A Temporal Logic for the Specification and Verification of Real-Time Systems

Yogesh Naik
University of Warwick

Thesis submitted in respect of the requirements for the degree of Doctor of Philosophy at the University of Warwick

March 1993

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of the University of Warwick, or any other university or institution of learning.

Abstract

The development of a product typically starts with the specification of the user's requirements and ends with the design of a system to meet those requirements. Traditional approaches to the specification and analysis of a system abstracted away from any notion of time at the specification level. However, for a real-time system the specification may include timing requirements. Many specification and verification methods for real-time systems are based on the assumption that time is discrete because the verification methods using it are significantly simpler than those using continuous time. Yet real-time systems operate in 'real' continuous time and their requirements are often specified using a continuous time model.

In this thesis we develop a temporal logic and proof methods for the specification and verification of a real-time system which can be applied irrespective of whether time is discrete, continuous or dense. The logic is based on the definition of the *next* operator as the next time point a change in state occurs or if no state change occurs then it is the time point obtained by incrementing the current time by one. We show that this definition of the *next* operator leads to a logic which is expressive enough for specifying real-time systems where continuous time is required, and in which the verification and proof methods developed for discrete time can be used. To demonstrate the applicability of the logic several varied examples including communication protocols and digital circuits are specified and their real-time properties proved. A compositional proof system which supports hierarchical development of programs is also developed for a real-time extension of a CSP-like language.

To
my son, wife and parents

Acknowledgements

I would like to thank Professor Mathai Joseph for giving me the opportunity to do a part-time Ph.D, and for detailed comments and advice on parts of this thesis.

I would also like to thank Liu Zhiming for detailed comments and many suggestions for improvements in the semantics of the logic, and Asis Goswami for many helpful discussions and constructive criticisms. I am also grateful to Professor Zhou ChaoChen for discussions at PRG, Oxford and while visiting Warwick University.

I am also very grateful to Professor Howard Barringer and Dr. Steve Matthews for examining the thesis and for detailed suggestions for improvements.

Finally, I would like to thank my family for making it possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of a product typically starts with the specification of the user's requirements and ends with the design of a system to meet those requirements. Extensive industrial experience in software engineering has shown that errors at the requirements specification level are not only common but have serious consequences in the development and maintenance of systems. At the same time, the increasing power and complexity of the systems being built have meant that they are difficult to understand, often do not meet the user's requirements and are prone to failures because of the faults introduced at the design and development stage. With complex systems affecting everyday life, it is essential that they are reliable. Failures in telecommunication networks and banking systems may cause monetary loss and inconvenience; but failures in critical applications such as nuclear reactors may lead to loss of human lives, environmental pollution and may have other long term irreversible consequences.

1.1 Specification and Verification of Real-time Systems

It is now widely accepted that a formal approach to system development is required to increase user confidence and the reliability of products. This is not surprising since designing and developing a system is a formal activity. Over the last twenty-five years, formal techniques have been developed for describing and analysing systems ranging from sequential to reactive ones. Sequential systems are also called transformational systems since they can be viewed as relations between their input and output states [Pnu86]. Several

formalisms such as Z [Hay87] and VDM [Jon86] have been developed to describe such systems. Reactive systems, on the other hand, cannot be viewed as a relation between their input and output states since they interact with their environments by receiving inputs and producing outputs and may also never terminate. To adequately describe such systems, one must refer to their behaviours, i.e. sequence of states or events. Various formalisms such as temporal logic [MP82], CSP [Hoa85], CCS [Mil89] and Petri nets [Pet77] have been developed for specifying and developing reactive systems.

Systems can further be classified according to whether their inputs and outputs must satisfy timing constraints, i.e. whether they are real-time systems. For a real-time system, it is not sufficient to view the behaviour simply as a sequence of states or events but the behaviour must also incorporate times when those states or events occur. Traditional approaches to the specification and analysis of reactive systems abstracted away from any notion of time. Time is implicit in the notion of a behaviour but its treatment at the specification level is qualitative. The advantage is that the analysis is simplified and can be applied to a variety of system architectures and implementations. However, for a real-time system, a qualitative treatment of time is inadequate. Consider the following examples:

> "A program S must terminate within k time units."

This is an example of a *bounded-response* property. It places an upper bound on the time by which an event must occur. Another class of properties is the *bounded-invariance* one. The properties in this class place a lower time bound on an occurrence of an event, for example,

> "A program S must not terminate within k time units."

The term 'bounded-invariance' is used here because the properties in this class assert that something will hold continuously (i.e. remain invariant) for a certain time period. More subtle requirements may state not only that an event must or must not occur within a certain time interval but may also contain a reference to the number of times an event must occur, for example,

> "The number of occurrences of a system failure within k time units must be less than or equal to two."

This requirement refers to time and to the number of occurrences of an event. Yet, another example refers to the cumulative duration of events, for example,

> "The total duration of gas leaks within any time interval greater than or equal k time units must be less than or equal to one-twentieth of the duration of the interval."

To describe these examples, we require a formalism that is capable of specifying and analysing real time as well as counting the number of occurrences of events and specifying their durations.

Introducing time within a formalism raises a fundamental issue, namely, which model of time should one adopt. Real-time systems operate in 'real' continuous time and therefore it would seem appropriate to use a continuous time model. There are also properties which can only be stated under a continuous interpretation of time, i.e. those which state when events occur with arbitrary precision. For example, the statement

> "If p occurs then q must occur exactly after 5 seconds"

refers precisely when q must occur in relation to p. It therefore cannot be stated under a discrete interpretation of time since it would imply that q could occur as little as 4.1 seconds or as much as 5.9 seconds after p. Yet, another reason for adopting continuous time is that requirements are often specified using continuous time. For this one must specify the requirements either in continuous time or use discrete time and prove that the properties being specified in discrete time imply the original user requirements. In either case one must be able to specify requirements using a continuous model of time.

An important issue related to the choice of the model of time are the proof and verification techniques for real-time systems. Verification of a system consists of showing that its implementation satisfies its specification. An implementation may be written in the same language as the specification in which case the proof techniques developed for the language can be used for solving the verification problem. If, on the other hand, an implementation is written in a different formalism from that of the specification, the verification problem can be solved by developing a proof method or by the so-called model checking algorithms. A proof method consists of proof rules so that an implementation can be proved to meet its specification. If the model checking algorithms are used then one shows that the set of sequences of events or states which satisfy the specification is a superset of that generated by the implementation. It has previously been observed that the proof and verification techniques for the formalisms which use a discrete model of time are significantly simpler than those using a continuous one [HMP91b]. This suggests conflicting requirements for

a formalism for the specification and verification of real-time systems. On one hand, a continuous model of time is required for expressive power and on the other, a discrete model leads to simpler proof techniques.

