform transitions in parallel (synchronised by a global clock) and the like-named channels (in this example 'd') may be forced to synchronise. Each ESM has a unique variable to record its state, and a formulae of RTTL may include this variable in correctness assertions.

Transitions in ESMs have minimum and maximum delay times which define the time-span within which a transition is allowed to fire, and these are used for calculating worst case execution times by examining the maximum cumulative delays of trajectories. Transitions also have enabling conditions. Sequences of transitions form trajectories with which the semantics of RTTL are

defined.

2.2.4 Real-Time Temporal Logic Syntax

RTTL uses a distinguished temporal domain, the ESM state variables, and the set of ESM transitions to form temporal formula. These are then proven using an axiomatisation of the system's ESM trajectories.

The following example is a safety assertion in RTTL, the variables 'x1' and 'x2' hold the current state of two ESMs:

$$\Box\neg(x1 = ingate \wedge x2 = up)$$

This asserts that the gate will never be up while a train is on the crossing.

The following is a real-time response assertion, the variable 'n' represents the next transition to fire, the variable T is the temporal domain:

$$(n = a \wedge t = T) \rightarrow \diamond(n = b \wedge t \leq T + 10)$$

which asserts that if the next transition is 'a1' then within 10 time units, transition 'b' will be ready to fire.

2.2.5 Assessment Of RTTL

RTTL has a complex and non-compositional proof system, all of the ESMs have to be designed before any theorems may be proved about them. There is a decision procedure for finite ESMs but due to the undecidability of predicate logic, a procedure for infinite ESMs can never be found. There is a method for RTTL but it is basic, and contains informal steps. Time is global and there is no means for incorporating multiple clocks. There is no syntax for describing criticality or faults, and a maximum parallelism model is assumed.

This assessment paints a somewhat gloomy picture for RTTL, but in general the logic meets its purpose well. Embedded control systems are usually quite small – so the lack of compositionality is not very important. The undecidability of predicate logic is not a crucial problem, it can be overcome by a number of techniques, including manual intervention, or limiting the data domains.

Perhaps more importantly, RTTL is a very 'expressive' syntax, the user can choose either temporal domain expressions, or operators. This flexibility may result in 'cleaner' specifications.

2.3 TRIO

2.3.1 Overview Of TRIO

TRIO [GMM89] is an extension to first-order predicate logic with two operators, *Futr* and *Past*, that range over formulae and temporal domain. These two operators, define future and past truths respectively. The temporal domain can be represented by natural numbers, integers, or reals, TRIO is unusual in being flexible in its temporal model. TRIO semantics are defined with a 'anchored'

validity model, i.e. the time of validity is relevant, hence both future and past time operators must be included in the language.

2.3.2 TRIO Syntax

The syntax of TRIO is built from terms and formulae, a term is defined inductively: every variable is a term, and every n-ary function applied to n terms is a term.

Formulae are similarly defined:

- every n-ary predicate applied to n terms (of the correct type) is an (atomic) formula.
- If A and B are formulae, then $A \rightarrow B$ is a formula.
- If A is a formula, then $\neg A$ is a formula.
- If A is a formula, and x is a *time independent* variable occurring free in A, then $\forall x \; . \; A$ is a formula.
- If A is a formula, and t is term of temporal type, then $Futr(A, t)$ and $Past(A, t)$ are formulae.

A time independent variable does not change over time i.e. given a value function σ ranging over variable names and a temporal domain T, then a variable 'x' is time independent if for some value v,

$$\forall t \in T \; . \; \sigma(x, t) = v$$

A time dependent variable changes over time, i.e. variable 'x' is time dependent if for some value v,

$$\exists t, t' \in T \; . \; \sigma(x, t) \neq \sigma(x, t')$$

A term is a temporal type if it evaluates to a value in the temporal domain.

Informally, the future and past operators can be thought of as having the semantics, 'Futr(A,t) means that after 't' time from the present, the formula A will evaluate to true', (in the past resp). From these two operators a collection of further ones can be derived, for example,

- $AlwF(A) \equiv \forall t \; . \; (t > 0 \wedge Futr(true, t) \rightarrow Futr(A, t))$.
- $Lasts(A, t) \equiv \forall t \; . \; (0 < t' < t \rightarrow Futr(A, t'))$.
- $Until(A_1, A_2) \equiv \exists t \; . \; (Futr(A_2, t) \wedge Lasts(A_1, t))$.

In [GMM89] thirteen operators are defined.

The semantics of TRIO are given by temporal structures which contain the type information of variables, the temporal domain definition, and time independent and time dependent definitions of states in terms of variable bindings and function and predicate definitions.

A temporal structure satisfies a TRIO formula if there is an instant of time in the temporal domain when the formula evaluates to true. A temporal formula is closed if it is temporally closed, i.e. the formula is independent of time, and it is classically closed (all variables are quantified.) Specifications are defined as closed TRIO formulae.

Decidability in TRIO is only achievable under certain restrictions. This is because of the inherent undecidability of the first order predicate calculus. The *tableaux method* of decidability restricts TRIO semantic structures to temporal domains with intervals of integers, and finite value domains for variables.

The method exhaustively builds a tree containing all possible satisfiable TRIO formulae given the expanding underlying state mappings. The tree will be built in a non-deterministic manner until either the formula to be decided is reached, or the complete tree is built. The fact that the domains are finite assures that the tree will also be finite, (families of equivalent formulae are represented by normal forms).

A Prolog program is outlined [GMM89] in which the decidability of a closed TRIO formula can be proven. Using variants of this code, and the concept of histories, a number of automated tools are made available to the user of TRIO. In particular, the verification of specifications with respect to histories can be automated

2.3.3 Assessment Of TRIO

TRIO formulae define the temporal relationship between events in the specified system. Absolute times are not mentioned, and instead constraints upon delays and execution times are given. In this framework, specific issues such as value and hardness can only be discussed in by defining some auxiliary function, and including an axiomatisation of predicates involving these functions. Multiple time bases cannot be defined (there is only one global clock). The semantics of TRIO are not compositional.

In use, TRIO 'feels' basic, it has no method and it is very flat (has no abstraction mechanism). I suspect that users are going to shy away from the monolithic specifications that will result for any realistic sized system.

2.4 RTL

2.4.1 Overview Of RTL

RTL [JM86] is a real-time logic with four basic concepts; *actions* which may be composite or primitive, *state predicates* which provide assertions regarding the physical system state, *events* which are markers on the (sparse) time line, and *timing constraints* which provide assertions about the timing of events.

2.4.2 Actions

Actions (which may be primitive or composite) are never instantaneous. For example, the action $VOFF$ (turn valve off) is primitive, the action $POFF; VOFF$ is a sequential composition of the

$POFF$ (pump off) action, and the $VOFF$ action. Actions may also be composed in parallel ($ACTION||ACTION$), or synchronised ($!N$ is the synchronisation point labeled N, allowing the denotations $!N$, $X!N$ *and* $!NY$ to enforce a precedence relation $X < Y$).

Each action has a duration which enables the user to determine worst-case execution times and timeliness properties. This is achieved by constructing a behaviour graph of the system (the action constructors define a partial ordering over the occurrence of actions) and weighting each transition with a cumulative execution time. The maximum execution time is then the path with the largest cumulative weighting.

2.4.3 Timing Constraints

The timing constraints define the periodicity of actions – they may be periodic or aperiodic, the following is an example of a periodic constraint:

while ELSM is off execute NOP1 with period=200ms, deadline=40

where $ELSM$ is a state predicate with values *on* or off and $NOP1$ is a composite action.

An example of an aperiodic timing constraint is:

When external event TMRINT occurs, execute TIH with deadline=60ms,
separation=100

The separation parameter defines the lower bound on the length of the interval separating any two occurrences of the event TIH.

2.4.4 The RTL Logic Syntax

RTL enables the user to write assertions about the behaviour of actions. This is achieved using predicate logic and a special function '@' which returns the time of a the occurrence of any event. Every action has two unique events associated with it to denote its start and finish. For action A, '$\uparrow A$' denotes the start of A, and '$\downarrow A$' denotes the finish. Because each action may start (occur) more than once, the function '@' is indexed for the occurrence of the event, e.g.

$@(\uparrow A, 5)$

gives the time of the fifth occurrence of the event marking the start of action A.

Actions may be subactions, and therefore an indexing of composite actions is provided, given a composite action A with a subaction B, then

$A.B_i$

denotes the ith occurrence of subaction B in A (where the occurrence is defined by the causality relation in A, *not* the time of the occurrence).

Example axioms of the RTL system include the *start/stop event axioms*,

$(i)\ \forall i \forall t\ .\ @(\downarrow A, i) = t\ \rightarrow\ @(\uparrow A, i) < t$

$(ii)\ \forall i \forall t\ .\ @(\uparrow C.A, i) = t\ \rightarrow\ \exists j\ .\ @(\uparrow A, j) = t$
$\wedge [@(\downarrow C.A, i) = t' \leftrightarrow @(\downarrow A, j) = t']$

which define a well-ordering between the start and finish events of actions; (i) asserts that all actions must have started before they may finish, and (ii) asserts that if an action is named as a composite action, then the occurrence of the subactions are the same occurrences as the actions each name represents (this allows for multiple occurrences of the same action).

2.4.5 Events

Apart from the start and finish events, external events may occur (as in the example of sporadic timing constraints). External events are marked with an Ω.

2.4.6 State Predicates

State predicates exist to denote the times at which the state holds a certain value. The predicates use different forms of parenthesis to define temporal boundary conditions. Given a state attribute s with events E_t and E_f which occur at the times at which s becomes true and false respectively, then state predicates are of the forms:

$$s[x, y],\ s(x, y),\ s < x, y >,\ s[x, y),\ s[x, y >,\ s(x, y],\ s(x, y >,\ s < x, y],\ s < x, y)$$

where x and y are times and,

$[x$ denotes that E_t occurs at time x.

$(x$ denotes that E_t occurs before or at time x.

$< x$ denotes that E_t occurs before time x.

$y]$ denotes that E_f occurs at time y.

$y)$ denotes that E_f occurs at or after time y.

$y >$ denotes that E_f occurs after time y.

2.4.7 System Specifications

Systems are firstly defined in terms of composite and primitive actions, and timing requirements in the form of timing constraints. A mechanical process is then applied to transform the actions and constraints into a set of RTL formulae. Safety and liveness properties are expressed in RTL, and are then proven using a predicate logic proof system.

The following example is a fragment from the 'Martian Lander' case study presented in [JM86]. The specification asserts that the lander may perform normal landing actions, or be interrupted with error conditions and asked to perform emergency landing actions. We first list the actions, events, and state predicates:

The actions:

IEM	initiate emergency mode (primitive)
ETC	some housekeeping function (primitive)
TIH = IEM;ETC	timer interrupt handler
ADJM	adjust motor thrust (primitive)
TDP	transmit info to display panel (primitive)
NOP2=ADJM||TDP	normal operation phase 2

and the event:

Ω IOINT i/o interrupt (external).

and state predicate:

ELSM emergency landing system mode (initially off).

The timing of the actions can be given by the table:

Action	Time in ms
ETC	10
ADJM	20
TDP	10
IEM	10

The following (periodic) timing constraint asserts that if the lander is not in emergency mode and normal operation ends, then there must be no more than a 40ms delay after the next 200ms period before the emergency landing system is operational, or the normal landing mode is re-entered.

$$\forall x \forall y \forall n \exists i [\overline{ELSM}[x, y) \wedge 200 * n < y - x] \rightarrow$$

$$[x + 200 * n \leq @(\uparrow NOP1, i) \wedge @(\downarrow NOP, 1) \leq x + 200 * n + 40]$$

The second (sporadic) timing constraint asserts that there is a minimum of 100ms between timing interrupts.

$$\forall i @(\Omega IOINT, i) + 100 \leq @(\Omega IOINT, i + 1)$$

Finally, the following assertion states that the emergency mode initiation action must have finished before the housekeeping action occurs.

$$\forall i @(\downarrow TIH.IEM, i) \leq @(\uparrow TIH.ETC, i)$$

2.4.8 Assessment Of RTL

Work is presently being carried out on finding an efficient general decision procedure for RTL formulae, presently it is a time consuming exercise to verify safety and liveness assertions using

standard deductive proofs. Also, a design method is mentioned which may provide an environment for the engineering of large real-time systems, Jahanian and Mok suggest that RTL may form a unified basis for a theory of decomposition.

RTL has been used with some success in industrial applications, it also is being used in a major IBM project called 'ORE' which is integrating RTL with a real-time programming language. There is a certain feeling of confidence with RTL perhaps due to its pragmatic nature (the mention of milliseconds in a specification is always good to hear).

2.5 PCTL

2.5.1 Overview Of PCTL

PCTL [Han89] is a real-time logic in which probabilities may be reasoned about. The semantics are defined using Markov chains. The temporal model is sparse, and the assertions involve states and state transitions in finite state machine semantics.

2.5.2 PCTL Syntax

PCTL syntax contains the standard propositional operators ($\neg$ and $\wedge$ etc) which define *state* formula. Using weak and strong until operators (U_w and U_s), *path* formulae are defined which are evaluated over sequences of transitions.

The path formulae may be decorated with probabilities to form state formulae, i.e. given the path formula:

$$f_1 \; U^{\leq t} \; f_2,$$

then $\quad [\; f_1 \; U^{\leq t} \; f_2]_{\geq p} \quad (where \; p \in [0,1] \; and \; U = U_w \; or \; U = U_s)$

is a state formula with the intuition that "before time t, f_1 will hold until f_2 does, with a probability of p".

2.5.3 PCTL Semantics

The semantics of PCTL are defined by Markov chains. A semantic structure is a 4-tuple (S, s^i, T, L) where S is a set of states, s^i is the unique initial state, T is a transition probability function mapping pairs of transitions to probabilities, and L is an evaluation function giving truth values to atomic propositions in specific states.

Given a semantic structure Γ, then the semantics of PCTL can be defined by the relations $\models^s$ (model over states) and $\models^\sigma$ (models over paths).

$$\sigma \; \models^\sigma_\Gamma \; f_1 U^{\leq t}_s f_2 \quad \text{iff } \exists i \leq t \; . \; \sigma[i] \; \models^s_\Gamma \; f_2 \; and \; \sigma[j] \; \models^s_\Gamma \; f_1 \quad \forall j . 0 \leq j < i$$

$$\sigma \; \models^\sigma_\Gamma \; f_1 U^{\leq t}_w f_2 \quad \text{iff } \sigma \; \models^\sigma_\Gamma \; f_1 U^{\leq t}_s f_2 \quad or \quad \sigma[j] \; \models^s_\gamma \; f1 \quad \forall j . 0 \leq j < t$$

$$s \; \models^s_\Gamma \; [f]_{\geq p} \quad \text{iff } \; \forall \, \sigma . (\sigma[1] = s \; \wedge \; \sigma \; \models^\sigma_\Gamma \; f) \rightarrow \mu(\sigma) \geq p$$

where f,f_1, and f_2 are state formulae, σ is a path in structure Γ, p is a probability, and t is a time. The indexing of the paths gives the state at the position index, and the μ function returns the product of the probabilities in the path parameter.

Two major derivable operators in PCTL are the A (universal quantification) and E (existential quantification) operators, provided by the definitions:

$$Af \equiv [f]_{\geq 1} \quad Ef \equiv [f]_{\geq 0}$$

Also, the operator G is used as an 'always' operator:

$$G^{\leq t}_{\geq p} f \equiv f \; U_w \, ^{\leq t}_{\geq p} \; false$$

with the intuition that "f will hold continuously for t time units with a probability of p". The 'sometimes' operator is similarly defined:

$$F^{\leq t}_{\geq p} f \equiv true \; U_s \, ^{\leq t}_{\geq p} \; f$$

Finally, the 'leads to' operator is defined:

$$f_1 \; \rightsquigarrow ^{\leq p}_{\geq t} \; f_2 \quad \equiv \quad AG \left[f_1 \rightarrow F^{\leq t}_{\geq p} \; f_2 \right]$$

The 'leadsto' operator has the intuition " after f_1 occurring, f_2 will occur within t time units with a probability of p". It is related to the concept of causality, and may be used to define safety requirements.

2.5.4 Assessment Of PCTL

The probabilistic approach to real-time logic allows the user to differentiate between hard and soft deadlines. A soft deadline may be expressed as "the deadline should be met at least 98% of the time". By using certainties we may define hard deadlines, in which cases the logic is expressively equivalent to RTTL or TRIO. There is no method associated with PCTL, or consideration of resources. The clock is global and there is no provision for multiple clocks. Communication and concurrency can not be reasoned about (the underlying finite state machines limit the language).

Only small examples are included in the reference paper, and larger systems need to be tried to investigate the decision procedure given (for finite time assertions).

PCTL is perhaps one of the most interesting formalisms mentioned in this review, it is the only one in which levels of criticality might be represented.

2.6 TEMPURA

2.6.1 Overview

TEMPURA [Mos86] is a programming language in which statements are real-time logic formulae, and program execution results in the finding of states for which the formulae evaluate to true. The underlying logic used in TEMPURA is Interval Temporal Logic (ITL) which uses sequences (intervals) of states in a sparse temporal model. ITL has a 'next' operator ($\circ$) which may be iterated to denote specific times.

The motivation behind TEMPURA is to provide an executable specification language in which the specification to implementation transformation steps are removed. Specifications are therefore written in a high-level programming language as well as a specification language, and this suggests that we may perform the same task with any formally defined programming language (for example SML [Wik87]).

The TEMPURA interpreter is **not** real-time, it cannot be used to execute real-time systems, for these systems it can only be used as a theorem prover.

Perhaps most importantly, TEMPURA contains a 'chop' operator for the sequential composition of formulae. This operator is essential for the translation of any imperative language into temporal logic. With the chop operator TEMPURA defines control flow constructs such as 'for-loops' and 'while-loops'.

2.6.2 Interval Temporal Logic

ITL contains expressions and formulae, both of which may contain the 'next' operator ($\circ$). In expressions (function evaluations and variables) the term $\circ e$ denotes the value of the expression e in the next discrete time step. In formulae, the term $\circ f$ expresses the assertion that f will evaluate to true in the next time step. Formulae may also contain the usual $\wedge$, $\Box$, and $\neg$ operators. The temporal model is anchored.

Example ITL formulae include:

i) $(J = 2) \ \wedge \ \circ(I = 3)$

ii) $(\circ\Box[I = 3]) \wedge \neg([\circ J] = 4)$

with the intuitive interpretations; i) J is equal to 2 now, and in the next time step I will be equal to 3, ii) from the next time step onwards I will always be equal to 3, and in the next time step the value of J will not be 4.

The ITL formulae are evaluated over sequences of states:

given a model (D, Σ, M) with D, a data domain, Σ, a set of states (mappings from names to values), M a set of meanings for functions, and σ, a sequence of states indexed by positive integers $(0..N)$, then the meaning of any term is given by the function M, for example:

$M_\sigma \ [V] \ = \ \sigma_0[V]$, where V is a variable.

$M_\sigma \ [f(e_1, .., e_n)] \ = \ M[f](M_\sigma[e_1], .., M_\sigma[e_n])$

$M_\sigma \ [\circ e] \ = \ M < \sigma_1 ... \sigma_{|\sigma|} > [e]$ iff $|\sigma| \geq 1$.

The language includes a number of constants, including *empty* and *more* which evaluate to true if the interval is empty at a specific time (for example $\circ \circ \circ empty$ is true of an interval of length 3), or in the case of *more*, if an interval is not empty at a specific time ($more \equiv \neg empty$). These operators allow us to reason about termination of programs.

2.6.3 The Language TEMPURA

The TEMPURA interpreter executes a set of formulae by searching for a sequence of states that satisfy those formulae.

A number of derived operators are added to the language to provide an imperative syntax. Including *stable*, (which asserts that a expression has a particular value and will remain bound to that value during all computations), *halt* (a formula becomes true only at the end of an interval), *gets* (an expression is equal to another expression with a time delay of 1), and $\simeq$ (two expressions are always equal).

The following TEMPURA program is true of a sequence of states in which a number (I) is repeatedly doubled until it exceeds 100.

$$(I = 1) \ \wedge \ halt(I > 100) \ \wedge \ (I \ gets \ 2 * I) \qquad (1)$$

I is initially set to the value 1, the halt statement is true of any sequence of states whose last state binds I to a value greater that 100, and the final conjunction forces each value of I to be double that of the previous value.

If we list any sequence of states in which formula (1) will evaluate to true, the final state will bind the value 128 to I, the formula could be said to be the program which computes the result $I = 128$.

2.6.4 Assessment Of TEMPURA

Tempura has a non-compositional proof system, can not represent criticality or faults,and has no syntax for concurrency or communication (although Moskowski is developing this aspect of the interpreter). A global clock is used.

2.7 Real-Time Logics – Comments

There are a number of other real-time logics which we may have looked at, but none with a substantially different underlying semantic model (excepting 'esoteric' temporal models, such as non-linear time etc). Pnueli's [Pnu87] temporal logic with fixed-points is seminal, and has been used as a basis for a number of logic systems (including Ostroff's RTTL). The paper by Galtan written as an overview of the book "Temporal Logics And Their Applications" [Gal87] contains a discussion on the representation of time within temporal logic, with particular reference to the domain/operator issue (also called the 'tensor/detensor' dichotomy).

The work by Barringer (also reported in Galton [Gal87]) is important because it presents a temporal logic with compositional semantics. It is becoming clear that such a compositional semantics is essential if we are to overcome the explosion in proof obligations caused by specifications of any realistic size. The composition is included by presenting incomplete semantic models which are parameterised over environments.

Fisher and Barringer present a short survey of programming logics [FB86] with a section on temporal logics.

3 Real-Time Process Algebra

3.1 Introduction To Process Algebra

Process algebra have carrier sets of processes, and operators which provide process constructors. The term (or free) algebra defines a language's syntax, and a set of equations provides a proof theory. Semantics are defined by a homomorphism from the term algebra into a labelled transition system, or a traces set. In real-time process algebra there exist process constructors which denote the passage of time. We examine four real-time process algebra: Timed-CCS, SCCS, Timed-CSP, and CSR.

We have used the algebraic specification language OBJ1 [GT79] to define the syntax and parts of the equational theory of each algebra.

3.2 An Introduction To OBJ1

OBJ1 is a specification language in which an algebra is defined using *objects*. Objects are carrier sets along with operations, and equational theories which are treated by the OBJ interpreter as re-write axioms. Each object is built from primitive *sorts* and enrichments of existing objects.

Proofs of equivalence are achieved automatically in OBJ by rewriting processes into their normal forms and testing for syntactic equivalence.

One important consideration when defining equational theories as sets of rewrite rules is that of *confluence*. A confluent set of rewrite rules guarantees that any term will eventually be rewritten into a unique (or isomorphic) normal form(s). Categorically (with terms as objects, and rule application as morphisms) a confluent rewrite set will result in a pushout existing for each span diagram. We would assume that process algebra equational theories formed confluent rewrite rule sets.

Termination is also important in automated rule rewriting systems, and many equational theories form non-terminating sets. Any rule directly expressing commutativity will form part of a non-terminating set (such a rule may be applied to processes infinitely often). In general, our use of OBJ1 for expressing equational theories for process algebra will encode non-terminating sets.

There are now a number of enhanced OBJ interpreters, including OBJ2, and OBJ3. The version we use here is modelled on the original OBJ, developed by Goguen and Tardo [GT79].

3.3 Timed CCS

3.3.1 Overview

Tofts [TM89] presents a real-time process algebra which is modelled on Milner's CCS [Mil80,89]. Timed-CCS includes primitives for asynchronous delay, timed delay, and strong and weak choice. The temporal model is sparse, assumes a global clock, and the operational semantics of the 'timed' actions (such as delay) are defined separately from the un-timed actions (such as communication). Un-timed actions are assumed to be instantaneous. An equational theory is presented with the usual expansion theorem. A bisimulation relation is used to prove semantic equivalence.

There are a number of versions of Timed-CCS, and we will concentrate on the *weak* version: wTCCS.

3.3.2 wTCCS Syntax

Primitive processes are built from a sequence of actions drawn from some predefined set. The sequence represents the order in which the process wishes to engage in the actions. The actions are either communication actions, or delays. Composite processes may be constructed from primitive processes using binary operators such as weak choice ($\oplus$), strong choice ($+$), or parallel composition ($|$). A fixed-point operator (μ) is provided to define recursive processes, and a restriction operator ($/$) allows for the 'hiding' of communication actions from the environment. The special process '0' denotes the process which may engage in no actions.

The set of actions includes δ, which denotes the ability for a process to wait asynchronously for a preemptive communication, and (t) where t is some positive integer, which forces a process to wait for at least t global clock ticks.

The following OBJ1 specification of wTCCS includes a fraction of the equational theory.

OBJ wTCCS-Process

Sorts Act

OPs

0	: $\rightarrow$ wTCCS-Process
.	: Act $\times$ wTCCS-Process $\rightarrow$ wTCCS-Process
$\mid$	: wTCCS-Process $\times$ wTCCS-Process $\rightarrow$ wTCCS-Process
$+$	: wTCCS-Process $\times$ wTCCS-Process $\rightarrow$ wTCCS-Process
$\oplus$	: wTCCS-Process $\times$ wTCCS-Process $\rightarrow$ wTCCS-Process
μ	: wTCCS-Process $\rightarrow$ wTCCS-Process
$/$	: wTCCS-Process $\times$ Act $\rightarrow$ wTCCS-Process

VARs

x : wTCCS-Process; a,s,t : Act (where s,t $\in$ Act.times, a $\in$ Act.comm)

EQNs

$\delta\delta x = \delta x$	$x + x = x$
$(s)(t)x = (s+t)x$	$x + y = y + x$
$\delta(t)x = \delta 0$	$0 + (t)x = 0$
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	$0 + ax = ax$
$\delta 0 = (t)\delta 0$	$x + \delta 0 = x$
...	

JBO wTCCS-Process

The notation used in the equational theory (above) does not use the prefix operator – this is a

common shorthand in wTCCS. Act.comm and Act.times are those subsets of the set of actions which represent communications and times respectively.

Distributed parallel, strong, and weak choice operators are also provided with the notation:

$\prod$, $\sum$ *and* $\coprod$ respectively.

The product and coproduct symbols are accurate indications of those limit constructions in the categorical representation of wTCCS [2]

The weak choice operator ($\oplus$) intuitively represents a construct whose processes may wait for a bounded length of time, until one process is able to communicate with the environment. If the two processes wait for longer than the prefixing delay of one of the processes, then that process becomes deadlocked. The strong choice operator ($+$) builds processes which will only wait for as long as *both* processes are able to wait, it then becomes inactive (if no communication is possible within that time).

The strong choice operator enables us to build a timeout operator in the following manner:

$$Timeout_t(P, Q) =_{def} \delta P + (t)Q$$

where t represents the timeout delay. The strong choice operator allows the two processes P and Q to wait for t ticks (the $\delta.P$ process is asynchronous, and so may wait for as long as necessary). If P has not communicated within t ticks, then Q *must* communicate, or the process will deadlock.

3.3.3 wTCCS Semantics

The operational semantics of wTCCS are defined by labelled transition systems (LTSs). The delay actions and communication actions are dealt with in two separate LTSs. The communication LTS is denoted by a relation:

$$\rightarrow \subseteq wTCCS - Process \times Act.comm \times wTCCS - Process$$

The temporal LTS is denoted by a similar relation :

$$\rightsquigarrow \subseteq wTCCS - Process \times Act.time \times wTCCS - Process.$$

A subsidiary semantic function is defined which returns the maximum amount of time a process is capable of waiting before it must participate in a communication action. The function is denoted $|.|_T$ and can be defined as follows:

$$\begin{array}{ll}
|0|_T = 0 & |P \oplus Q|_T = max(|P|_T, |Q|_T) \\
|X|_T = 0 & |a.P|_T = 0 \\
|(s).P|_T = s + |P|_T & |\delta.P|_T = \omega \\
|P + Q|_T = min(|P|_T, |Q|_T) & |P|Q|_T = min(|P|_T, |Q|_T) \\
|P/a|_T = |P|_T & |\mu x.P|_T = |P_i[\mu x.P/x]|_T
\end{array}$$

[2] For example,if we represent wTCCS as a category with wTCCS processes as objects, and temporal *simulations* [Mil89]p208 as morphisms, then finite coproducts represent weak choice constructions.

Note that the function does not include any time for the 'unwinding' of the recursive processes, (this is also reflected in the temporal semantics), this is because recursion is a *syntactic* device to facilitate the description of infinite processes, it is not a model for recursive function calling. The above function is used in the LTS transition rules that define the temporal semantics of the weak choice operator.

$$[1] \quad \frac{P \overset{t}{\rightsquigarrow} P'}{P \oplus Q \overset{t}{\rightsquigarrow} P'} \qquad (|Q|_T < t) \qquad\qquad [2] \quad \frac{Q \overset{t}{\rightsquigarrow} Q'}{P \oplus Q \overset{t}{\rightsquigarrow} Q'} \qquad (|P|_T < t)$$

Rule [1] asserts that if a process P may wait for t ticks (the $\rightsquigarrow$ transition models the passage of time), then the weak choice of P with some other process, Q, will eventually be able to evolve into just the P' process if Q cannot wait as long as P. Rule [2] is the symmetric assertion on Q.

Equivalence between processes in the wTCCS semantics is proved using a temporal bisimulation relation [TM89]. Such a relation defines two processes to be equivalent if they may both perform the same oredered sequences of actions at the same times. A temporal bisimulation relation is a set of pairs of processes (a subset of wTCCS-Process $\times$ wTCCS-Process), which is closed under the following property:

$$P \sim Q \quad \text{iff } \forall a \in Act.Comm, t \in Act.time,$$

$$\begin{array}{l} if\ P \xrightarrow{a} P'\ \ then\ \ Q \xrightarrow{a} Q'\ for\ some\ Q'\ with\ P' \sim Q' \\ if\ Q \xrightarrow{a} Q'\ \ then\ \ P \xrightarrow{a} P'\ for\ some\ P'\ with\ P' \sim Q' \\ if\ P \overset{t}{\rightsquigarrow} P'\ \ then\ \ Q \overset{t}{\rightsquigarrow} Q'\ for\ some\ Q'\ with\ P' \sim Q' \\ if\ Q \overset{t}{\rightsquigarrow} Q'\ \ then\ \ P \overset{t}{\rightsquigarrow} P'\ for\ some\ P'\ with\ P' \sim Q' \end{array}$$

A completeness result is proved for the semantics with a restricted syntax.

3.3.4 A Case Study In wTCCS

We now present a case study in wTCCS. The case study is based on a fault-tolerant reactor control system. The system constantly monitors the core temperature, and if the temperature becomes too high, it drops control rods into the core. The fault tolerance is provided by ensuring that the only way the rods can be kept out of the core is by periodic instructions to do so by the monitors within the system.

The case study highlights a number of problems with the existing syntax. Firstly, there is no sequential operator for processes. By assuming the existence of this operator we have managed to shorten the specification below by a factor of around 50%. Secondly, the semantics for the inactive process 0 are very strict, and a process that becomes 0 stops the global clock – all other processes within the system also stop. One solution might be to replace all 0 processes with $\delta 0$, but this substitution has a very different semantics and needs to be used with care.

In More Detail

Six temperature sensors provide readings simultaneously to three periodic processes called *assessors*. Each assessor performs some computation on the sensor readings, and sends the result along a channel to the voter. A *voter* process waits asynchronously to receive the readings along the three channels, votes on the values, and send the vote result along a channel to the *actuator*. The actuator process waits for the vote result, and if it does not receive a result within a pre-determined time, the actuator will timeout and drop some control rods into the reactor core. If the actuator receives a 'drop' vote, then the rods are also dropped. The only way of keeping the rods out of the core is a periodic instruction to do so from the voter process.