An issue related to the verification of a real-time system is the choice of a computation model. Two models that have been widely covered in the literature are the interleaving and the maximal parallelism models. In the interleaving model, each action is assumed to be instantaneous with minimal and maximal delays within which it must be executed after being enabled. Simultaneous actions are nondeterministically sequentialised so that only one action needs to be analysed at any time. In the maximal parallelism model, each process is assumed to have its own processor. Parallel actions may overlap as long as they do not violate synchronisation requirements. The interleaving model, in general, leads to verification methods which are simpler than those based on the maximal parallelism model. On the other hand, the maximal parallelism model can easily be extended to a more general model of real-time systems in which resources are limited and have to be scheduled to meet timing constraints.

1.2 Current Approaches to Real-time Specification and Verification

Several alternative approaches to the specification and verification of real-time systems have been suggested. These include a first-order predicate logic with time for verifying programs [Hoo91], variants of temporal logics [Ost89, HLP90, AL91, Koy89, ACD90, AH89, CHR91], Real-Time Logic [JM86a], Timed CSP [RR86], algebraic techniques [MT90, Yi90, BB90, NRSV89] and timed Petri nets [Ram74, MS76, Sif77, Raz83]. We briefly survey them in this section.

We begin by describing a Hoare logic for the verification of real-time programs [Hoo91]. This is followed by a section on temporal logics in which we first describe logics which treat time qualitatively. We then describe temporal logics which treat time quantitatively. These include the so-called explicit-clock [Ost89, HLP90, AL91] and the bounded-operator [Koy89, ACD90] temporal logics. Two temporal logics, Timed Propositional Temporal Logic [AH89] and the Duration Calculus [CHR91], which do not fall into the above classification are then described. This is followed by a description of other alternatives such as Real-Time Logic [JM86a], Timed CSP [RR86], algebraic approaches [MT90, Yi90, BB90, NRSV89] and timed Petri nets [Ram74, MS76, Sif77, Raz83].

1.2.1 Hoare Logics

Hoare logic [Hoa69] was first proposed for verifying sequential programs. It uses a correctness formula $P\{S\}Q$ (known as a Hoare triple), where S is a program and P and Q are assertions in the first-order predicate logic. The informal meaning of the triple is that if S is executed in a state in which P holds and if it terminates then Q will hold in the final state. Axioms and rules of inference for various programming language constructs are given. For example, the assignment statement satisfies the axiom $P^e_x\{x := e\}P$ where P^e_x denotes the substitution of e for x in P. Verification techniques for sequential programs have been widely covered in the literature (e.g. [Apt81]).

The Hoare triple has been extended in various ways to express additional properties. For example, the Hoare triple is used for expressing only partial correctness, i.e. the post-condition holds if the program terminates. Therefore, to express total correctness the triple is extended by an assertion called commitment C which is used for recording information about termination. The correctness formula then becomes $C : P\{S\}Q$. Various compositional [MC81, Sou83, ZRB85, Zwi88] and non-compositional [OG76, AFR80, LG81] proof systems for concurrent systems have been developed which use Hoare triples or extended Hoare triples. In [Hoo91], a compositional proof system for a real-time extension of an Occam-like language is given in which the commitment assertion contains a reference to a special variable *time* to denote the termination time of a program. A bounded-response property stating that a program S terminates within k time units is written using an extended Hoare triple as :

$$time \leq k : time = 0\{S\}true$$

and a bounded-invariance property stating that it does not terminate within k time units is asserted as :

$$time > k : time = 0\{S\}true$$

1.2.2 Temporal Logics

Untimed Temporal Logics

Temporal logic is now widely accepted as a specification language for describing the behaviour of reactive systems. Its use is motivated by two reasons. Firstly,

the underlying semantic model of temporal logic fits in well with the notion of a behaviour of a system. Secondly, its operators can be used for stating common properties concisely and in an abstract way.

Temporal logic was developed for reasoning about situations which evolve in time. The basic semantic object is a state which describes the static aspects of a situation. The dynamic aspect of a situation is described by a sequence of states; time and states are therefore implicit in the model and are not introduced explicitly in the logic. Many different temporal logics have been proposed. TL [MP82] has four temporal operators: $\bigcirc$ (next), $\mathcal{U}$ (until), $\Box$ (henceforth) and $\Diamond$ (eventually). The formula $\bigcirc\varphi$ holds in a state iff φ holds in the successor state. The meaning of the formula $\varphi\mathcal{U}\psi$ is that φ holds in all the states until ψ holds. $\Diamond\varphi$ holds in a state iff there exists a future state in which φ holds. It can be defined using *until* as $true\mathcal{U}\varphi$. $\Box\varphi$ can be derived as $\neg\Diamond\neg\varphi$: it asserts that φ holds in all future states. The logic can be used for the specification of common safety and liveness properties. For example, to state that a program does not terminate can be written as :

$$\Box\neg term$$

where *term* holds iff the program terminates. A liveness property stating that a program eventually terminates can be asserted as :

$$\Diamond term$$

We can assert that the number of occurrences of a system failure is less than or equal to two by using nested *until* as:

$$\Box(\neg f\mathcal{U}(f\mathcal{U}(\neg f\mathcal{U}(f\mathcal{U}((\neg f\mathcal{U}(term \wedge \neg f)) \vee term)))))$$

where f asserts that the system is in the fail state.

In [BB86], a temporal logic with a fixed point schema $\nu\xi.\chi(\xi)$ where ξ is a variable in χ and is bound to ν was proposed. The meaning of $\nu\xi.\chi(\xi)$ is defined as the greatest set of models in which χ is satisfied. The fixed point schema $\mu\xi.\chi(\xi)$ is defined as $\neg\nu\xi.\neg\chi(\neg\xi)$ and its meaning is defined as the least set of models in which χ is satisfied. $\varphi\mathcal{U}\psi$ can be defined using the fixed point schema and the *next* operator as $\mu\xi.(\psi \vee (\varphi \wedge \bigcirc\xi))$ from which the other temporal operators can be defined.