We can therefore isolate three types of process. An assessor (of which we need three), a voter (of which we need one), and an actuator (of which we also need one). We next specify the behaviour of these process types.

The Specification Of Subprocesses

Each assessor synchronously, and concurrently, receives temperature readings from six sensors. The readings are used in some computation, and the result is sent along a channel to the voter process.

There are some syntactic changes to TCCS in this (and future) processes, we will discuss these in some detail later.

$$A_n \ =_{def} \ \left(\prod_{i \in \{1..6\}} t_i(x_i).0\right) \ ; \ (\beta).\overline{c_n}(\alpha(x_i | i \in \{1..6\})).A_n$$

The β delay models the computation time of the assessor function α. The subscripted variables x_i can be thought of as an array of values.

The voter process waits for the readings from the three assessors, and chooses the value produced by the most channels (must be at least two). If one of the assessors fails to provide a reading then the voter remains inoperative.

$$V \ =_{def} \ \left(\prod_{i \in \{1..3\}} \delta.c_i(y_i).0\right) \ ; \ (\gamma).\overline{a}(v(y_i \mid i \in \{1..3\})).V$$

The γ delay models the computation time of the voting function v, which provides a boolean value based on the readings supplied by the assessor processes.

The actuator waits asynchronously for the voter result, and times out if it does not receive any result within a pre-determined interval. The rods are dropped unless a periodic instruction not to do so is received from the voter.

$$Acc \ =_{def} \ Timeout_t(a(x).Acc'(x) \ , \ \overline{drop}.Alarm)$$

$$Acc'(x)' \ =_{def} \ if \ x = tt \ then \ \overline{drop}.Acc \ else \ Acc$$

The timeout process $Alarm$ is presently undefined, it is assumed that some error handling process will be defined here. The timeout interval is denoted by t.

The Complete System

We compose the assessors and the voter, and restrict the channels between them to give us:

$$Sys =_{def} \left(\left(\prod_{i \in \{1..3\}} A_i\right) \mid V\right)/c_i$$

And finally, the system is composed from the assessor/voter subsystem and the actuator.

$$System =_{def} (Sys \mid Act)/a$$

Notes On Syntactic Amendments

In all the above processes we have use the construct of concurrent processes terminating and then sequencing to a further process. This is not a primitive in Timed-CCS, and has to be formally justified.

For now, we can assume that the syntax used is a shorthand $=_{sh}$, i.e.

$$\left(\prod_{i \in S} P_i\right) \; ; \; Q \quad =_{sh} \quad \left(\left(\prod_{i \in S} P_i[\surd.\delta.0/0]\right) \mid (\delta\surd)^{max(S)}.Q\right)/\surd$$

If $\surd$ is not in the sort of P_i or Q.

In our examples this equivalence can be trivially proved correct.

The problem is due to the use of 0 as a destructor, any process in concurrent composition with 0 is forced to terminate, i.e. time is stopped by this process, there is no concept of *well-termination*.

3.3.5 Assessment Of wTCCS

wTCCS is an extension of Milner's CCS, it provides an 'add-on' approach to real-time languages in which a new labelled transition system is included to cover two distinguished actions; delay, and bounded delay. A new weak choice operator is also included.

Timed-CCS can not represent criticality.It has a global clock (although local clocks can be defined with some complexity). It does have a compositional proof system (theorems may be proved about individual processes in isolation), but it is limited to temporal bisimulation i.e. a strict equivalence relation. The proof system could be extended to cover more detailed properties.

Faults may be represented in wTCCS by using the τ communication (the τ may occur without provocation from outside, and is invisible to an outside observer). The τ communication introduces *instability*, and true non-determinism into wTCCS processes.

Resources can not be reasoned about, and the maximal parallelism hypothesis is assumed.

Milner's CCS has been used with some success in industrial applications, and this gives some confidence in wTCCS (the fact that Milner was Tofts' supervisor, and helped with the development of wTCCS increases this confidence still further). The primitives chosen by Tofts seem to be a powerful and expressive set, some experimentation has been undertaken in specifying both multiple clocks and resource contention, and the language performs well.

3.4 SCCS

3.4.1 Overview

The Synchronous Calculus Of Communicating Systems (SCCS) [Mil82a] is an extension to CCS [Mil80]. Processes are defined as possibly non-deterministic sequences of actions drawn from some set of primitive actions *Act*. The concept of synchrony is enforced by the semantics of the parallel composition operator over processes (|). Two processes in parallel composition must proceed in a lock step fashion with each process providing an action at each step. The next actions from the two processes combine at each step into a single complex action. It is assumed that there is a global clock whose ticks trigger the next action in each process. The operational semantics are defined by a labelled transition system.

SCCS does not have instantaneous actions or communications (unlike wTCCS, see above), instead every action takes exactly one clock tick. The global clock therefore needs to have a granularity as fine as the minimum duration of any communication in the system, and longer communications need to be 'padded out' with delay actions. There is no concept of deadlock due to a process being unwilling to 'listen' to a communication, the only way of enforcing deadlock is by using the restriction operator.

3.4.2 SCCS Syntax

A special action '1' is provided to model a process idling for one step, this action forms an identity over the action set under the parallel composition operator. The prefix operator (:) is used to combine actions from the action set *Act* to form sequences of actions, or processes. The special process '0' represents the process that can engage in no more actions. The choice operator (+) represents a non-deterministic choice between two processes. The fixed point operator (μ) provides a means of constructing recursive processes. Finally, the restriction operator (/) restricts the actions a process may engage in.

The syntax of SCCS can therefore be described by the OBJ1 specification:

OBJ SCCS-Process

Sorts Act

OPs

0 : $\rightarrow$ SCCS-Process
: : Act $\times$ SCCS-Process $\rightarrow$ SCCS-Process
+ : SCCS-Process $\times$ SCCS-Process $\rightarrow$ SCCS-Process
| : SCCS-Process $\times$ SCCS-Process $\rightarrow$ SCCS-Process
/ : SCCS-Process $\times$ Act $\rightarrow$ SCCS-Process
μ : SCCS-Process $\rightarrow$ SCCS-Process

VARs

P,Q,R : SCCS-Process; α, β,a : Act

EQNs

$P|Q = Q|P$ $\qquad\qquad$ $\alpha : P|\beta : Q = \alpha\beta(P|Q)$

$(P|Q)|R \;=\; P|(Q|R)$ $\qquad$ $P|0 \;=\; 0$

$P|(Q+R) \;=\; (P|Q)+(P|R)$ $\qquad$ $E|\mu(x=1:x) \;=\; E$

$(\alpha : P)/a \;=\; 0 \quad if\ \alpha = a$

$(\alpha : P)/a \;=\; \alpha : (P/a) \quad if\ \alpha \neq a$

....

JBO SCCS-Process

For conciseness, a distributed choice operator is defined. The following statement represents the term with i instantiated with values from the set I, and the processes P_i composed with the choice operator $+$:

$$\sum_{i \in I} P_i$$

For example, the process $CHAOS$, which can repeatedly engage in any action from the primitive action set Act.

$$\text{CHAOS} \;=_{def}\; \sum_{a \in Act} a : \text{CHAOS}$$

Winskel [Win85b] provides a modal assertion language for SCCS. The language is based on Hennessey-Milner logic (also called *dynamic logic*).

3.4.3 SCCS Semantics

The semantics of SCCS are given by two labelled transition systems. In the first, the labels correspond to individual elements of the primitive action set and the transition relation models the prefixes of a process. For example, given the process:

$\alpha : 0$

The only legal transition is the one labelled by α taking the process '$\alpha : 0$' to the process '0'. This is shown by the transition:

$\alpha : 0 \xrightarrow{\alpha} 0$

There is a slightly more complex labelled transition system where the labels are sequences of actions, this is shown by the 'fat' transition arrow $\Rightarrow$, for example,

$\alpha : \beta : 0 \stackrel{s}{\Rightarrow} 0$ where s=$\alpha\beta$

The calculus of SCCS is extended by defining equivalence relations under the labelled transition systems. The calculus is also extended by defining more complex operators. For example, the idle operator, δ ,'allows' a process to idle indefinitely before engaging in its first action:

$\delta(P) \;=_{def}\; 1 : \delta(P) \;+\; P$

SCCS may be extended with a large toolkit of such operators.

3.4.4 Assessment Of SCCS

SCCS is one of the most expressive process algebra – from it we may derive an asynchronous calculus and CCS (for a discussion of this, see [Mil89]). The algebra is amenable to automation. SCCS is not as 'user-friendly' as CCS, there is no equivalent of an expansion theorem, and reasoning about processes at the syntactic level is a complex task.

Criticality and fault tolerance may not be represented in SCCS. There is a single global clock and local clocks would be difficult to define. The proof system is compositional but would need extending to include properties other than equivalence. Resources may not be reasoned about.

3.5 Timed-CSP

3.5.1 Overview Of Timed-CSP

Timed-CSP [DS89] is an extension of Hoare's Communicating Sequential Processes [Hoa85], with a dense temporal model providing a global clock. A delay operator is included along with some extended parallel operators. There is an assumption of a minimal delay between any two dependent action occurrences, but no minimum delay on any two independent actions. The semantics of Timed-CSP are given by timed traces, and a specification relation $\underline{sat}$ is provided for verifying predicates over traces.

3.5.2 Timed-CSP Syntax

Processes in Timed-CSP are built from sequences of communication actions. A special process $WAIT$ is parameterised over times from the real number time line and denotes a delay in the process. A number of binary operators for choice (deterministic and non-deterministic) are included.

The syntax of Timed-CSP can be defined by the following OBJ1 specification. again only a fraction of the equational theory is presented.

OBJ Timed-CSP-Process

$Sorts$ Act,Time

OPs

$STOP$	: $\rightarrow$ Timed-CSP-Process
$SKIP$	: $\rightarrow$ Timed-CSP-Process
$\perp$	: $\rightarrow$ Timed-CSP-Process
$WAIT$	: Time $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$\rightarrow$	: Act $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$\Box$	: Timed-CSP-Process $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$\sqcap$	: Timed-CSP-Process $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$\|\|$	: Timed-CSP-Process $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$_A\|\|_B$	: Timed-CSP-Process $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$\|\|\|$	: Timed-CSP-Process $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
;	: Timed-CSP-Process $\times$ Timed-CSP-Process $\rightarrow$ Timed-CSP-Process

μ : Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
f : Timed-CSP-Process $\rightarrow$ Timed-CSP-Process
$/$: Timed-CSP-Process $\times$ Act $\rightarrow$ Timed-CSP-Process

VARs
P,Q,R : Timed-CSP-Process; t1,t2 : Time

EQNs

$WAIT\ t1; WAIT\ t2 \equiv WAIT(t1+t2)$
$P\ \Box STOP \equiv P$
$STOP\ |||\ Q \equiv Q$
$(a \rightarrow P); Q \equiv a \rightarrow (P; Q)\ if\ a \neq \surd$
...

$P; (Q; R) \equiv (P; Q); R$
$SKIP; Q \equiv Q$
$WAIT_t 1\ ||\ WAIT_t 2 \equiv WAITmax(t1, t2)$

JBO Timed-CSP-Process

The intuition behind the operators is as follows:

'$a \rightarrow P$' is the process that is ready to engage in the action 'a' *at any time*, and then behave like the process P. Timing is introduced by assuming that after the occurrence of the action 'a', a short delay occurs before the process is ready to behave like the process P. The delay is defined by the global constant δ.

The family of operators $|$, $_A||_B$, and $|||$ are concurrent process composition operators. The operator $|||$ provides a true parallel operator rather than a non-deterministic interleaving. The alphabeticised operator $_A||_B$ provides a restricted form of parallel composition, the $||$ operator is unrestricted.

'$Wait\ t$' is a process that enforces a wait of t global clock ticks before correctly terminating.

'SKIP' is a process that models the correct termination of a process.

'STOP' is a deadlock process which may make no internal progress.

'$\perp$' is a divergent process that has no external actions visible but may continue with internal actions indefinitely.

'$\Box$, $\sqcap$' are deterministic and non-deterministic operators.

';' is a sequence operator.

'$/A$' is a hiding operator, hidden actions from the set A are no longer externally visible (but they still cause delay).

'f' is a renaming function over processes.

'μ' is a fixpoint operator that allows the definition of recursive processes.

3.5.3 Timed-CSP Semantics

The semantic model of Timed-CSP is based on observation and refusal *timed traces*. The traces are formed from sequences of time stamp and action ordered pairs. The observation traces denote those actions that a process may offer to the environment, the refusal traces give information on the relationship between observation traces and those actions a process may refuse the environment at any specific time.

The semantics are defined by functions from Timed-CSP terms into sets of traces. The function can be thought of as a homomorphism from the term (free) algebra (above) into an algebra of sequences of actions.

The semantic function $F_T[]$, as defined in Reed [Ree88], maps processes onto observation traces, and is (partially) defined by the equation:

$$
\begin{aligned}
F_T[a \rightarrow P] \; =_{def} \; & \{(<>, \aleph) | a \ni \sigma(\aleph)\} \quad \cup \\
& \{(<(t,a)> ^\frown (s + (t+\delta)), \aleph) | t \geq 0 \; \wedge \; a \ni \sigma(\aleph \lceil t) \; \wedge \\
& \quad (s, \aleph - (t+\delta)) \in F_T[P]\}
\end{aligned}
$$

where $^\frown$ is sequence concatenation, $t \lceil s$ is a restriction operator that removes elements of the trace 't' with actions not in the restriction set 's', $\aleph$ is the refusal set of the process, i.e. a set of pairs of time stamps and actions, and 'σ' is a projection function from the refusal set into the actions in the pairs of the refusal set.

The function F_T is also defined in terms of the remaining operators.

There exists a satisfaction relation $\underline{sat}$ between trace predicates and traces. The predicates are specifications of required behaviour.

3.5.4 Assessment Of Timed-CSP

It is important in specification languages for real-time systems that the temporal relationships between actions is maintained throughout the manipulation of the specification. In non real-time process algebra, the concurrency operators are usually *conservative*, that is, they degenerate into non-deterministic interleaves of the constituent processes' actions. In Timed-CSP, the concurrency operators are non-conservative, they do not degenerate. Instead, the algebraic rules for concurrent processes define the effect of composition on the temporal domain. For example, in the process that is the concurrent composition of two $WAIT$ processes, the result is a process that waits for the maximum of the two delays.

There exist no tools for the manipulation of specifications written in Timed-CSP. It is worth considering though, that the programming language occam, (see [INM88]), is based on the specification language CSP, (see [Hoa85]), which is an untimed version of the language reviewed here. It is argued by many that occam is simply an animation of CSP and as such provides a constructive proof method of the correctness of CSP specifications. This does not translate naturally to Timed-CSP, because divergence or non-termination are not provable by execution.

There are a number of other versions of Timed-CSP. The original timed version was presented by Reed and Roscoe [RR86], and the language presented in the Davies and Schneider paper uses Reed and Roscoe's semantic models.

3.6 CSR

3.6.1 Overview Of CSR

Communicating Shared Resources (CSR) [GL89] is a language with a higher level abstraction mechanism in the form of a *resource*. Resources are collections of processes that reside on a single *physical resource* – they are different to processes in that they may execute in true concurrency rather than interleaved concurrency. The process algebra is very similar to Timed-CSP with point to point synchronous communications, but unlike Timed-CSP, CSR uses a sparse clock, and contains extensions for asynchronous communications. Resources may operate in true concurrency, processes (on the same resource) must interleave their actions. Primitives for timeouts and interrupts are included in the syntax. CSR events are prioritised, and so choices on guards may be resolved in a similar manner to the PRI ALT construct in occam [INM88]. The semantics of CSR are given by state pairs, each pair consisting of a termination record, and a history. The termination record asserts that the corresponding history represents either a completed computation, or a computation still executing.