The logics given in [MP82] and in [BB86] contain only future temporal operators. It was shown in [KVR83] that using both past and future operators result in specifications which are more elegant. A temporal logic with both past and future fragments is given in [LPZ85] in which for each temporal operator in the future fragment, a symmetric one is defined for the past fragment. For example, the meaning of $\varphi S \psi$ is that φ is true since ψ held sometime in the past and $\diamondsuit \varphi$ means that φ was true sometime in the past.

The logics given in [MP82], [BB86] and [LPZ85] are based on sequences of states which may contain adjacent states which are equal, i.e. all variables have the same value in each of these states. These logics have been criticised for forcing too much detail in the semantic descriptions of programming languages [BKP86]. In particular, the use of the *next* operator leads to a semantics of programs in which the lowest level of atomicity is visible. To overcome this, a logic EL was proposed in [Fis87]. It is based on sequences of states with the restriction that two adjacent states are different except for the states in the tail of a sequence which may be identical. This makes the logic insensitive to finite stuttering (i.e. duplication of states) but makes it sensitive to infinite stuttering. EL is therefore similar to TL except that the *next* operator is interpreted as a 'next-change' instead of a 'next-time' operator. Another alternative for achieving an abstract semantics of programming languages is to use a dense instead of a discrete sequence of states and therefore do away with the *next* operator. Such a logic called TLR is described in [BKP86].

In TL and in other discrete linear-time temporal logics, a formula is said to be valid if and only if it holds in all the states in all the sequences of states. This definition of validity leads to a $\Box$-Introduction rule $\varphi \vdash \Box \varphi$ which states that if φ is a theorem then $\Box \varphi$ is also a theorem. The inclusion of the rule in the logic invalidates the deduction theorem (i.e. $\varphi \vdash \psi$ iff $\vdash \varphi \Rightarrow \psi$) used in the first-order predicate logic. To see this, it is sufficient to observe that although $\varphi \vdash \Box \varphi$ holds it does not follow from it that $\vdash (\varphi \Rightarrow \Box \varphi)$ also holds. To overcome this, an 'anchored' version of TL is described in [MP89a] in which two types of validity are defined. A first-order formula without any temporal operators is said to be valid iff it holds in every state in all the sequences of states. A temporal formula is said to be valid iff it holds in the first state in all the sequences of states. Therefore, in the anchored version the $\Box$-Introduction rule does not hold.

The underlying semantic model of temporal logics discussed so far is linear sequences of states. Each state in these logics has a unique successor. In a 'branching-time' logic, a state may have many successors, each giving rise to a different future. Two approaches are possible in branching-time logics. One is to introduce quantifiers over each future (i.e. paths) as done in UB [BMP83] and CTL [EC82] and the other is to redefine temporal operators as in [Lam80].

In UB and CTL, two operators "$\exists$" and "$\forall$" are introduced representing "for some path" and "for all paths" respectively. These operators are prefixed to temporal assertions with the restriction that only a single occurrence of a temporal operator can occur in the assertion. In [Lam80], the meaning of temporal operators is changed. $\Diamond\varphi$ means that φ holds sometime during every possible future and $\Box\varphi$ means that φ always holds in all possible futures.

All the temporal logics discussed so far are based on viewing time as points instead of intervals. In [MM83], a logic based on intervals was proposed. In an interval temporal logic, $\Diamond\varphi$ means that there exists a subinterval in which φ holds and $\Box\varphi$ means that φ holds in all subintervals. A binary operator *chop* ($\hat{}$) is also introduced. $\varphi\hat{}\psi$ holds for an interval iff it can be subdivided into two subintervals of which φ holds in the first and ψ in the second. To state in an interval temporal logic that a program does not terminate we use

$$\Box\neg term$$

A liveness property stating that a program eventually terminates is asserted as:

$$\Diamond term$$

We can assert that the number of occurrences of a system failure is less than or equal to two by using the *chop* operator as follows :

$$(\Box f \vee \Box\neg f)\hat{}(\Box f \vee \Box\neg f)\hat{}(\Box f \vee \Box\neg f)\hat{}(\Box f \vee \Box\neg f)$$
$$\hat{}(((\Box\neg f)\hat{}\Box(term \wedge \neg f)) \vee \Box term)$$

Verification of a system in an untimed temporal logic can be done algorithmically by using model checking algorithms or deductively by developing proof rules and techniques. In the case where the implementation and the specification are written in the same language the proof techniques developed for the language can be used. Two proof techniques are widely used in a discrete linear-time temporal logic. To prove a safety property, one uses a computation induction principle by proving that the property holds in the initial state and that it is preserved by any transition by the system. Liveness properties, on the other hand, are proved by structural induction on some element of the state. In the case where the implementation and the specification are not in the same language a proof method is developed based on defining a relation **sat** between the implementation and the specification languages and by giving a proof rule for each statement in the implementation language. For example, the proof rule for the output statement in CSP [Hoa78] is stated as :

$$c!e \textbf{ sat } idle(\varnothing)\mathcal{U}(send(c,e) \vee block(\{c!\})) \vee \Box idle(\varnothing)$$

The rule asserts that environment transitions take place with no communication on offer until either an internal transition takes place with the value of e being output on the channel c and no change in program variables or the communication on the channel c is blocked. A complete proof method for reactive systems based on an interleaving model using an untimed temporal logic can be found in references such as [MP82] and [BKP84, BKP85]. In [MP89b], a proof method in which the proofs of temporal properties are reduced to assertional ones is given.

Note that in the examples given so far no time bound is specified on when events can occur. To introduce time into temporal specifications the explicit-clock and the bounded-operator approaches have been proposed [HMP91a]. Both approaches are described in this section.

Explicit Clock Temporal Logics

Time in an explicit clock temporal logic is introduced by using a variable, usually T, to denote the current time of a conceptual global clock. This approach does not require any new temporal operators. Several explicit clock temporal logics have been proposed, notable ones being RTTL [Ost89], XCTL [HLP90] and TLA [Lam91, AL91]. The underlying semantics is given by sequences of states which are timed using a global clock. Time satisfies the following two requirements: it does not decrease in a successor state and eventually it must increase. Both RTTL and XCTL use a discrete model of time. A bounded-response property stating that a program will eventually terminate within k time units is expressed in RTTL and XCTL as :

$$init \wedge (T = x) \Rightarrow \Diamond(term \wedge (T \leq x + k))$$

where *init* holds in the initial state and *term* in the termination state. This formula uses a free variable x to 'freeze' the current value of T so that it can be compared with the value of T when *term* occurs. Free variables in temporal logics are often referred to as global or rigid variables since their values remain the same in all the states. On the other hand, the values of the variables such as T can change from one state to another. They are known as flexible, program, state or local variables. In RTTL and XCTL, quantification is only allowed over free variables.