3.6.2 CSR Syntax

A CSR system is represented by a collection of resources, each resource being a collection of processes. Resources may execute in true concurrency; the processes wihin a single resource must interleave their actions. Processes are built from sequences of computations and communications.

Three functions statically define the priority of events. Communication events may be given different priorities depending on whether they occur in an input or output clause. The functions are denoted PRI_i for input actions, PRI_o for output actions, and PRI_l for resolved communication actions (computations).

Assuming we have the predefined sorts $Chan$ for channel names, $Comp$ for primitive computations (expression evaluations etc), $Comm$ for input and output communications (either $c!s$ or $c?s$ for output and input of message s along channel c resp.), and $Time$ representing a sparse clock, then we can denote the syntax of CSR with the following OBJ1 specification:

OBJ CSR-Statement

Sorts Chan,Comp,Comm,Time

OPs

$wait$: CSR-Statement × Time → CSR-Statement
$skip$: → CSR-Statement
$exec$: Comp × Time × Time → CSR-Statement
$arec$: Chan × CSR-Statement → CSR-Statement
$asnd$: Chan × CSR-Statement → CSR-Statement

$;$: CSR-Statement $\times$ CSR-Statement $\rightarrow$ CSR-Statement
$\Box$: Comm $\times$ CSR-Statement $\times$ Comm $\times$ CSR-Statement
$\times$ Time $\times$ CSR-Statement $\rightarrow$ CSR-Statement
$wthn$: Time $\times$ CSR-Statement $\times$ Time $\times$ CSR-Statement $\rightarrow$ CSR-Statement
int : Comm $\times$ CSR-Statement $\times$ Comm $\times$ CSR-Statement $\rightarrow$ CSR-Statement
$loop$: CSR-Statement $\rightarrow$ CSR-Statement
$evry$: Time $\times$ CSR-Statement $\rightarrow$ CSR-Statement

JBO CSR-Statement

OBJ CSR-Process $ENRICHMENT\ OF$ CSR-Statement

Sorts Chan,Pri

OPs
$plft$: CSR-Statement $\rightarrow$ CSR-Process
$inpp$: Chan $\times$ Pri $\times$ CSR-Process $\rightarrow$ CSR-Process
$outp$: Chan $\times$ Pri $\times$ CSR-Process $\rightarrow$ CSR-Process
$locp$: Chan $\times$ Pri $\times$ CSR-Process $\rightarrow$ CSR-Process
& : CSR-Process $\times$ CSR-Process $\rightarrow$ CSR-Process

JBO CSR-Process

OBJ CSR-resource $ENRICHMENT\ OF$ CSR-Process

Ops
$rlft$: CSR-Process $\rightarrow$ CSR-Resource
|| : CSR-Resource $\times$ CSR-Resource $\rightarrow$ CSR-Resource

JBO CSR-Resource

Note : The 'lift' operators ($plft$ and $rlft$) provides a syntactic mechanism to enforce correct typing.

The intuition behind the operators are as follows:

$wait\ t$ forces the process to wait for a minimum of t time units.

$skip$ causes the process to $wait1$ (syntactic sugar).

$exec$ is a bounded computation, it defines a minimum and maximum bound on the execution time.

$arec$ provides for an asynchronous receive action, this is also syntactic sugar and is converted into the synchronous version along with a medium process to perform the storage and forwarding.

asnd as *arec* for asynchronous send.

; provides a process sequence operator.

$\Box$ provides a $PRI-ALT$ construct with input and output guards, and a timeout parameter along with an exception process. The priority of the guards are given statically by the priority functions. The timeout is optional, in which case the ALT will block until a guard is ready.

wthn is a 'within' statement, which provides a timeout and exception process.

int is an interrupt statement, again with an interrupt handler. The handler is only invoked if the correct action is interrupted by an action with a higher priority.

loop provides a continuous invocation operator.

evry is a periodic invocation operator with the period specified.

inpp, *outp* declares a prioritised asynchronous channel for input (or output resp).

locp declares a prioritised local computation.

& is an interleaved concurrent operator.

|| is a parallel (true concurrency) operator.

The example specification given in [GL89] is that of a robot which must, using touch, sense the position of some object. The System is defined as four resources, each of which has a number of processes.

There is no equational proof system in CSR, so properties about the example could not be proved at the syntactic level.

3.6.3 CSR Semantics

The semantics of CSR are denotational, each set of resources is defined by a collection of state-history pairs. Each state is either terminated ($\top$), or non-terminated ($\bot$). Each history is a sequence of *assumption records* which are pairs of sets of events, and sets of sets of events. The sequences reflect the time of the global clock, the $i'th$ assumption record contains the set of events that may occur at the $i'th$ tick of the clock, and sets of higher priority events that may also occur at that time.

Semantic processes are defined which are prefixed-closed domains of histories. A meaning function is defined to translate the syntactic systems into semantic processes.

3.6.4 Assessment Of CSR

CSR is a process algebra for real time systems that treats resource issues as primitives, to be considered from the first specification. To facilitate this, *resources* are introduced as an extra level of abstraction, and the difference between true, and interleaved, concurrency is exploited in

an intuitive way. The largest drawback with CSR is the lack of a syntactic proof system – it is difficult to prove properties in the semantic domain (especially a denotational semantics).

Gerber and Lee intend to implement an operational semantics in the form of a labelled transition system to facilitate automated reasoning. Also, work is being undertaken to include a dynamic priority scheme in future versions.

3.7 Process Algebra – Comments

Specification languages based on process algebra are amenable to automated proof systems, (for example, in using the OBJ interpreter which treats the equational theory as a set of re-write rules).

Languages specified by process algebra techniques are inevitably design oriented, as they explicitly define the 'structure' of the systems they denote. With CCS, a forerunner of wTCCS, a process logic is introduced which fulfills the rôle of a high-level specification language, wTCCS does not yet have this facility and an interesting exercise would be to define extensions to Hennessey-Milner logic to provide this. CSR provides a resource oriented specification language in which configurations are defined, the serious limitation of this formalism is the lack of a syntactic proof system.

4 Pogramming And Graphical Languages

4.1 Introduction To Programming And Graphical Languages

There are a number of real-time high-level programming languages which have a formally defined semantics. These languages either provide a formal target for program verification, or a derivation procedure with automated assistance for proving program correctness. In this section we examine five such languages RUTH, RT-ASLAN, LUSTRE, TB-Nets, and Statecharts.

4.2 LUSTRE

4.2.1 Overview

LUSTRE [HPO88] is a high-level programming language based on sequence variables and multiple discrete clocks. Each variable is a sequence of values with the position of the elements reflecting the ticks of the variable's own clock. Clocks are defined as boolean predicates on the system global clock. Variable operations are defined only for variables with equal clocks, and operate on all of the values within the variable sequence. Programs are built from *nodes* (the LUSTRE equivalent of procedures), and *assertions* (which characterise the environment and provide an encoding of safety assertions). The LUSTRE compiler produces *control automata* which define the flow of control within the program. Temporal properties may be verified by model checking on an intermediate model produced by the compiler.

4.2.2 Clocks and Sequence Variables

LUSTRE has a global system clock (sparse) from which all other clocks may be defined by means of predicate. Therefore the system clock provides the minimum granularity of time. Variables are associated with clocks and represent sequences of values, with a single element for each tick of the variable's clock. Constants use the system clock, and have the same value at all times.

Data operators are only defined on variables whose clocks are identical, and perform an element by element calculation. For example, given two variables X and Y:

$$X := (Xe_0, Xe_1, .., Xe_n) \quad Y := (Ye_0, Ye_1, .., Ye_n)$$

then $Z := X + Y$ evaluates to:

$$Z = (Xe_0 + Ye_0, Xe_1 + Ye_1, .., Xe_n + Ye_n)$$

There are operators that work on sequences themselves, including head insertion, clock conversion, variable initialisation, and temporal filtering.

4.2.3 LUSTRE Programs

Programs are built from nodes, which provide input parameter binding and which return a value to the calling code. Nodes are referentially transparent and may be used in any expressions (regarding the same restrictions on clocks). Multiple output parameters are dealt with in tuples of values which may be dismantled into their constituent sequence values.

Assertions are used to define the program environment. There are two types, invariants and exclusion sets. Invariants define boolean expressions which will always remain true during program execution. Exclusion sets determine that only one of the boolean expressions in the list will evaluate to true at any one time. Both kinds of assertion can be thought of as *safety assertions*.

4.2.4 Compilation

Compiling a LUSTRE program gives rise to a control automaton which maps the declarative style code into a sequential program. The compiler flattens all of the nodes into a single set of equations and then searches for boolean 'memories' which become components of a flow graph. The initial state in the graph defines the initialisation operations, and the transitions are labelled with boolean state changes. The compiler also takes the assertions into account when building the graph, and ensures the graph meets the assertions.

4.2.5 The Verification Of Properties

The verification of safety properties is achieved by creating a finite program behaviour tree from the possibly infinite behaviour. A closed bisimulation relation is defined on vertices of the behaviour tree and, assuming convergent behaviour, any program may be represented in this way. The assertions and exclusion sets may now be automatically checked by visiting each node of the tree in turn. Program equivalence is provable by finding a bisimulation between root vertices.

Because the program behaviour tree is created at compile time, and the properties are expressed within the program as assertions, the program must be recompiled each time we have a new property that we wish to prove. Work is being carried out presently on producing an incremental compiler which expands the behaviour graph rather than creating a new one.

4.2.6 Assessment Of LUSTRE

LUSTRE is a high-level programming language that may also be used for writing specifications. By using assertions within the code, safety properties may be verified by the compiler. The use of assertions on the input and output parameters of nodes provides a modular proof system which greatly reduces the explosion of proof obligations created by monolithic specifications.

An interesting consideration with LUSTRE is that, to retain consistency, the data operations must take place instantaneously, i.e. if we are to add two sequences, then the resulting sequence must be formulated before the next clock tick. This is called the *strong synchrony* hypothesis – all operators must react instantaneously to their inputs. In reality this hypothesis only dictates that the execution of the program be fast enough that the environment is not delayed in any way.

Also, there is no concept of *failure* in LUSTRE. In real-time systems, deadlines may be missed, and LUSTRE has no way of detecting this and acting accordingly. In many ways, the assumption that the environment will be temporally correct is not realistic, and this problem needs to be addressed.

4.3 RUTH

4.3.1 Overview

RUTH [Har88] is a real-time declarative language with a formally defined semantics. A RUTH program consists of a number of functions operating on time-stamped data streams. Functions may operate concurrently, and each may utilise a separate (sparse) clock. A number of streams are 'environment' streams and connect to i/o devices such as screens, keyboards, and disks. It is assumed that no clock drift occurs between functions. Functions may not be created dynamically, or destroyed during system execution. Channel time stamps define when messages are required to arrive rather than being stamped as they arrive, giving the programmer an opportunity to define required behaviour, and recover from temporal faults if they occur.

4.3.2 RUTH Programs

RUTH consists of a number of primitive data domains with associated operators, some control flow structures, and configuration structures. The control flow structures include conditionals (**if..then..else**), lambda abstractions, assignments, and recursive assignments. Data structuring is achieved using list s-expression style equations.

Channels are evaluated in a lazy manner allowing them to be represented by functions, and evaluated as their results are needed. To ensure a strictly monotonic time stamping in channels an incremental encoding is used. The time stamp of a channel element represents the difference in time between the last time stamp and the current one over and above the minimum increment of 1.

Channel constructor operations are provided to take the head element and tail channel of any channel. Syntactic sugaring allows channels to be defined by using channel delimiters. The following RUTH expression shows a channel construction function which adds a message to a channel:

$$
\begin{array}{ll}
\textbf{ConsCh} & (a, t, ch) = \\
& [\{t, a\}, (t \geq t' \rightarrow \bot,\ [\{t' - t - 1, a'\}, rest])] \\
& \textbf{where} \\
& [\{t', a'\}, rest] \ = \ ch
\end{array}
$$

The function takes a channel (ch), a message (a), and a time stamp (t), and adds the message and time stamp onto the front of the channel. The function is successful only if the parameter time stamp is less than the stamp on the existing head (the incremental encoding demands this restriction). The channel returned if the time stamp does not meet this requirement is the one with the message as the first element and the undefined channel $\bot$ as the tail.

Time is accessed using the function *Time* on channels, which returns the time stamp of the first message in the head. Using this function we may define a timed merge function (producing a lazy channel with uncomputed tail) by the following code:

$$
\begin{array}{ll}
merge & = \textbf{lambda}\ (ch1, ch2) \\
& \quad \textbf{if time}\ (ch1)\ \leq\ \textbf{Time}(ch2) \\
& \quad \textbf{then ConsCh}\ (\textbf{HeadCh}(ch1), \textbf{Time}(ch1) + d,
\end{array}
$$

$$merge(\mathbf{TailCh}(ch1), ch2))$$
else ConsCh $(\mathbf{HeadCh}(ch2), \mathbf{Time}(ch2) + d,$
$$merge(ch1, \mathbf{TailCh}(ch2)))$$
where
d = ...; – some non-negative constant
endwhere

The timed merge function merges two channels dependent upon the time stamps of the messages.

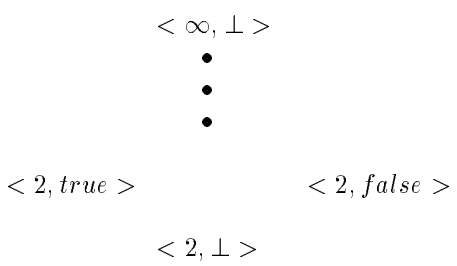
RUTH configurations consist of a mapping from channels between external devices and RUTH functions. In the following configuration the *kb* channels are connected to keyboards, and the *scr* channels are connected to screens. The **is** clause maps these channels to specific functions.

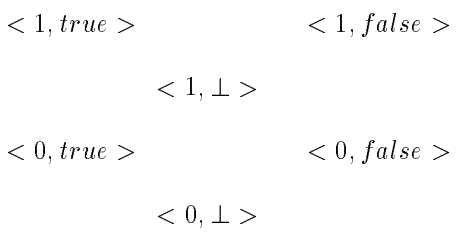
Configuration *system*
Output $scr1, scr2$
Input $kb1, kb2$
is $scr1, scr2 = supervisor(kbs);$
$kbs = merge(kb1, kb2);$
end

4.3.3 RUTH Semantics

RUTH's semantics are denotational and use *herringbone domains*. Herringbone domains were first used in ART [Bro87]. These domains contain time stamps (from a suitable temporal domain such as the real or integer numbers) on every value element. Thus domains are all extended by producing a Cartesian product of the original value domain and the clock value domain. The new ordering is defined point-wise on (primarily) the time, and (secondarily) the value.

Herringbone domains are named after their shape when drawn as graphs with node positions reflecting the partial ordering. An example is the domain HD_{bool} (herringbone domain of booleans) with the graph shown overleaf. In this example, the ordering represents the time taken to apply a function. For example, the value $< 2, true >$ represents a function completing execution at time 2 and producing the result *true*. The value $< 2, \perp >$ shows that a function has not yet completed execution at time 2.





HerringboneDomainOfBooleans

Because herringbone domains are not standard domain theory (yet), Harrison justifies their correctness by showing how all such domains may be proved to be isomorphic to lifted (non herringbone) domains.

The semantics of RUTH expressions are given by a set of denotational functions parameterised over environments and clocks (each function may have a different clock). The semantic functions are monotonic over the herringbone domains.

4.3.4 Assessment Of RUTH

Harrison aimed to produce a real-time language that fulfilled a number of requirements, including: the ability to define the occurrence times of particular events, detection of temporal errors and recovery, and a formal semantics to allow program verification. In his 1988 thesis, Harrison suggests that there are no theoretical problems in producing an implementation of RUTH, but, as we have already seen with languages like occam [INM88], it is difficult to ensure that the implementation of a formally defined language is faithful to the semantics.

The importance of RUTH lies in the formal semantic definition. There are few languages which can claim to have such a semantics. Using RUTH it would be possible to perform program verification without leaving the formal environment. Specifications written in temporal logic could be interpreted over herringbone domains.

4.4 RT-ASLAN

4.4.1 Overview Of RT-ASLAN

RT-ASLAN [AK86] is both a specification and implementation language for real-time systems. It is not an *executable* specification language, but rather a language that may be used to express highly abstract properties, and implementation details. This flexibility in development language allows a formal derivation (or refinement) process to take place. An RT-ASLAN specification/program consists of a number of levels, each of which is a refinement of a higher level. Each level describes the behaviour of a number of processes and state transitions. The processes are thought of as concurrent, and may communicate through *formal interfaces*. Invariants and constraints are defined at each level, and the RT-ASLAN compiler generates proof obligations which occur between levels that the user must discharge to confirm that the refinement steps are correct.

4.4.2 RT-ASLAN Specifications

A specification in RT-ASLAN consists of a number of layers, each of which may contain a number of processes with invariants and constraints. *Transitions* define the state changes possible, and using logical relations, computation results may be specified. Subspecifications are composed into larger specifications (providing abstraction). Specifications may be of one of two types *interface*, or *communicating*. Interface specifications may contain variables that are visible to other specifications, and they may also have IN and OUT parameters. The former allows communication of values, the latter allows the 'calling' of transitions. Called transitions may be exclusive (only executed in isolation) or inherited (executed many times in parallel). Transitions in communicating specifications may also be defined as cyclic (with a specified period), or external (aperiodic). To perform sequencing, a **DO..BEFORE..OD** construct is provided.

The following example is a fraction of the specification found in [AK86], the line numbering has been introduced for reference:

[01] SPECIFICATION $Process_Control_System$
....
[02] COMMUNICATING SPECIFICATION $Controller$
[03] LEVEL Top_Level

[04] TYPE $Signal_Type$

[05] CONSTANT
[06] $Process_Adjustment(Data_Type, Data_Type) : Signal_Type$

...
[07] TRANSITION $Control_Process$ CYCLE 2
[08] EXIT
[09] DO
[10] $Start_Read$
[11] BEFORE $Read_Copy$
[12] BEFORE End_Read
[13] & $Control_Signal = Process_Adjustment(Last_Value', Copy')$
[14] & $Last_Value = Copy'$
[15] OD
[16] END Top_Level
....
[17] INTERFACE_SPECIFICATION $Controlled_Date_Set$
....
[18] INITIAL
[19] FORALL d:$Copy_Type$
[20] $Readers(d) = EMPTY$ && $Writer(d) = Null_Process$
....
[21] INVARIANT
[22] $State(Completed_Copy) = Completed$
....
[23] TRANSITION $Start_Read$ EXCLUSIVE

....

There are two types of specification in this example, lines 1-16 contains a communicating specification, lines 17-21 an interface specification. The transition *Control_Process* is activated periodically. On exit it performs sequential calls to *Start_Read*,*Read_Copy* and *End_Read* (lines 10-12). The transition *Start_Read* (line 23) is an exclusive transition in the interface specification *Controlled_Date_Set*. The invariant (line 21) will give rise to proof obligations between this level and the next level of refinement. The initial assertion (line 18) will provide the justification of the inductive basis for the proof of induction generated by the compiler for the correctness of the code. The transition call to *Process_Adjustment* (line 13) is a parameterised transition within an interface specification, the quote is used as an 'old-value' operator (viz Z).

4.4.3 The Semantics Of RT-ASLAN

The RT-ASLAN compiler produces a list of first order logic assertions from the specification syntax. From this list, the invariants and constraints are used to form proof obligations (so-called conjectures). State trees are built from the 'initial' clauses and the transition rules. Proofs of correctness (of the invariants) are inductive over these trees.

RT-ASLAN has a non-procedural semantics, that is, unless a variable is explicitly defined after a state transition, then it is considered undefined. However, this does not pose a problem to the programmer, who is blissfully unaware of most of the underlying re-writing the compiler performs. In the case of 'simple' variables, the post-quote ($'$) is used to denote the next value of any variable, and the compiler inserts the qualification that all other variables remain unchanged. In the case of complex data structures e.g. arrays, the compiler expands the syntax into quantified expressions.

The periods of the cyclic transitions can be thought of as performance constraints. Firstly, all transitions must be able to complete processing within the period dictated by their cycle time. Secondly, assuming each specification denotes a configuration assigning processes to processors, the total work must be be performed within the amount of time dictated by the set of periods. The RT-ASLAN compiler produces proof obligations based on these premisses.

4.4.4 Assessment Of RT-ASLAN

RT-ASLAN produces 'implementation-like' code through a process of derivation (or refinement). There does exist an implementation of ASLAN (a precursor to RT-ASLAN), and currently, the RT-ASLAN extensions are being added. RT-ASLAN provides an automated tool to provide proof obligations (although it does not try to perform the proofs – this is an obvious next step).

4.5 Timed Petri-Nets

4.5.1 Overview

In timed Petri Nets, time can be represented by placing timing constraints on the transitions or the states. An example would be to give each transition a maximum and minimum duration with

which they may wait after being enabled before they fire. In TB-Nets [GMMP89] the tokens are themselves time stamped with the time at which their creating transition was fired. A function is associated with each transition which is parameterised over the time stamps of the tokens in the transition's input places, and defines an interval within which firing may take place. Transitions are not forced to fire.

4.5.2 TB-Net syntax

TB-Nets are Petri-Nets (see Reisig, Petersson [Pet81],[Rei85]) with tokens which are time stamped. Each transition is decorated with a time condition function which maps time stamps to sets of times. The time condition function defines the temporal conditions on which the transition is allowed to fire.

The transition is never *forced* to fire, and to reflect this fact, TB-Nets are called *weak* time semantics. The addition of the token time stamp and the condition function is the only difference between TB-Nets and ordinary Petri Nets.

By defining the condition function as an interval based on the input token's time stamp, minimum and maximum bounds can be defined for transitions.

4.5.3 TB-Net Semantics

The TB-Net semantics are defined by translating the TB-Nets into ER-Nets, which are high-level nets with no primitive notion of time. ER-Nets have tokens which contain environment functions, i.e. functions mapping identifiers to values. Each transition has an associated predicate on environments, and a transition is enabled if there is an environment (token) in each input place, and the transition's predicate evaluates to true.

ER-Nets are a very powerful general net type in which TB-Nets can be embedded. The time is modelled in ER-Nets by including a temporal variable in each environment, and ensuring that a timed net conforms to a number of temporal conditions (particularly monotonicity on a local and global level). TB-Nets are then defined by encoding the temporal conditions in the transitions, and including a temporal variable in the tokens.

This semantic description is somewhat recursive, TB-Nets have semantics that are defined in terms of ER-Nets. ER-Nets have semantics which are defined in standard net theory terms.

4.5.4 Assessment Of TB-Nets

Petri Nets have a well defined decision procedure in the form of reachability trees, (although for general Petri Nets, the decision procedure is not total, and this would include ER-Nets – however in many cases the tree method works well !). Timed Petri Nets would also be amenable to automated theorem provers concerning possible net markings. However, timed nets suffer from the same disadvantages as all Petri Nets, i.e. a lack of compositionality, both in the syntax, and the semantics. Nets need to be complete before any properties are proven, or reachability graphs drawn.

4.6 Statecharts

4.6.1 Overview

Statecharts [HRR89] provide an abstraction mechanism based on finite state machines. Charts denote composition of state machines into super-machines which may execute concurrently. The state machines contain transitions which are marked by enabling and output events. It is assumed that events are instantaneous, and a global discrete clock is used to trigger sets of concurrent events. Statecharts are hierarchical, and may be composed into complex charts. Transitions may be linked between charts by chart construction operators. The semantics of statecharts are given by maximal computation histories. An axiomatic proof system is presented.

4.6.2 Statecharts Syntax

The language of statecharts presented here is based on earlier work by Harel [Har87], and extended by Hooman, Ramesh, and de Roever [HRR89]. Statecharts have been proved to be at least as expressive as state machines, and the succinct justification for them is provided by the 'equation':

Statecharts = state-transitions+depth+orthogonality+broadcast communication.

The depth is provided by allowing statecharts to be built from other statecharts in a compositional syntax. The communication is broadcast because any event created by a state transition may be used in any level to enable another transitions.

A basic statechart is defined by a triple $[I, O, N]$ where I is a set of input transitions, O is a set of output transitions, and N is the state name. This statechart is built from a single state. Given a statechart we may perform a number of operations, including: 'statification' (making a higher level statechart from one (or more) other statecharts), 'Or' construct (connects two statecharts in a disjunction), 'And' construct (connects two charts in a conjunction), 'connect' construct (connecting transitions) , and a 'hide-closure' construct (hiding events from the environment). The labels on the transitions are pairs (E/A) where E is an enabling condition built from boolean expressions on events, and A is a set of events which are 'caused' by the transition.

Every statechart has a number of control tokens which move from state to state as the transitions become enabled. As transitions fire the caused events are passed to the environment, and may be used to enable future transitions. The 'or' and 'and' constructions correspond to a choice and parallel composition in process algebra.

The basic statechart can be graphically represented by the diagram:

The 'or' and 'and' constructions cause statification before the disjunction/conjunction.

The arrows point to the default entry charts/states, and inter-chart arrows show causal dependency between charts.

The following example (taken from [HRR89]) shows a complete system with an OR between charts B and C, and an 'and' within B of D and E. The default entry point is state I in C, if the event 'a' does not occur at the first global tick, the state K is entered. From here, if 'a' continues to be dormant, then the charts D and E are entered at G and H, and so on...

The formal syntax is defined as follows:

```
S       ::= Disj | Conj
Disj    ::= Prim | Or(Disj—Disj) | Connect(Disj—T1—t2)
Conj    ::= And(Default,Default) | And(Default,Conj)
Prim    ::= Basic | Default | HiCl(S,a)
Default ::= Stat(Basic,S,T1)
Basic   ::= [I,O,N]
```

where I and O are sets of transitions, S is a statechart, $T1$ and $T2$ are transitions, and 'a' is an event.

There are a number of restrictions on the syntax to ensure well-foundedness of all statecharts.

4.6.3 Statechart Semantics

The semantics of statecharts are given in histories, a history is a 5-tuple (ss,i,f,o,es), where:

ss – is the start step (transition).

i – is an incoming transition (may be '*' to denote implicit entry).

f – is a function that denotes the events caused by the statechart at each tick of the global clock, 'f' also encodes all of the events caused by the environment.

o – is an outgoing transition (or '*' for implicit exit, or $\perp$ when there is no exit).

es – is an exit step.

The relation $\underline{sat}$ is used to denote the satisfaction of an assertion by a statechart, and an axiomatic proof system is provided. An example of a proof rule is as follows:

$$\frac{S_1 \ \underline{sat} \ \phi_1 \ , \ S_2 \ \underline{sat} \ \phi_2 \ ,}{Or(S_1,S_2) \ \underline{sat} \ \phi_1 \vee \phi_2}$$

Which states that the Or construct for statecharts models the disjunction in propositional logic.

The soundness and relative completeness of statecharts has been proven.

4.6.4 Assessment Of Statecharts

Statecharts provide a graphic representation of a specification language which is very similar to wTCCS. The events are instantaneous, there is a global sparse clock, processes may be constructed in choice or concurrent constructs, and events (actions) may be hidden from the environment. The major difference between Statecharts and wTCCS is the true concurrency of events. There have been a number of theoretical issues with statecharts which have been uncovered (for example the

transition which may enable itself), and the semantics have had to have been altered to solve this kind of problem.

4.7 Programming And Graphical Languages – Comments

Programming languages with formal semantics provide a target for formal verification methods. However, with the exception of LUSTRE, the languages reviewed above are not implemented (although in both cases the necessary work is being undertaken).

The main problems with LUSTRE are the strong synchrony hypothesis, and the lack of compositionality. However, the ability to mix assertions and code, and have the compiler verify the assertions is a very powerful method. Also, LUSTRE cannot detect errors in the environment, it is assumed that the environment behaves perfectly.

Graphical languages provide an intuitive visualisation of the specification as it is being written. Often, dependencies can be identified with greater ease if the user is forced to draw communication pathways between, and enclosures around, modules. In statecharts this would certainly be the case, where communication restriction is visible. However, it is clear that in most graphical languages the explosion in the physical size of the specification becomes a hinderance rather than a help. There are few examples of such languages being successfully used (in their graphical representation) for any sizable specifications. Also, there exist no graphical proof systems, and therefore all of these languages must be first translated into a mathematical intermediate code, losing much of the advantage of the graphics.

5 Current Research Projects

5.1 Introduction

This section gives an overview of some of the current major research projects involved in the formal development of real-time systems.

5.2 SPEC

The SPEC project is a large Esprit project (Esprit 3096) researching into a number of different formalisms for real-time systems. The project includes the METATEM group, the Statecharts work at Eindhoven, and the Oxford PRG work on timed versions of CSP:

METATEM. Gabbay (Imperial College) and Barringer (University Of Manchester) are developing the executable specification language METATEM [BFGGO89]. METATEM is a general framework for executing temporal logics. A METATEM 'program' consists of a set of antecedent/consequence pairs (in the form of temporal logic implications), which are executed as conditional imperative statements. The underlying temporal logic is expected to have operators which may represent both past and future truths.

Statecharts. Hooman and de Rooever are concentrating on the Statecharts specification language (reviewed in the section on programming and graphical languages) [HRR89] [KST90]. The Statecharts language is a graphical, hierarchical, specification language with concurrency and real-time primitives. Future plans include developing assertion/guarantee semantics.

Oxford PRG. The Programming Research Group at Oxford University is continuing the development of timed versions of CSP. The initial temporal model for CSP was developed by Reed and Roscoe [RR86], and has since been developed further by Davies and Schneider [DS89]. Current work includes instability sets, event timing functions, timewise refinement, and probabilistic models.

Morgan is also pursuing temporal refinement extensions to his refinement calculus [Mor90] [MRG88].

5.3 Real-Time Computing: The Next Generation

This project is a SERC funded project based at the University Of York. The project objectives are to identify issues relevant to the specification, design, and implementation of distributed real-time systems, and the design of an appropriate development environment. The TAM specification language [SZ90] has already been developed, and future plans include the development of PICEP – a pictorially interactive specifier and theorem prover for TAM.

5.4 CoS

The CoS project (Combined Specifications) is a SERC project based at the University of York. CoS will concentrate on examining existing specification languages for real-time systems, and how those languages might be combined. The CoS project will also concentrate on the executability of formal languages.

5.5 TEMPURA

Moskowski's TEMPURA project [Mos86] at the University of Newcastle is responsible for the executable interval logic interpreter TEMPURA (reviewed in the previous section on real-time temporal logics). Work in progress includes the SERC funded project to extend TEMPURA to cover concurrency and fault tolerance (there are no plans to add real-time specific features). The project also includes researchers from the University of York.

5.6 ORE

The ORE project [DJM89] is concerned with the design and analysis of real-time systems. The ORE run-time system was designed primarily to allow for experimentation, both at the application layer, and the system layer. The process model of ORE was inspired by the work of Jahanian and Mok [JM86], and the influence of the RTL specification language has grown until it now forms part of the ORE programming language syntax. The project is now investigating the use of temporal assertions as invariants in the program syntax, and is moving towards temporal refinement.

5.7 DEMON

The DEMON project (DEsign Methods based On Nets) is an Esprit project at the University of Hildesheim [BCDE90] [BV84]. The work is not specifically about real-time Petri-Nets, instead it concentrates on basic research into free choice nets, traps, and net compositionality. The project has provided proofs about a number of important properties of nets e.g. the proof that the home state property is polynomially decidable.

5.8 Edinburgh University

Walker and Stirling are concentrating on the Modal-mu Calculus [KST90]. The Modal-mu Calculus is a logic which is particularly suited to reasoning about event transition systems (e.g. labelled transition systems or finite state automata). The calculus is not quantitative, but real-time extensions may be added in future versions.