Using an explicit clock temporal logic, a liveness property can be expressed as a safety property under the assumption that time progresses [PH88]. For example, the above formula can be written as :

$$init \wedge (T = x) \Rightarrow (T < x + k) \mathcal{U} term$$

which asserts that T will not exceed k time units before *term* becomes true. A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$init \wedge (T = x) \Rightarrow \Box((T \leq x + k) \Rightarrow \neg term)$$

We can assert that the number of occurrences of a system failure is less than or equal to two in an interval of k time units by

$$\Box((T = x) \Rightarrow \neg f \mathcal{U} (f \mathcal{U} (\neg f \mathcal{U} (f \mathcal{U} ((\neg f \mathcal{U} ((T = x + k) \wedge \neg f)) \vee (T = x + k))))))$$

where f denotes that the system is in the fail state.

RTTL and XCTL are essentially similar in all respects except one. RTTL allows nested quantification over global variables whereas XCTL does not. One can view an XCTL formula as an RTTL formula in which quantification is only allowed at the outermost level. Therefore, it is not possible to express properties in which quantification of free variables cannot be reduced to the outermost level. For example, if we wanted to state "if p always leads to q being true after five time units then r always leads to s being true after ten time units", we would use the following formula.

$$(\forall x. \Box(p \wedge (T = x) \Rightarrow \Diamond(q \wedge (T = x + 5)))) \Rightarrow (\forall y. \Box(r \wedge (T = y) \Rightarrow \Diamond(s \wedge (T = y + 10))))$$

This formula cannot be reduced to a formula with quantification at the outermost level. On the other hand, it has been shown in [AH89] that unrestricted quantification leads to undecidability.

Verification of a real-time system in an explicit clock temporal logic can, in principle, be done by using the proof rules and techniques developed for an untimed temporal logic. In particular, the computation induction principle for

the untimed temporal logics can be retained to prove safety properties if the condition that time increases by at most one time unit is also included. A proof method for a real-time extension of a CSP-like language based on the maximal parallelism model of computation and using a continuous model of time can be found in [ZHK91] and for a transition system using discrete time and an interleaving model in [HMP91a].

Temporal Logic of Actions

The Temporal Logic of Actions (TLA) [Lam91] combines two logics: a logic of actions and a simple temporal logic. An action is defined as a boolean-valued expression formed from variables, primed variables and values. For example, the action $x' = x + 1$ where x is a variable, is a relation between states in which the unprimed variables refer to the old state and the primed to the new: thus, $x' = x + 1$ states that the value of x in the new state (i.e. x') is one greater than its value in the old state (i.e. x). A predicate is a boolean-valued expression with no primed variables.

An RTLA (Raw TLA) formula consists of actions, the usual classical operators ($\neg$, $\vee$, $\forall$) and a single unary temporal operator $\Box$. The meaning of $\Box$ is given over behaviours which are infinite sequences of states. $\Box\varphi$ asserts that φ is true in all states. $\Diamond$ is derived from $\Box$. Quantification is allowed only over rigid variables.

A TLA formula is formed using actions, the usual classical operators, $\Box$ and $[\mathcal{A}]_f$, where $\mathcal{A}$ is any action and f is an expression (called a state function) formed from variables, constants, and operators. The meaning of an action, the classical operators and $\Box$ remain the same. The meaning of $[\mathcal{A}]_f$ is that either $\mathcal{A}$ holds or f remains unchanged. Quantification is also allowed over flexible variables. If x is a flexible variable and φ is a formula in TLA then $\exists x.\varphi$ is true for a behaviour iff there exists a sequence of values that can be assigned to x such that φ holds.

Time is introduced in TLA using a variable *now*, denoting an explicit clock [AL91]. The value of *now* is a real number and it never decreases. It is expressed using a TLA formula as follows :

$$RT \triangleq (now \in \mathbf{R}) \wedge \Box[(now' \in (now, \infty)) \wedge v' = v)]_{now}$$

where $\mathbf{R}$ is the set of real numbers, (now, ∞) is the set $\{t \in \mathbf{R} | t > now\}$ and v is a set of program variables. The formula asserts that the program

variables do not change when *now* increases. Real-time constraints are imposed by adding conjuncts to a specification. For example, to state a bounded-response property that a program will eventually terminate within k time units is expressed as :

$$init \wedge (now = x) \Rightarrow \Diamond(term \wedge now \leq k)$$

A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$init \wedge (now = x) \Rightarrow \Box((now \leq x + k) \Rightarrow \neg term)$$

Verification of a real-time system in TLA can be done by using the standard techniques developed for the untimed temporal logics. The underlying method of proving safety and liveness properties is assertional. The advantage of using TLA is that the algorithms and properties are both written in the same language. Therefore, a property is proved using logical implication and no programming language-based proof rules are given.

Metric Temporal Logic

Metric Temporal Logic (MTL) [Koy89] is a linear-time temporal logic in which temporal operators are associated with explicit time bounds. In an untimed temporal logic such as the one described in [MP82], one uses the temporal formula $\Diamond\varphi$ to state that φ will eventually hold in some state in the future. No time is specified for when φ must occur in relation to the present. In MTL, temporal operators with time bounds are defined: for example, the formula $\Diamond_{\leq k}\varphi$ states that the event φ occurs within k time units from now. To define time-bound temporal operators, a metric function is used to relate two points in time by measuring the distance between them. The underlying model of time is continuous and therefore MTL does not have a *next* operator.