Milner, Parrow and Walker are developing a higher order version of CCS called Mobile-CCS [MPW89]. With this process algebra we may reason about resources, permissions, and higher-order computations. Again, Mobile-CCS is not specifically real-time, but it may be possible to add real-time extensions of the form used in wTCCS [TM89].

6 Conclusions

In this section we summarise the important characteristics of the languages surveyed in the previous three sections. We also discuss areas in which most of the languages are incomplete. Finally, a metric for expressibility (based on category theory) is proposed.

6.1 Summary Of Language Characteristics

In the following tables the criteria headings have the following meanings:

Temporal model is the model of time used by the language. A sparse model has discrete instances of time and there is a minimum granularity. A dense model is not discrete, between any two instances in time there is an infinite number of other instances.

Domain/operators in temporal logic indicates that the logic has operators to axiomatise time (e.g. the henceforth operator $\Box$), or extends predicates over a temporal domain (e.g. P(t) is true of predicate P at time t).

Modular abstraction indicates whether the language syntax supports compositionality (note: the syntax may offer compositionality without semantic support).

Development technique refers to any specific method available for the language.

Criticality/failure/concurrency/multiple clocks/communication/resources refer to the syntactic support for these issues.

6.1.1 Real-Time Logic

Criteria	RTTL	TRIO	RTL	PCTL	TEMPURA
Temporal Model	sparse	either	sparse	sparse	sparse
Domain/Operators	both	domain	domain	both	operators
Semantics	trajectories	model	model	Markov	intervals
Proof System	axiomatic	axiomatic	axiomatic	axiomatic	automated
Modular Abstraction	ESM	none	action	none	none
Development Tech	semi-formal	none	semi-formal	none	none
Automated Support	dec. proc.	dec. proc.	none	none	execution
Criticality	none	none	none	probabilities	none
Failure	none	none	none	none	none
Concurrency	interleaved	none	interleaved	none	none
Multiple Clocks	none	none	none	none	none
Communication	sync	none	none	none	none
Resources	none	none	none	none	none
Reference	[Ost89]	[GMM89]	[JM86]	[Han89]	[Mos86]

Notes.

The semantics of TRIO may be instantiated with either a sparse, or dense, temporal model. However, the decision procedure needs a sparse model (and finite domains).

The axiomatic proof system used in RTTL is adopted from Manna and Pnueli's three layered proof system [MP83]. The layers provide the valid theorems of the logic, an axiomatisation of the data domains, and an axiomatisation of the system and requirements.

The probabilistic semantics of PCTL defines criticality by associating a probability for each deadline being met.

6.1.2 Real-Time Process Algebra

Criteria	wTCCS	TCSP	SCCS	CSR
Temporal Model	sparse	dense	sparse	sparse
Domain/Operators	domain	domain	domain	domain
Semantics	LTS	traces	LTS	traces
Proof System	bisimulation	axiomatic	bisimulation	axiomatic
Modular Abstraction	process	process	process	resource and process
Development Tech	none	none	none	none
Automated Support	none	none	none	none
Criticality	none	none	none	none
Failure	none	none	none	none
Concurrency	interleaved	both	interleaved	both
Multiple Clocks	none	none	none	none
Communication	sync	sync	sync	sync & async
Resources	none	none	none	yes
Reference	[TM89]	[DS89]	[Mil82a]	[GL89]

Notes.

CSR has both interleaved and true concurrency operators. This reflects the extra abstraction mechanism of the *resource*, resources may run in parallel but processes on resources must interleave their actions.

CSR also provides for synchronous and asynchronous communication mechanisms. In other process algebras, asynchronous communication may be modelled by defining an explicit medium, but it is not a primitive concept.

6.1.3 Programming And Graphical Languages

Criteria	RUTH	LUSTRE	RT-ASLAN	TB-NETS	Statecharts
Temporal Model	sparse	sparse	sparse	either	sparse
Domain/Operators	domain	domain	domain	domain	domain
Semantics	domains	graph	model	model	model
Proof System	axiomatic	automated	axiomatic	reach	axiomatic
Modular Abstraction	functions	nodes	layers	none	charts
Development Tech	none	none	refinement	none	none
Automated Support	executable	executable	executable	none	none
Criticality	none	none	none	none	none
Failure	detected	none	none	none	none
Concurrency	none	none	true	interleaved	true
Multiple Clocks	none	available	none	none	none
Communication	none	none	synchronous	synchronous	synchronous
Resources	none	none	none	none	none
Reference	[Har88]	[HPO88]	[AK86]	[GMMP89]	[HRR89]

6.2 Problems With Existing Languages

6.2.1 Methods

A formal method is a development technique which uses a formal language and an associated proof system. For example, constructive *refinement* is a method which translates a high level specification into an implementation by applying a number of rewrite rules. Another example is the *divide and conquer* method of compositional proof systems – a specification is subdivided, and each section is proven correct independently.

Most languages we have surveyed have no associated method, and the problem of using them is therefore analogous to trying to learn a foreign language without a phrase book or teacher. Some small examples are often given in introductory papers, but there is rarely a discussion of how specifications and designs are arrived at, or what to do with them once they are produced.

Temporal logic languages have an implicit basic method, where system behaviour is specified in the logic, and each requirement is then proven as a consequence of the system specification. But this method is *too* basic – in general it is not compositional, and without a method for producing a design from the specification, it is of limited benefit.

Process algebras define process equivalence relations of varying 'strength'. The *bisimulation* method used in CCS [Mil80,89], SCCS [Mil82], and wTCCS [TM89], is an algorithm which constructs an equivalence relation using a functional closure of the operational semantics of the algebra. However, equivalence is a very strong property, and there are a number of other weaker properties that we might wish to prove. For example, in wTCCS we might wish to prove that a process is always capable of performing an action at a specified time; this is not easily achieved using equivalence relations. Process logics [Mil89] [Hol89] may be used to prove weaker properties in the untimed process algebras, but there are no process logics for wTCCS or SCCS.

Real-time languages inherit some methods from programming, but most of these are informal and unstructured. Those methods that are structured are often controversial (for example, design by abstract data types [Wik87], and object oriented techniques [Atk90]). Examples of methods which could be defined for formal programming languages include proving translations correct, finding worst case execution times, and proving termination.

Providing methods would increase acceptance of formal languages by the general computing community. Providing small examples instead of explicit methods is not enough.

6.2.2 Automated Support

Automated support for formal system development is a potential answer to the criticism that formal languages are too complex to use without making syntactic and semantic mistakes. Computers may be used for checking syntax, and for modelling semantics.

Automated tools are common in formal languages, but usually in a restricted form. RT-ASLAN [Ak86] produces proof obligations (but does not attempt to prove them), TEMPURA [Mos86] uses an interpreter to find states that satisfy assertions (but cannot prove consequence), and LUSTRE produces behaviour graphs that conform to assertions (but needs to recompile the entire program when any single assertion is changed). A more complete tool would allow us to ask "What if ?" questions – "What if I increase this deadline to 20ms ?", "What if I do not allow this event to occur while this other one is occurring ?", etc.

An unexploited advantage of process algebras is their amenity to automation. The operational semantic rules may be translated into function definitions, and algorithms based on graph theory used on the behaviour graphs constructed from the semantics. The Concurrency Workbench software [CPW83] is an example of automated support for a process algebra, it constructs bisimulation relations for CCS agents; it could easily be extended to prove other properties.

Automated theorem proving is becoming faster, and covering a wider range of proof systems. There are new techniques which may play a significant rôle in increasing the speed of deductive reasoning. For example, *simulated annealing* techniques can be efficient in searching large problem spaces [Tch90], and may be able to find proofs more quickly than conventional techniques. Perhaps we may combine these new, faster, techniques with an interactive environment which encourages more problem exploration and trial and error experimentation. However, at present automated theorem proving is limited to small examples, and have limited usability.

6.2.3 Explicit Support For Criticality And Fault Tolerance

In none of the languages we have surveyed has there been a mechanism for specifying the *criticality* of a requirement. In safety-critical systems we need to be able to reason about the trade-off between the functional and temporal correctness of the system, and this is not possible with any of these languages. Compensation functions [SZ90] [Bur90] is one technique which may provide an answer to this problem, by encoding value and damage metrics for event occurrences.

Failure semantics are rarely provided in formal development languages. It is often assumed that a specification describes the perfect behaviour of a system, and that the design will reflect this

behaviour. In reality this assumption cannot be supported; often the environment in which a system is operating is faulty, and those faults will manifest themselves within the system. Fault detection and tolerance have been investigated within a process algebra framework [Nor90] [Pra84], but a common problem is representing the 'lack of willingness' of a system to behave in the required way. For example, in CSP [Nor90], the semantics cannot model a process which non-deterministically decides to refuse to participate in an event. In CCS [Pra84] faults are modelled by unstable agents, but the transiens of faults cannot be captured.

6.2.4 The Specification And Design Dichotomy

Most formal development languages are intended for either specification or design, rarely both. It is argued that this is because specifications and designs are very different, the former are abstract and the latter are concrete [Sho83]. However, there are two problems with this view:

- Because specifications are often *translated* into designs, there is an increased probability of introducing errors into the development process.
- The user needs to learn two different languages and proof techniques, as well as the translation process itself.

In contradiction, it is argued that we rarely write purely abstract specifications, or concrete designs; often we wish to include a mixture of both [Mor90]. This is particularly true of real-time systems, where the temporal correctness of a system is dependent upon implementation details. For example, a requirement might be:

> "Task T should terminate before task K if T is running on an otherwise unloaded processor"

The ordering constraint on the two tasks is abstract but the processor loading is concrete. In most of the languages we have surveyed, this requirement could not have been expressed (one language which would have been sufficient is CSR [GL89] which has primitives for task allocation).

In principle we can solve these problems by using a single language for all of the development process, including specification, design, and even implementation. With a single language we may perform *refinement* which is is a process that translates a specification into an implementation by applying of a number of rewrite rules. Refinement *is* possible between different specification and implementation languages [McD89], but the process is more complex. On the other hand, a single language that is capable of dealing with all of these problems will itself be complex and it may be necessary to make compromises in the language to address all the relevant issues.

There are no complete (both abstract and concrete) languages for real-time system development, but such a language may prove particularly suited to the problem area. The temporal effects of design and implementation issues could be reasoned about in the specification.

6.2.5 A Summary Of Problems

The problems discussed above are summarised in the following list:

- Lack of methods: very few languages have associated methods, this reduces their acceptance.
- Automated support: automated tools are often incomplete and restricted. In particular, the user cannot ask "What if ?" questions.
- Criticality and fault tolerance: there is no support for criticality, and very little for fault tolerance. The lack of failure semantics reflects the assumption that the environment will always behave correctly.
- Specification and design dichotomy: most languages are intended for specification or design, but in real-time systems the abstract and concrete issues often cannot be separated.

In the final section we suggest some possible solutions to these problems.

6.3 Expressibility

The ease with which we may reason about a type of requirement is as important as the ability to represent it. For example, we may represent multiple clocks in wTCCS by defining a family of clock processes, and composing them in parallel with the system that needs to access them:

$clock_n =_{def} (d[n])tick_n.clock_n$ where $d[n]$ is the speed of clock n.

However, if no process is listening to a particular clock process for any period, then that clock will stop. This problem can be solved by adding processes which absorb the 'unheard' ticks, but the system then becomes larger and more complex, and proving properties about it also becomes more difficult.

In contrast, the language TAM [SZ90] has a primitive for defining multiple clocks. Every requirement may refer to a different clock, and the result is a parsimonious language in which reasoning about multiple clocks is straight-forward.

Both languages are equally expressive (in terms of multiple fixed-speed clocks), but TAM is easier to work with.

Therefore, we need to consider two metrics, the first measures the expressibility of a language, the second measures how easily certain concepts may be reasoned about. For the expressibility metric there are a number of techniques we may use, and below we examine one based on category theory. However, with the 'ease of use' metric the task is more difficult because we are dealing with subjective judgement.

An Expressibility Metric Based On Category Theory

It may be possible to define a metric of expressibility using category theory [BW90b] [RB88] [Mac71]. A formal language may be represented as a category, with a proof system defining the morphisms, and equivalence between categories demonstrating equivalence in expressive power.

For example, the following table describes the correlation between weak choice in wTCCS, and disjunction in the propositional logic – both operators form coproducts in their respective categories.

Category	Objects	Morphisms	coproduct
wTCCS	process	simulation	weak choice
propositional logic	formula	proof	disjunction

Here *proof* is the usual Gentzen deductive proof system for propositional logic[3] [Gen36], and *simulation* is a relation on processes which captures the intuition that:

> P is a simulation of Q if P can do at least as much as Q can.

This correlation demonstrates that both languages are equally expressive in terms of determinant choice in behaviour (relative to the proof systems involved). By finding similar constructions for other operators, the relative expressive power of each language may be measured.

Categorical representation provides other benefits; automated tools may find constructions which form proofs of confluence, termination, and equivalence.

[3]The proof that disjunction is a coproduct of the category of propositional logic is as follows. The canonical injections into the disjunction are formed from the rule '$P \vdash P \vee Q$', giving the binary coproduct cocone:

$$P \longrightarrow P \vee Q \longleftarrow Q$$

The universal property holds because any formulae which may be derived from both P and Q separately must be in conjunction, or be valid, in both P and Q. Therefore, disjunction elimination can be used to derive it from the coproduct.

The proof that weak choice is a coproduct in the category of wTCCS with simulation morphisms is similar.

7 Terminology

There are a number of terms which have not been used consistently in the literature that has been surveyed in this document. In some cases this is due to a certain confusion amongst all authors as to the most appropriate meaning of a particular term, in some instances the usual meaning of the term is not quite applicable. Therefore, this section outlines the meaning of each of these terms as they have been used in *this* document.

Asynchronous communication

An asynchronous **send** communication involves no second party; a message is sent to a communications medium along with the identification of an intended receiver, and the sending process continues with its next task. To ensure that a message reaches the correct destination, a communication protocol must be established.

An asynchronous **receive** involves a receiving process taking a message from the communications medium, it may choose to do this at any time, or not at all.

The clear advantage of asynchronous communication is that of non-blocking, a process does not depend upon the correct, and timely, operation of a second process. The disadvantage comes from the extra complexity of adding a protocol to ensure correct communication.

Asynchronous computation

An asynchronous computation process may proceed at any speed, its progress is not dictated by a clock, or any other process (excepting voluntary communication).

Axiomatisation

An axiomatisation of a formal language involves the denotation of a number of axioms (or axiom schemas), and inference rules, which model the semantics. It is well known that there are some semantics which can not be completely axiomatised (e.g. the behaviour of the integer numbers [Gal87]).

Closed Formula

A formula is closed if it contains no unquantified variables. In logics this implies that all variables must be bound to truth values and, in the lambda calculus, all functions must be applied to the desired number of values etc.

Complete

A formal system is complete if the equality in the semantics is respected by the equality in the syntactic proof system.

Compositionality

Compositionality in semantics enables the user to prove properties of 'chunks' of his specification without having to specify the complete system.

Consistent

A logic system is consistent if none of the formula that evaluate to 'true' also evaluate to 'false' and vice-versa. An inconsistent logic enables the user to prove any theorem true – if the impossible is made possible (something is true and not true), then all things are possible (formally RAA is used on a negation of the desired theorem).

Continuous function

A continuous function on a domain preserves the least upper bounds and greatest lower bounds i.e. the ordering of objects is strictly preserved by the function. More generally, a continuous function on a complete partial ordering preserves upper bounds of directed subsets. The important point is that all continuous functions on domains have least fixed points (ensuring termination of closure functions).

Continuous transformation

A continuous transformation of a topological structure preserves the open sets of that structure (open sets are analogous to directed sets in cpos, as are continuous transformations analogous to continuous functions). We can intuitively think of a continuous transformation as a 'stretch' or 'bend' in a structure, rather than a 'break' or 'tear'.