MTL provides a concise notation for specifying common real-time properties. For example, a bounded-response property stating that a program will eventually terminate within k time units, is expressed as:

$$init \Rightarrow \Diamond_{\leq k} term$$

where *init* holds in the initial state and *term* in the termination state. A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$init \Rightarrow \Box_{\leq k} \neg term$$

To assert that the number of occurrences of a system failure is less than or equal to two in any interval of k time units we use

$$\begin{aligned}\Box(\exists a, b, c, d, e.((a+b+c+d+e=k) \\ \wedge \neg f\mathcal{U}_{=a}(f\mathcal{U}_{=b}(\neg f\mathcal{U}_{=c}(f\mathcal{U}_{=d}(\neg f\mathcal{U}_{=e}\neg f))))) \\ \vee \exists a, b, c, d.((a+b+c+d=k) \wedge \neg f\mathcal{U}_{=a}(f\mathcal{U}_{=b}(\neg f\mathcal{U}_{=c}(f\mathcal{U}_{=d} true)))))\end{aligned}$$

Verification of a real-time system in MTL can be done, in principle, by decorating the proof rules for an untimed temporal logic with time bounds. Note that the computation induction principle used in the untimed linear-time temporal logics for proving safety properties cannot be used in MTL because it uses a continuous model of time and therefore does not have a *next* operator. A compositional proof system for a real-time extension of a CSP-like language using the maximal parallelism model is given in [Hoo91]. In [HMP91a], a proof method based on an interleaving model for a transition language using a bounded-operator temporal logic with discrete time is given.

Timed Computation Tree Logic

Timed Computation Tree Logic (TCTL) [ACD90] is a timed version of CTL [EC82]. Its semantics is based on modelling the behaviour of a system as a tree structure instead of a linear sequence. Three temporal operators are defined in CTL : $\exists\bigcirc$, $\exists\mathcal{U}$, and $\forall\mathcal{U}$. $\exists\bigcirc\varphi$ means that there exists a successor state in which φ holds. $\exists\varphi\mathcal{U}\psi$ means that there exists a computation path in which φ holds until ψ. Similarly, $\forall\varphi\mathcal{U}\psi$ holds iff for all computation paths $\varphi\mathcal{U}\psi$ holds. Some commonly used abbreviations are defined as follows : $\exists\Diamond\varphi$ for $\exists true\,\mathcal{U}\varphi$, $\forall\Diamond\varphi$ for $\forall true\,\mathcal{U}\varphi$, $\exists\Box\varphi$ for $\neg\forall\Diamond\neg\varphi$ and $\forall\Box\varphi$ for $\neg\exists\Diamond\neg\varphi$. In TCTL, CTL operators are redefined by using subscripts to specify time bounds. So, for example, $\exists\varphi\mathcal{U}_{\leq k}\psi$ means that for some computation path ψ holds within k time units and φ holds until then. A similar interpretation is given for $\forall\varphi\mathcal{U}_{\leq k}\psi$. TCTL does not have a *next* operator since only a continuous model of time is considered. Some commonly used abbreviations are defined as follows: $\exists\Diamond_{=k}\varphi$ for $\exists true\,\mathcal{U}_{=k}\varphi$, $\forall\Diamond_{=k}\varphi$ for $\forall true\,\mathcal{U}_{=k}\varphi$, $\exists\Box_{=k}\varphi$ for $\neg\forall\Diamond_{=k}\neg\varphi$ and $\forall\Box_{=k}\varphi$ for $\neg\exists\Diamond_{=k}\neg\varphi$.

A bounded-response property stating that a program will eventually terminate within k time units is expressed in TCTL as :

$$init \Rightarrow \forall\Diamond_{\leq k} term$$

where *init* holds in the initial state and *term* in the termination state. A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$init \Rightarrow \forall \Box_{\leq k} \neg term$$

To assert that the number of occurrences of a system failure is less than or equal to two in any interval of k time units we use

$$\begin{aligned}\Box(\exists a,b,c,d,e.((a+b+c+d+e=k) \\ \wedge \forall(\neg f \mathcal{U}_{=a}(f \mathcal{U}_{=b}(\neg f \mathcal{U}_{=c}(f \mathcal{U}_{=d}(\neg f \mathcal{U}_{=e} \neg f)))))) \\ \vee \exists a,b,c,d.((a+b+c+d=k) \wedge \forall(\neg f \mathcal{U}_{=a}(f \mathcal{U}_{=b}(\neg f \mathcal{U}_{=c}(f \mathcal{U}_{=d} true))))))\end{aligned}$$

In [CES86], algorithms for model checking for CTL are presented using a discrete time domain which can be used for verification. These results are extended to TCTL in [ACD90] for a dense domain.

Timed Propositional Temporal Logic

Timed Propositional Temporal Logic (TPTL) [AH89] is an extension of the propositional part of TL [MP82]. It uses auxiliary variables to record the current time of a global clock by using a quantifier to freeze its value and therefore does not make any explicit references to a global clock. The syntax of the quantifier is given by $x.\varphi(x)$ and is interpreted to hold for a timed sequence σ (i.e. a sequence of (time,state) pair) iff $\varphi(x)$ holds for a timed sequence σ' which at most differs from σ by the interpretation given to x and in which x is given the value of the current time. To express a bounded-response property that a program will eventually terminate within k time units, we write

$$x.init \Rightarrow \Diamond y.(term \wedge (y \leq x + k))$$

where *init* holds in the initial state and *term* in the termination state. The current value of time is recorded in x and is compared with the value of y which is the time when *term* holds. A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$x.init \Rightarrow \Box y.((y \leq x + k) \Rightarrow \neg term)$$

We assert that the number of occurrences of a system failure is less than or equal to two in an interval of k time units by

$$\Box x.(\neg f\mathcal{U}(f\mathcal{U}(\neg f\mathcal{U}f(f\mathcal{U}((\neg f\mathcal{U}(y.(y = x + k) \wedge \neg f)) \vee y.(y = x + k))))))$$

where f denotes that the system is in the fail state.

Expressions in TPTL can only have the form $x + c$ where x is a time variable and c is an integer constant. It is shown in [AH89] that restricted expressions and a discrete time domain are required for the logic to be decidable.

A Calculus of Durations

An alternative to the explicit-clock and the bounded-operator temporal logics is proposed in [CHR91] using a calculus of durations. The calculus is based on an interval temporal logic [MM83]. It uses integrated durations of states in an interval to reason about timing requirements. No reference is therefore made to explicit time.

A state is defined using boolean variables which are mapped to either 1 or 0, where 1 denotes that the system is in the state and 0 denotes that the system is not in the state. The duration of a state φ in a closed interval $[b, e]$ is defined by an integral $\int_b^e \varphi(t)dt$ and is denoted by $\int\varphi$. It is easy to see that $\int 1$ in any interval $[b, e]$ is $e - b$ and $\int 0$ is zero. Given this definition, axioms are given which relate the durations of different states.

A further notation is introduced which converts a state φ to a duration predicate ($\lceil\varphi\rceil$). $\lceil\varphi\rceil$ is defined as :

$$\lceil\varphi\rceil \triangleq (\int\varphi = \int 1) \wedge \int 1 > 0$$

That is, $\lceil\varphi\rceil$ holds for a nonempty interval iff φ holds almost throughout the interval. A point interval is defined as :

$$\lceil\ \ \rceil \triangleq \int 1 = 0$$

A formula $D_1\hat{}D_2$, where D_1 and D_2 are duration predicates, is introduced and holds for an interval which can be subdivided into two sub-intervals of which D_1 holds in the first and D_2 holds in the second. Interval temporal operators *eventually* ($\Diamond$) and *henceforth* ($\Box$) are defined in terms of the *chop* ($\hat{}$). The logic has an induction rule

$$\frac{\vdash R(\lceil\ \ \rceil), R(X) \vdash R(X \vee X\hat{}\lceil\varphi\rceil \vee X\hat{}\lceil\neg\varphi\rceil)}{R(true)}$$

where X is a formula letter and φ is a state, to prove properties over arbitrary real intervals. The resulting logic can be used for expressing timing requirements concisely. For example, the bounded-response property

$$(\lceil init\rceil\hat{}true\hat{}\lceil term\rceil) \Rightarrow \int 1 \leq k$$

asserts that a program terminates within k time units. A bounded-invariance property stating that a program does not terminate within k time units is stated as :

$$(\lceil init\rceil\hat{}true\hat{}\lceil term\rceil) \Rightarrow \int 1 > k$$

To assert that the number of occurrences of a system failure is less than or equal to two in any interval of k time units we use

$$\Box((\Diamond\lceil f\rceil)\hat{}(\Diamond\lceil\neg f\rceil)\hat{}(\Diamond\lceil f\rceil)\hat{}(\Diamond\lceil\neg f\rceil)\hat{}(\Diamond\lceil f\rceil) \Rightarrow \int 1 > k)$$

where f denotes that the system is in the fail state. The total duration of gas leaks in any interval greater than or equal to k time units is less than or equal to one-twentieth of the duration of the interval is stated as :

$$\int 1 \geq k \Rightarrow 20 \int l \leq \int 1$$

where l denotes that the system is in the leak state. The soundness and relative completeness of the calculus is shown in [CH91]. A proof system for verifying the timing properties of programs written in a CSP-like language and implemented on shared processors is given in [CHRR92].

1.2.3 Real-Time Logic

Real-Time Logic (RTL) [JM86a, JM86b] was developed to reason about the timing constraints of a system. A system is initially described in RTL using an event-action model. During this phase, the system designer specifies actions,

events, state predicates and the timing relationships between them. Actions are either primitive or composed from the primitive ones using the sequential (;), parallel (||) and nondeterministic (|) composition operators. Examples of primitive actions are PB (push button) and SRC (send request to control unit) which can be combined, for example, to form the action $PB; SRC$. Events in RTL are used as markers to specify time of starting or finishing of an action, or the change in a state predicate of the system. Two kinds of timing constraints are specified - periodic and sporadic. Periodic timing constraints are specified using the syntax

> while <predicate>, execute <action> with period = <time>,
> deadline = <time>

and sporadic constraints by

> when <event>, execute <action> with deadline = <time>,
> separation = <time>

The event-action model captures timing constraints which are transformed into an RTL formula. An RTL formula is formed using constants and an occurrence function. There are three kinds of constants: actions, events and integers. The occurrence function @ is introduced to capture the notion of time. Given an event e and a nonnegative integer i, the @ function returns the time of the i^{th} occurrence of e. The model of time used is discrete. Using the occurrence function, one can specify bounded-response and bounded-invariance properties. For example, to state that a program will eventually terminate within k time units one would write:

$$@(term, 1) - @(init, 1) \leq k$$

where *init* holds in the initial state and *term* in the termination state. A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$@(term, 1) - @(init, 1) > k$$

To assert that the number of occurrences of a system failure is less than or equal to two in any interval of k time units one writes :

$$\forall i \geq 1.@(\uparrow f, i+3) - @(\uparrow f, i) > k$$

where $\uparrow f$ denotes the starting of a system failure.

Further notations are used for describing that a state predicate is true in an interval. For example, $\varphi[t_1, t_2]$ asserts that φ becomes true at t_1 and remains true until it becomes false at t_2, and $\varphi[t_1, t_2>$ states that φ becomes true at t_1 and remains true until it becomes false at some time after t_2.

RTL is used for reasoning about the safety properties of a system. To prove a safety property, its negation is shown to be unsatisfiable. In practice, the negation of an RTL formula is transformed into an equivalent one in Presburger arithmetic with uninterpreted functions. Full Presburger arithmetic with uninterpreted functions is undecidable. Hence, to improve decision procedures, knowledge of the application domain is used for restricting RTL formulae. In [JS88], Modecharts, a visual formalism for a subset of RTL formulae is introduced which can be translated into computation graphs for verification.

1.2.4 Timed CSP

CSP [Hoa85] is a language for describing and reasoning about concurrency. The most basic concept of CSP is the alphabet of a system, which is the set of events needed for its description. For example, the alphabet of a simple stop-and-wait protocol may have two events - input and output of messages. A process in CSP is used for describing the behaviour of an object using the events from its alphabet. The most basic process is $STOP_A$ which describes the behaviour of a process with an alphabet A which does not engage in any of its events. More complex processes can be constructed using various operators. For example, if a is an event and P is a process then $a \rightarrow P$ describes the behaviour of an object which first engages in the event a and then behaves as P. Other operators include recursion, concurrency, nondeterminism, concealment, and sequential composition. Recursion is used for describing a system which repeats its behaviour. For example, a stop-and-wait protocol which keeps inputting and outputting messages is described as :

$$SW = in \rightarrow out \rightarrow SW$$

Two processes P and Q can be combined in parallel to form a process $P \parallel Q$. Events which are common to both processes require the participation of both P and Q and those which are not common may occur independently.

The processes deadlock if they disagree on what the first event should be. A special kind of event which requires the participation of both processes is communication. An output event represented by $c!e$ is used for describing the transmission of the value of e on a channel c. Input event $c?x$ receives a value on a channel c which is stored in x. Nondeterminism is introduced by the $\sqcap$ and $\|$ operators. A process $P \sqcap Q$ behaves like P or like Q, the choice being made arbitrarily and without the control of the environment. For a process $P \| Q$, the environment controls the choice of whether P or Q is selected, depending on the first event of P and Q. If both can be selected the choice is non-deterministic. Concealment is used for hiding events which occur internally. $P \backslash \{C\}$ is a process which behaves like P but with any event in C occurring automatically and instantaneously without being observed. Hiding of events may lead to an infinite sequence of hidden events - a phenomenon known as divergence. A divergent process leaves its environment waiting eternally. An example of such a process is $SW \backslash \{in, out\}$.