Coproduct

A categorical coproduct is a (co)limit which is formed from injection morphisms from a number of objects. The universal property which defines a coproduct is a function mapping other candidate coproducts to morphisms from the coproduct to the candidates.

In process algebra categories, coproducts represent choice or alternate processes.

Deadlock

A process is in deadlock when it requires some resource (communication or physical resource), but will never gain that resource. Synchronous communication mechanisms may give rise to mutual deadlock, asynchronous communication (disregarding any possible protocol) cannot give rise to deadlock.

Decidability

A formal language is decidable with respect to a classification system when there exists an algorithm which can determine the class of any sentence in that system in *finite* time. In the case of propositional logic, the classification is is usually based on truth value, and there exist a number of decision procedures.

Divergence

A process is divergent whenever it is *possible* that the process may reach a state whereby it continues to be active, but never produces a result (this is also known as 'livelock'). A process is *strongly* divergent if it always reaches a state where it is divergent.

Domain

A domain is a set and a partial ordering whereby all pairs of objects in the set have a least upper bound and greatest lower bound within the set. Domains are used in denotational semantics as types or value sets.

Fairness

A process is fair if, given an infinite execution, every optional computation is executed a number of times proportional to the number of times the choice is enabled. This is known as *strong* fairness.

Justice

A process is just if, for every option that is infinitely enabled (assuming an infinite computation), that computation is chosen at least once (avoiding infinite starvation).

Monotonic function

A function is monotonic with respect to some partial ordering if application preserves the ordering, i.e. given an ordering $x \geq y$ then $f(x) \geq f(y)$ for all x, y in the ordering means that f is monotonic.

Normal form

A normal form is a representational structure which is capable of expressing all sentences in a formal system (with respect to some equivalence relation). For example, in CCS [Mil80,89], there is a normal form consisting of the fixpoint operator, prefix, and summation.

Planar graph

A graph is planar on a surface if it can be drawn on that surface with no intersecting edges. In Euclidean space, the maximal complete graph is K_4 (a graph with all four nodes connected to all other nodes).

Product

A categorical product is the dual concept of the coproduct. It is formed from an object with projection functions to those objects. The universal property of a product maps candidate products onto morphisms from the candidate to the product.

In many process algebra categories, the product froms the parallel composition of processes.

Proof system

A proof system is a method of proving equivalence in the syntax of a formal system. In process algebra it is usually represented by an equational theory, in logics, a deductive apparatus is provided.

Refinement

Refinement is a process of transforming a specification into executable code by a series of semantic preserving steps. The transformation steps are *sound* with respect to the program semantics, and can be used to guarantee a correct implementation.

Sort

A sort is an unstructured collection of primitive values. Unlike a type, a sort does not have operators or constructors, simply a method of determination. Sorts are used as primitive type constructors.

Sound

A proof system is sound with respect to a defined semantic equivalence if it respects the semantic equivalence, i.e. whenever two formula are proved equal in the proof system, they must be able to be proved equal in the semantics.

Synchronous communication

Synchronous communication is a point-to-point communication method that needs simultaneous involvement of both the sender and receiver. Both processes block until the communication is completed.

Synchronous computation

A process in synchronous computation proceeds in lock-step with a clock. This means that every action must have a maximum duration (or one clock) tick. Theoretically this form of computation has some very pleasant properties.

Tableaux

A semantic tableaux method is a decidability procedure which usually includes heuristics to guide the procedure to a conclusion. A number of tableaux methods exist for restricted forms of predicate logic, and modal logics.

8 Annotated Bibliography

8.1 Introduction

This section presents a bibliography of references used in the review, and of general interest. The following acronyms are used:

- LNCS – Springer-Verlag Lecture Notes In Computer Science.
- GTM – Springer-Verlag Graduate Texts In Mathematics.
- PHI – Prentice-Hall International Series In Computer Science.
- LFCS – Laboratory for Foundations of Computer Science, Edinburgh University.
- PRG – Programming Research Group, Oxford University.
- UCCL – University of Cambridge Computer Laboratory.
- CACM – Communications Of The Association Of Computing Machinery.

8.2 Bibliography

[Atk90] M. C. Atkins, "Implementation Techniques For Object-Oriented Systems", D.Phil. Thesis, University Of York Dept. Computer Science. YCST90/01. 1990.

[AK86] B. Auernheimer. R. Kemmerer, "RT-ASLAN: A Specification Language For Real-Time Systems", Transactions On Software Engineering, Vol 12(9). Sep 1986.

> RT-ASLAN is an extension of the ASLAN programming language. It is a language for multi-level refinement specification with shared variable communication and RPC's as primitives. The RT-ASLAN compiler produces proof obligations between levels that should be proved by the user to justify refinement steps.

[Bac87] R. Backhouse, "The Martin Löf Theory Of Types, A Computing Scientist's Perspective". Proceedings BCS-FACS Seminar Imperial College. September 1987.

> Including constructive proofs, rule construction (axiomatisation of types), and program development. Backhouse has a different style from Nordstrom et.al. particularly in assumption scoping, and a number of people have commented on the fact that the resulting derivations are hard to read. The comments by Backhouse on the rôle of type-theory in computer science are particularly interesting, reflecting the history of formal program construction.

[Bac88] R. J. R. Back, "A Calculus Of Refinements for Program Derivations", Acta-Informatica, 25, p593-624. 1988.

[Bac89] R. Backhouse, "Do-It-Yourself Type Theory". BCS-FACS Journal, Vol.1, No.1. 1989.

A somewhat potted version of the above with fewer examples. The same stylistic comments apply.

[Bak80] J. de Bakker, "Mathematical Theory Of Program Correctness", PHI 1980.

Expands Hoare and Floyd's work on verification triples to provide a mathematical semantics for imperative programming languages. Uses domain theory (see Stoy below) and fixpoint semantics for well-termination. The book is self contained, and includes much work on weakest preconditions etc.

[BCDE90] E. Best, L. Cherkasova, J. Desel, J. Esparza, "Traps, Free Choice And Home States", Informatik-Berichte Nr5/90, University Of Hildersheim. 1990.

[BFGGO89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, R. Owens, "METATEM: A Framework For Programming In Temporal Logic", Department Of Computer Science University Of Manchester, Department Of Computing Imperial College, Draft Report. 1989.

[Bro87] M. Broy, "Semantics Of Finite And Infinite Networks Of Concurrent Communicating Processes", Distributed Computing 2, Springer-Verlag. 1987.

A process algebra is defined in which loosely coupled processes may communicate over streams. No timing aspects are included.

[Bur90] A. Burns, "The Application Of Utility Functions In Real-Time System Design And Scheduling", Department Of Computer Science Internal Report, 1990.

[BV84] E. Best, K. Voss, "Free Choice Systems Have Home States", Acta-Informatica 21, 89-100. 1984.

[BW89] R. J. R. Back, J Wright, "Refinement Calculus, part I: Sequential Programs", REX, Workshop For Refinement Of Distributed Systems, Nijmegen. 1989.

[BW90a] A. Burns, A. Wellings, "Real-Time Systems And Their Programming Languages", Addison-Wesley. 1990.

An undergraduate bible for real-time systems. Contains introductory material. Includes a substantial case study in Ada. Languages used are mainly Ada, Occam, and Modula-2. Discussion on Virtual-Nodes (pre-partitioning distributed system design). Covers fault-tolerance, distribution, efficiency issues, concurrency.

[BW90b] M. Barr, C. Wells, "Category Theory For Computer Scientists", PHI. 1990.

This is THE introduction for computer scientists. Many examples are based on computer science including: programming language semantics, program verification, domain theory, etc. A basic introduction to graph theory and function theory builds into a good grounding. Includes adjunctions, topos theory, sheaf theory, Cartesian closed categories and the lambda calculus, and an introduction to 'sketch theory' (a Barr and Wells construction).

[BW90c] R. J. R. Back, J Wright, "Refinement Concepts Formalised In Higher Order Logic", BCS-FACS, Vol 2, No.3. 1990.

[CM84] W. Clocksin, C. Mellish, "Programming In Prolog", Springer-Verlag. 1984

Introductory level to PROLOG. Edinburgh syntax (VAX orientation) includes examples for natural language parsing.

[CPS89] R. Cleaveland, J. Parrow, B. Steffen, "The Concurrency Workbench", University Of Edinburgh Technical Report LFCS-89-83. Aug 89.

The concurrency workbench is an automated toolset for CCS.

[Dij76] E. Dijkstra, "A Discipline Of Programming", Prentice-Hall Inc. 1976.

Dijkstra has produced one of computing sciences most famous, and controversial, books. This work expounds the virtue of structured programming with a mathematical rigour. Dijkstra will brook no criticism, and the book is full of very strong condemnation of programming practices that some can not condemn. The now-famous weakest precondition calculus is the main theme of this book.

[DJM89] M. Donner, D. Jameson. W. Moran, "Events: A Structuring Mechanism For A Real-Time Runtime System", Technical Publication IBM T.J. Watson Research Center, Yorktown Heights, New York. Sep 1989.

[DS89] J. Davies, S. Schneider, "An Introduction To Timed-CSP", PRG. 1989.

Assumes a working knowledge of CSP (see Hoare below), discusses the differences between the timed and untimed theory. Includes examples of timeout operators, trace semantics and specification satisfaction.

[Fid87] C.J. Fidge, "A LISP Implementation of the Model for CSP", Software Practice and Experience Vol18(10). 1987.

Provides an LISP based interpreter for embedding and executing CSP processes. Hoare provides a similar (but sketchy) system in his book (see below), but this system is much more complete, and provides an interesting introduction to CSP.

[FB86] M. Fisher, H. Barringer, "Programming Logics : A Short Survey", Manchester University Technical Report. Nov 1986.

Includes modal logic, tense logic, dynamic logic, temporal logic, and process logic.

[FP90] C.J. Fidge, M. Pilling, "Specification Languages for Distributed Real-Time Systems", University Of Queensland Technical Report 156. Mar 1990.

This review looks at 25 specification languages, including some non real-time languages such as Z. Reference list is very useful.

[Gab89] D. Gabbay, "The Declarative Past and Imperative Future", Imperial College Technical Report. Jan 1989.

The logic presented allows reasoning about past and future truths. Includes fixed point and next/last operators. Formulae of the form $A \rightarrow B$ are interpreted as assertion/obligation pairs, if A is true now, then B must become true. This form can be viewed as a programming language (similar to TEMPURA, see Moszkowski below).

[Gal87] A. Galton (ed), "Temporal Logics And Their Applications", Academic Press. 1987.

Contains papers by: Howard Barringer on compositional semantics of concurrent systems, Roger Hale on temporal logic programming (see Moszkowski and TEMPURA), and Dov Gabbay on modal and temporal reasoning. A selection of papers that discuss many relevant issues in temporal logic. Barringer's compositional semantics is very similar to Milner's process logic and provides an insight into the complexity of compositional semantics.

[GL89] R. Gerber, I. Lee, "Communicating Shared Resources: A Model For Distributed Real-Time Systems", Proceedings IEEE conference on Real-Time. 1989.

Based on a CSP-like language, Communicating Shared Resources (CSR), the important difference is that resources are considered important from the specification onwards. CSR encodes priorities as a primitive notion. A robot example is given, denotational semantics in the form of Histories are provided.

[GMM89] C. Ghezzi, D. Mandrioli, A. Morzenti, "TRIO : A Logic Language For Executable Specifications Of Real-Time Systems", Politechnico di Milano Technical Report TR 89-006. 1989.

The logic language TRIO is introduced with a number of small examples. Theoretical issues such as decidability are discussed, and a general tableaux method is presented.

[GMMP89] C. Ghezzi, D. Mandrioli, S. Morasca, M. Pezze, "A General Way To Put Time Into Petri Nets", Politechnico di Milano. CACM Volx, No.x. 1989.

Introduces TB Nets with timing properties on each transition which filter any firings that are 'stale'. ER nets are also discussed and suggested as a basis for describing other formalisms. There are no sizable examples in this paper, and therefore no properties of systems proved.

[GMW79] M. Gordon, R. Milner, C. Wadsworth, "Edinburgh LCF", LNCS-78. 1979.

LCF (logic for computable functions) is a system for doing formal proofs interactively. It introduces the programming language ML (later SML - see below). Theorys are built in ML terms, and theorems are proved in a goal-searching manner. LCF is particularly interesting because of the type inference system used, one of the first true polymorphic type systems to be implemented.

[GT97] J.A. Goguen, J. Tardo, "An Introduction To OBJ: A Language For Writing And Testing Software Specifications", Proceedings IEEE Conference On The Specification Of Computer Software, IEEE. 1979

[Hal88] R. Hale, "Using Temporal Logic for Prototyping : The Design Of a Lift Controller", University Of Cambridge Technical Note. 1988.

Uses the temporal logic ITL to produce a specification of a list control system. The language TEMPURA is used (see Moszkowski below) for prototyping. Roger Hale did the original work in building the TEMPURA interpreter, and this paper gives an interesting insight into the possible uses of the language.

[Har87] D. Harel, "Statecharts : A Visual Formalism For Complex Systems", Science Of Computer Programming, 8(3):231-274. 1987

[Har88] D. Harrison, "Functional Real-Time Programming: The Language RUTH and its Semantics", University Of Stirling Ph.D. TR 59. 1988.

Includes a lazy evaluation functional programming language RUTH with semantics using denotational herringbone domains. Programs can be analysed for temporal correctness. Example programs include 'minesweep' a real-time game.

[Hay87] I. Hayes (ed), "Specification Case Studies", PHI. 1987.

Includes an introduction to the set-based specification language Z, with around 12 large case studies by members of the Oxford PRG group. This book shows you how to write Z well, and very poorly.

[HC68] G.E. Hughes, M.J. Cresswell, "An Introduction To Modal Logic", University Paperbacks. 1968.

A classic in modal logic. Includes s4,s5 and T systems, non standard systems. Introductory level material and upwards.

[Hen88] M. Hennessey, "Algebraic Theory Of Processes", MIT Press. 1988.

An exposition of the underlying mathematical theory of process algebra, including initial semantics, equational theories and soundness/ completeness. Full abstraction in semantics. Testing equivalences, acceptance tress. The book provides a theory in which to discuss all process algebra, and discusses CCS in these terms. It provides a self contained book on the subject.

[HK90] R. P. Hopkins, M. Koutny, "Modelling Concurrent Systems With Dynamic Communication Structure", University Of Newcastle upon Tyne Technical Report 282. April 1989.

[HKR89] R.P. Hopkins, M. Koutny, B. Randell, "Some Results On Dynamically Structured Communicating Systems", University Of Newcastle-Upon-Tyne. Jan 1989.

Includes both synchronous and asynchronous communication in the same process algebra. The processes exhibit dynamic structure, i.e. communication links may be created and destroyed during system execution.

[Hoa69] C.A.R. Hoare, "An Axiomatic Basis For Computer Programming", Communications Of The ACM, 12, p576-580. 1969.

[Hoa78] C.A.R. Hoare, "Communicating Sequential Processes", Comm ACM 21(8), 666-677 (1978).

The original version of the CSP language. Essentially an extension to Dijstra's guarded command language [Dij76].

[Hoa85] C.A.R. Hoare, "Communicating Sequential Processes", PHI. 1985.

An introduction to the theory of CSP with numerous examples (including the ubiquitous vending machine). The theory is introduced very slowly and includes an introduction to the mathematical theory of processes. This is a culmination of many years work and it is instructive to read the fore-runners in the form of papers published between 1978 and 1985. The theory has changed considerably during that time, and the rational behind some design decisions are particularly interesting.

[Hol89] S. Holmström, "A Refinement For Specifications In Hennessey-Milner Logic With Recusrsion", BCS-FACS, Vol 1,No.3. 1989.

[HPO88] N. Halbwachs, D. Pilaud, F. Oubdesselam, "Specifying, Programming and Verifying Real-Time Systems Using a Synchronous Declarative Language", Project Spectre, Laboratoire de Genie Informatique de Grenoble. Jul 1988.