A mathematical model of CSP is a 3-tuple (A, F, D) where A is the alphabet of a process P, F is a set of failures and D is a set of traces (i.e. sequences of events) of P known as divergences after which it diverges. A failure of a process P is a pair (tr, X) where tr is a trace of P and X is a set of events (known as refusals) which its environment is prepared to engage in but refuses to do anything after P has engaged in the sequence of events recorded by tr.

The timed extension of CSP [RR86, RR88] uses the same syntax as for CSP with addition of processes $wait\ t$ $(0 \leq t)$ and $\perp$. The process $wait\ t$ terminates successfully after t time units. A diverging process $\perp$ engages in events invisible to its environment. The process $a \rightarrow P$ initially engages in a and after a short delay δ time units behaves like P. Recursion also introduces a short delay of δ time units. The meaning of the other operators remain the same. Note that the delay constant δ is required to ensure that only a finite number of events can occur in a finite interval of time. The timed models introduce a 3-tuple consisting of timed traces, timed refusals and a stability value represented by $(tr, \aleph, \alpha)$. A timed trace of a process is a sequence of events, each tagged with the time at which it occurs. In addition, a trace may contain an event decorated with a hat ($\hat{}$) if it occurs at the first moment of its availability. The computation model in CSP and Timed CSP is interleaving, therefore only a single event is recorded in the trace at any time. Timed refusals are pairs of left-closed right-open finite intervals and a set of events in which the environment is prepared to engage during the observation of tr but which are not offered by the process. The pair $(tr, \aleph)$ is known as timed failures. The value α called the stability value is the earliest time by which the process must stabilise, i.e. there cannot be any change in the state without an external event occurring.

A specification of a timed CSP process is a predicate over all its possible behaviours. For example, a bounded-response property expressing that a process will terminate within k time units is stated as :

$$\forall t_1, t_2 \in \mathbf{R}. < (t_1, init), (t_2, term) > \text{ in } tr \Rightarrow (t_2 - t_1) \leq k$$

where *init* is the initialisation event and *term* denotes the termination of the process. A bounded-invariance property stating that a process does not terminate within k time units is stated as :

$$\forall t_1, t_2 \in \mathbf{R}. < (t_1, init), (t_2, term) > \text{ in } tr \Rightarrow (t_2 - t_1) > k$$

The number of occurrences of a system failure is less than or equal to two in any interval of k time units is stated as :

$$\forall t_1, t_2, t_3 \in \mathbf{R}. < (t_1, \uparrow f), (t_2, \uparrow f), (t_3, \uparrow f) > \text{ in } tr \Rightarrow (t_3 - t_1) > k$$

where $\uparrow f$ denotes the start of a system failure.

Verification in Timed CSP consists of developing a proof method based on defining a relation **sat** between Timed CSP and the specification language and by giving a proof rule for each process construct. For example, the proof rule for the *STOP* process is stated as:

$$\frac{tr = <> \Rightarrow S(tr, \aleph, \alpha)}{STOP \ \textbf{sat} \ S(tr, \aleph, \alpha)}$$

A complete proof system for the timed failures model is given in [DS89].

1.2.5 Timed Process Algebras

Process algebras which have been extended to incorporate time include CCS [Mil89], ACP [BK84], and ATP [NRSV89]. CCS is a general calculus for reasoning about concurrent systems. The basic language consists of a set of action names and agent expressions. The set of action names corresponds to the alphabet in CSP terminology and agent expressions to processes. Actions and agent expressions can be combined to form more complex agent expressions. If a is an action and P is an agent expression then the correspondence with the CSP syntax is as follows:

$a.P$	corresponds to $a \rightarrow P$
$\mathbf{0}$	corresponds to $STOP$
$\mathbf{fix}(P = a.P)$	corresponds to $P = a \rightarrow P$

CCS also has a perfect action τ which is an internal action and cannot be observed externally. The parallel operator denoted by | is more complex than the CSP one since it involves interleaving, synchronisation and nondeterminism. Actions in CCS can be of two types - either simple (e.g. a) and ones with bars over them (e.g. $\bar{a}$). When two processes are run in parallel they may synchronise if one process engages in a simple action and the other in the corresponding action with the bar symbol over it. The result of synchronising is that the event is replaced by τ. In CSP, synchronised events are observed externally unless explicitly hidden.

Nondeterminism in CCS is expressed by the + operator. It involves nondeterminism in its purest form where the environment has no control ($\sqcap$ in CSP) and also where it has control ($\Box$ in CSP). The concealment operator in CCS is defined in terms of the restriction operator \ which removes actions from the set of action names and replaces them by τ in the agent expressions.

CCS has fewer operators than CSP. This was done in order to define different equivalences between agent expressions, each of which leads to a different model of concurrency. For example, in strong equivalence one can distinguish between $a.\tau.\mathbf{0}$ and $a.\mathbf{0}$ but under observational equivalence they are identical.

In CCS, the implementation and the specification languages are one and the same. Therefore, to show that a process meets its specification, we describe both as processes and prove that they are equivalent (using algebraic laws) according to a model of concurrency. Another approach provides a specification language in the form of modal logic. Two operators, <> and [] are introduced to express properties of processes. The modality $< a > \varphi$ states that a process P may do a and then behave as φ and its dual $[a]\varphi$ asserts that a process P does a and behaves as φ.

Time is introduced in various ways in CCS. For example, in Synchronised CCS [Mil89], all processes progress in lockstep; therefore time is discrete. In [MT90], two operators are added to CCS to include time. $(t).P$ represents a process which will behave like P after exactly t time units. $\delta.P$ is a process which waits any amount of time before behaving as P. A bounded-response property expressing that a process will terminate within k time units is stated as :

$$init.(t).term.\mathbf{0} \text{ for } t \leq k$$

where *init* and *term* are initialisation and termination events respectively. A bounded-invariance property expressing that a process does not terminate within k time units is stated as :

$$init.(t).\delta.term.\mathbf{0} \text{ for } t > k$$

A sound and complete axiomatisation based on integer time is given in [MT90].