This paper introduces LUSTRE, a high level real-time programming language with a formally defined semantics. LUSTRE programs contain safety assertions as part of the code. The assertions are checked at compile time. Modular proofs are possible by giving assertions on the input and output parameters of nodes (functions).

[HRR89] J. Hooman, S. Ramesh, W.P. de Roever, "A Compositional Semantics For Statecharts", Eindhoven University, The Netherlands. 1989.

Introduces statecharts (from original work by Harel see above), which provide a compositional graphic language for event logic. The semantics presented are denotational and compositional and would allow reasoning about safety and liveness. Statecharts are constructed using 'and' and 'or' joins, with links being connected to provide a flow of control from state to state.

[INM88] INMOS Ltd, "occam 2 Reference Manual", PHI. 1988.

Standard reference manual for the occam programming language.

[JG88] M. Joseph, A. Goswami, "Formal Description Of real-Time Systems: A Review", University Of Warwick Technical Report 129. Aug 1988.

Discusses program correctness and verification, a few real-time formal languages (RTL, CSP, Koyman's Temporal Logic) and some implementation languages, including LUSTRE, ESTEREL, and SIGNAL.

[Jef89] H. Jefing, "Process Simulation And Refinement", BCS-FACS, Vol 1, No.3. 1989.

[JM86] F. Jahanian, A. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", IEEE Transactions on Software Engineering Vol12, No.9. 1986.

A event logic based specification system with a two-tiered representation of actions and events. The real time logic (RTL) is introduced, and some examples of proofs are included. The language is designed to express proofs of safety in real time systems. This is the original paper on RTL and contains some interesting notes on the relation between RTL and fault trees. Also, RTL was later used in the ORE programming language project.

[Jos88] M. Jopseph (ed), "Formal Techniques in Real-Time and Fault-Tolerant Systems", LNCS 331 (Proceedings of a Symposium – Warwick). Sep 88.

Contains a number of formalisms, including DRTL (distributed RTL), a paper by Pnueli on Temporal Logics and real-time systems, and Zwarico's Timed Acceptances.

[KMS84] J. Kramer, J. Magee, M. Sloman, "The Conic Programming Language Version 2.4", Imperial College Technical Report. Oct 84.

[KST90] M. Z. Kwaitkowska, M. W. Shields, R. M. Thomas (eds), "Semantics For Concurrency", Springer-Verlag, BCS-FACS Workshop, Leicester. July 1990.

[Lam80] L. Lamport, " "Sometime" is Sometimes "Not Never" ", Proc 7th Symp. on Principles Of Programming Languages 174-185. 1980.

[Li83] W. Li, "An Operational Approach To Semantics And Translation for Concurrent Programming Languages", University Of Edinburgh Ph.D. Thesis CST-20-83. 1983.

Discusses operational semantics in general and includes specific semantics for Ada (a subset), CCS, and CSP. Includes an operational translation theory. A complete, basic, exposition of operational semantics.

[Mac71] S. Mac Lane, "Categories For The Working Mathematician", GTM 5, 1971.

THE category theory book for mathematicians. Definitely NOT THE category theory book for computer scientists. Very brief introduction to the rational of category theory with emphasis on the relationship between category theory and mathematical foundational theory.

[McD89] J. McDermid, "The Theory And Practice Of Refinement", Butterworths. 1989.

Includes case studies on refinement in Z, including work on interactive systems, hierarchical systems, and the IBM CICS system. Many examples of Z, and how it should be written.

[Mil80] R. Milner, "A Calculus Of Communicating Systems", LNCS 90. 1980.

The original work on CCS, contains no reference to bisimulation, or a complete axiomatisation of the language. Interesting discussion on process algebra rational.

[Mil82a] R. Milner, "A Synchronous Calculus For Communicating Systems", University Of Edinburgh Internal Report CSR-xx-82.

Milner presents SCCS, a synchronous process algebra. SCCS is particularly elegant, with only four operational semantic rules, and five syntactic operators. The paper derives an asynchronous calculus as a sub theory of SCCS. Later (in Communication And Concurrency) Milner shows how CCS is also a sub theory of SCCS.

[Mil78] R. Milner, "Algebras For Communicating Systems", University Of Edinburgh Internal Report CSR-25-78. 1978.

An early paper on the route to CCS (see below). Discusses the homomorphism from an algebra of nets to synchronisation tress (later agents to trees).

[Mil89] R. Milner, "Communication And Concurrency", PHI. 1989.

Introduces the process algebra CCS (calculus of communicating systems). This is the culmination of a number of years work, and includes the (weak) bisimulation method suggested by Park after the original book was published (see below). This book also introduces Hennessey-Milner process logic, and SCCS. There are some very interesting chapters on confluence and determinacy.

[Mil82b] R. Milner, "A Finite Delay Operator in Synchronous CCS", Technical Report (LFCS). May 82.

Milner introduces a finite delay operator into SCCS and finds a new congruence between processes limited to this delay operator.

[Mit85] K. Mitchell, "An Introduction to PFL", *Informal note from Mitchell.* Oct 1985.

PFL (parallel functional language) pronounced 'piffle', is a set of SML routines for executing CCS process descriptions. In effect, this enables the user to execute CCS specifications directly, although the final result is very slow to execute. PFL is automatically released as part of the SML language from Edinburgh.

[MM89] A. Mili, F. Mili, "Relational Heuristics For Programming: Advances and Perspectives", Faculty Of Sciences of Tunis Belvedere (notes). Nov 89.

[Mor90] C. Morgan, "Programming From Specifications". PHI. 1990.

A refinement calculus for transformational programs. Includes introductions to the predicate calculus, Dijkstra's weakest preconditions, and the single language approach. This book is an introduction to the work by Morgan, Robinson and Gardiner (see below) on the refinement calculus developed at Oxford. It is self contained with a number of small and large examples of programs refined from predicate calculus specifications. A point worth noting is that the calculus does not guarantee correct implementations, and may produce ill-typed programs. In all, the earlier work (see below) is probably a better guide to the calculus, although there are some interesting comments in the book regarding syntactic issues in transformational programs in general.

[Mor87] J. M. Morris, "A Theoretical Basis For Stepwise Refinement And The Programming Calculus", Science Of Computer Programming, 9, p287-306. 1987.

[Mos86] B. Moszkowski, "Executing Temporal Logic Programs", Cambridge University Press. 1986.

Moszkowski presents an interpreter for the specification language TEMPURA based on the interval temporal logic (ITL). Programs are formulae in the logic, and execution consists of searching for states which satisfy the formulae. TEMPURA contains a 'chop' operator, which is used to formalise the concept of sequential control flow.

[MP83] Z. Manna, A. Pnueli, "Verification Of Concurrent Programs: a Temporal Proof System". Technical Report STAN-CS-81-836, Dept. Comp. Sci. Stanford University. Jun 81.

[MPW89] R. Milner, J. Parrow, D. Walker "A Calculus Of Mobile Processes, I and II", University Of Edinburgh Technical Reports ECS-LFCS-89-85/86. 1989.

The Calculus Of Communicating Systems is extended to allow the passing of channel names along channels. This extension allows higher-order processes in CCS, and the modelling of resource management, permissions, and higher-order functional computation.

[MRG90] C. Morgan, K. Robinson, P. Gardiner, "On The Refinement Calculus", PRG-70, Oct 1988.

The original report on the refinement calculus developed at Oxford. It consists of a number or papers on program refinement, datatype refinement, issues to do with 'miracles', and the formal semantics of refinement. It also presents a good introduction discussing the original motivation.

[NDGO85] V. Nguyen, A. Demers, D. Gries, S. Owicki, "A Model And Temporal Proof System For Networks Of Processes", Cornell University Technical Report 85-690. Jun 85.

A general model which can describe asynchronously and synchronously communicating networks of processes is presented. The proof system is compositional.

[Nor90] J. Nordahl, "Dependability in a Process Algebraic Framework", Department Of Computer science, Technical University Of Denmark. April 1990.

[NPS90] B. Nordstrom, K. Petersson, J. Smith, "Programming In Martin Löf Type Theory". Oxford Science Publications. 1990.

A clear, basic, description of Martin Löf type-theory with numerous examples of program development and verification. Includes sections on sets as propositions, monomorphic and polymorphic sets, and a subset theory (which describes program optimisation problems). Because Martin Löf type theory is based on a lazy-evaluation functional language, it is only suitable for transformational programs, but it does give a general overview of constructive proofs of correctness which might be extended to reactive systems generally.

[Ost89] J. Ostroff, "Temporal Logic For Real-Time Systems", Research Studies Press, 1989.

Covers specifically discrete event process specification, design and implementation. The language Real Time Temporal Logic (RTTL) is used (based on Manna and Pnueli's temporal logic system) for specification. Extended State Machines (hierarchical, concurrent state machines) provide semantics, and CONIC is used as the implementation language. Case studies are included. The book was originally Ostroff's thesis, and contains a lot of interesting detail on temporal logic generally.

[Pet81] J. Peterson, "Petri Net Theory And The Modelling Of Systems", Prentice-Hall, 1981.

A good introduction to petri-net theory with a clear exposition of the mathematical semantics. A lot of examples, discusses decidability and reachability (graphs). Also some common extensions are covered (inverse transitions etc).

[Pnu87] A. Pnueli, "Applications Of Temporal Logic To Reactive Systems", Proceedings *Logic and it's Application to Computing Science*, University Of Newcastle-Upon-Tyne. Sep 87.

Paper reproduced from LNCS 224 (1986). Contains Pnueli's extended temporal logic with fixed points. Includes examples of communicating systems, formalising shared variables. Discusses fairness and justice.

[Pra84] K.V.S. Prasad, "Specification And Proof of a Simple Fault Tolerant System in CCS", University Of Edinburgh Internal Report CSR-178-84. Dec 1984.

Assumes all processes are fail-stop, and introduces instability for faults. It would be greatly simplified by using a timed process algebra with timeouts etc. Very detailed treatment.

[RB88] D.E. Rydeheard, R. Burstall, "Computational Category Theory", PHI. 1988.

Provides a coding for basic category theory constructs in SML. Includes algorithms for computing unification (as a co-limit). Introduction would be enhanced with a working knowledge of constructive type theory (category theory is largely constructive). Includes work on adjuncts, topos theory, sheaf theory etc. A good basic introduction.

[Ree88] G.M. Reed, "A Uniform Mathematical Theory For Real-Time Distributed Computing" Oxford University D.Phil. Thesis. 1988.

[Rei85] W. Reisig, "Petri Nets", European Association for Theoretical Computer Science, Springer-Verlag. 1985.

The classic exposition of Petri Nets. Introductory level through to the more 'interesting aspects'. A very extensive bibliography.

[RR86] G. M. Reed, A. W. Roscoe, "A Timed Model For Communicating Sequential Processes", LNCS 226, Proceedings Of ICALP'86. 1986.

[Sch90a] D. Scholefield, H. Zedan, "An Event Logic For Value Functions", Seminar Notes Presented At The 4th ACM SIGOPS Conference, Bologna Italy. Sep 1990.

[Sch90b] D. Scholefield, H. Zedan, "TAM: A Temporal Agent Model For Distributed Real-Time Systems", North-Holland, EUROMICRO '90 Workshop, August 1990.

[Scho90c] D. Scholefield, "TPL: A Process Logic For Timed-CCS", Work In Progress, University Of York, 1990.

[Sho83] M. Shooman, "Software Engineering", McGraw-Hill. 1983.

Including life-cycles, testing, complexity, specifications, and tools and techniques. It is not language specific. It provides an extensive bibliography.

[SL85] A. Udaya Shankar, S. Lam, "Time-Dependent Distributed Systems Proving Safety, Liveness And Real-Time Properties", University Of Texas Technical Report 85-24. Oct 85.

Processes have independent clocks which order the event occurrences. The formalism is aimed at modelling communication protocols, and the processes will represent send and receive stations. Safety and liveness proof rules are given for the behaviour of systems in sparse time.

[SMS90] "sci.maths.symbolic", USENET, Computer News. 1990.

[Spi88] J. M. Spivey, "Understanding Z", Cambridge Tracts In Theoretical Computer Science, Cambridge University Press, 1988.

[Sti82] C. Stirling, "Future Branching Time Logics and Fairness", University Of Edinburgh Internal Report CSR-105-82. Mar 1982.

Demonstrates that fairness is expressible in branching time logic. Discusses branching time criteria and models in general.

[Sti89] C. Stirling, "Local Model Checking in the Modal Mu-Calculus", LFCS-89-78. 1989.

Introduces Pratt and Kozen's Modal Mu-Calculus, based on a fixed-point operator with semantics containing sets of truth mappings. The paper provides a tableau method for the calculus.

[Sto77] J. Stoy, "The Scott-Strachey Approach To Programming Language Theory", MIT Press, 1977.

A seminal work. Introduces Dana Scott and Christopher Strachey's work on domain theory for programming language semantics. Provides a denotational semantics for the lambda calculus. A lot of detail on domains, (including power-domains, lifted domains etc).

[Str89] N. Strutt, "A Survey Of Formal Methods", Report From The Ministry Of Defence, ARE TM (AXC) 89001. Jan 1989.

[Sub89] P.A. Subrahmanyam, "What's in a Timing Discipline ?", Bell Labs, Holmdel, New Jersey. 1989.

Provides a general theory of timing using homomorphisms between event structures representing system events, and a *timing discipline* represented by another event structure (see Winskel below). Examples include hardware timing graphs.

[SZ90] D. Scholefield, H. S. M. Zedan, "TAM: Temporal Agent Model For Real-Time Distributed Systems", Proceedings Euromicro 90, Bologna. 1990.

[Tch90] K. Tchinscol, "Allocating Real-Time Tasks, An NP-Hard Problem Made Easy", University Of York, Dept. Computer Science, Internal Report. 1990.

[TM89] C. Tofts, F. Moller, "A Temporal Calculus Of Communicating Systems", LFCS 89-104.

Chris Tofts introduces the theory of weak Timed-CCS which uses two new action types into CCS, the asynchronous delay, and the bounded delay. Two types of choice operator are also included. The equational theory is surprisingly large, and soundness and completeness are only proved for a limited subset of the theory.

[Wik87] A. Wikström, "Functional Programming Using Standard ML", PHI. 1987.

A good, basic, introduction to SML including the formal operational semantics. However, it does not include any of the recent developments on Modules and Signatures. It also promotes a particular development strategy which is not always justified.

[Win80] G. Winskel, "Events In Computation", University Of Edinburgh Ph.D. Thesis CST-10-80. 1980.

Winskel invented *event structures* which provide a fundamental building block for causality structures such as Petri Nets and domains etc. Event structures are now used extensively in concurrent system semantics. The work is very complex but detailed.

[Win85a] G. Winskel, "Petri Nets, Algebras And Morphisms", UCCL 79. Sep 1985.

Provides a categorisation of Petri Nets and shows how they can then be related to CCS or CSP. The categorical constructions provide a way of investigating net behaviour. Importantly, the categorisation of nets, particularly the formalising of morphisms, provides a composition mechanism.

[Win85b] G. Winskel, "A Complete Proof System For CCS With Modal Assertions", UCCL 78. Sep 85.

A logic based on Hennessey-Milner logic is introduced, and a denotational semantics for SCCS. A proof system for SCCS is also presented.

[You82] S.J. Young, "Real-Time Languages Design And Development", Ellis-Harwood. 1982

[Zed90] H. Zedan (ed), "Real-Time Systems, Theory And Applications", Proceedings Of the BCS Real-Time Conference at The University Of York, North-Holland 1990.

Includes papers on Timed-CSP (Davies and Schneider), Ada Virtual Nodes (Hutcheon and Wellings), Temporal Logic For Prototyping (Hale), work on Interval Temporal Models (Holt), Milestones (Stone and Zedan). A mixture of theory and practice and a comprehensive overview of the state of the art in real-time systems.