In [Yi90], SCCS is extended to include an arbitrary time delay. For example, in SCCS, $P \xrightarrow{1} Q$ means that P will behave as Q after one time unit. This is extended using the notation $P \xrightarrow{\pi(t)} Q$ to mean that P will idle for t time units and then behave as Q. Time can be either discrete or dense.

ACP [BK84] has similar operators to CCS. For example, $+$, $\|$ ($|$ in CCS) and τ have the same meaning. In addition, ACP has left merge, deadlock (δ) and communication merge ($|$) operators. Timed ACP [BB90] is an extension of ACP to include time represented by the non-negative reals. Each action is timestamped with the time at which it occurs. Thus, the basic set of actions is $\{a(t) | a \in A_\delta \wedge t \in \mathbf{R}^{\geq 0}\}$ where A_δ is a set of actions including δ.

ATP [NRSV89] is a process algebra with operators like those of CCS and ACP. In addition, it contains a distinguished action χ. Actions are considered instantaneous except for χ which represents the progress of time. χ is not included explicitly in the syntax but appears in the definition of the semantics. To express timing constraints, a binary delay operator is introduced. The meaning of a process $\lfloor P \rfloor(Q)$ is that it behaves as P if it starts before an occurrence of time unit χ; but if it does not perform P and χ occurs, it then behaves as Q. To model delay statements of programming languages, two derived operators are introduced. $\lfloor P \rfloor^d(Q)$ (start delay within d) has the following meaning : it behaves as P if the initial action of P occurs before the d^{th} occurrence of χ or else it behaves as Q after the d^{th} occurrence of χ. Similarly, $\lceil P \rceil^d(Q)$ (execution delay within d) is a process which behaves like P until the d^{th} occurrence of χ and like Q afterwards.

A process in Timed ACP and ATP like CCS is regarded as both a specification and an implementation. Therefore, verification consists of demonstrating using algebraic laws that two processes are equivalent.

1.2.6 Timed Petri Nets

A Petri Net [Pet77, Rei85] is a formalism for modelling concurrent systems. The basic (untimed) Petri net model consists of 5-tuple (P, T, I, O, μ) where

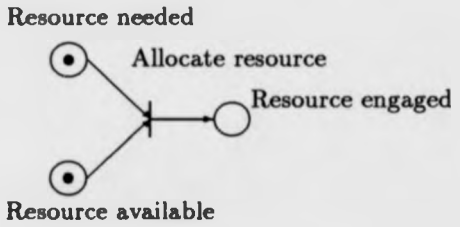

Figure 1.1: A Petri net graph

P is a finite set of places, T is a finite set of transitions, I is an input function which maps transitions to a set of input places and O is an output function which maps transitions to a set of output places. These four items define the static properties of a system. Petri nets are usually illustrated using directed graphs, as in Figure 1.1.

A Petri net graph consists of two types of nodes - circles to represent places and bars to represent transitions. The input and output functions are represented by directed arcs from places to transitions and from transitions to places respectively. To model a system, conditions are associated with places and events with transitions. To state which condition is true at any time, tokens indicated by black dots are put in places. An assignment of places is called a marking μ and is used for describing the dynamic behaviour of a Petri net. The state of a Petri net is defined by a marking. A Petri net executes by firing transitions when they are enabled. A transition is enabled iff all the places whose output arcs are directed to it have at least one token in it. After a transition has been fired a token from each of its input places is removed and a token placed in each of its output places. Therefore, firing of a transition may lead to a possible change in a marking.

Transitions in Petri nets are instantaneous and do not have a duration. There have been several proposals to extend the basic Petri net model to include time. All of these involve introducing delays with either transitions or places. For example, in [Ram74] it is proposed to associate delays with transitions. Transitions are disabled for a period equal to the delay associated with it after all its input places have at least one token in them, but may fire immediately afterwards. Another proposal is to associate both a minimum and a maximum delay with transitions [MS76]. A transition τ can fire at time k iff it has been continuously enabled during the interval $k - Min(\tau)$. The value of k must be between $Min(\tau)$ and $Max(\tau)$. A timed Petri net model therefore

consists of the 5-tuple for the basic net plus Min and Max functions which map transitions to real numbers. The timed model is equivalent to the basic model iff the minimum time is zero and the maximum is ∞ for all transitions. Associating delays with places is proposed in [Sif77]. In [LS87], it is claimed that both mechanisms are equivalent and can be translated from one into the other. Yet, another proposal is to use enabling and firing times [Raz83]. Tokens are consumed after the enabling time has elapsed and sent to output places after the firing time has elapsed. Bounded-response and bounded-invariance properties are expressed using Petri net graphs by labelling transitions or places with time bounds.

Petri nets are analysed using reachability trees which consists of nodes representing markings and arcs representing transitions. Safety properties state that an undesirable marking does not occur as a node in the tree and therefore cannot be reached by a sequence of transitions. Another safety property of interest is the maximum number of tokens in any place at any time and therefore finiteness of the state-space. Liveness properties concern transitions which are potentially fireable in all reachable markings. These can be used for analysing a system for potential deadlocks.

1.3 Aim of the Thesis

The aim of this thesis is to develop a temporal logic ωTL and a proof method for the requirements specification, and the design and verification of real-time systems. The logic we develop is an explicit clock temporal logic. It differs from other explicit clock temporal logics in two important respects. Firstly, it is independent of whether time is discrete, continuous or dense. We can use either a continuous, a discrete or a dense time model depending on the application area. Secondly, we define the *next* operator in the logic as the next time point at which a state transition occurs if such a time point exists in the future. If no state transitions occur in the future then the *next* operator is defined as the time point obtained by incrementing the current time by one. Informally, a state transition occurs at a time point τ iff there does not exist another time point which precedes it or there exists a time point τ' such that τ' precedes τ, and the state at τ' is not equal to the state at τ and at all time points between τ' and τ the state is equal to the state at τ'. This definition of the *next* operator has four important consequences.

Firstly, it means that ωTL is a combination of a 'next-change' and a 'next-time' temporal logic. It is a 'next-change' logic if there exists a future state transition point and it is a 'next-time' logic if no state transitions occur in the

future.

Secondly, it means that the logic is insensitive to stuttering which occurs for only a finite time interval but is sensitive to stuttering which occurs for an infinite interval of time. It is therefore in the tradition of EL [Fis87] and can be used for achieving an abstract semantics of programming languages.

Thirdly, it allows us to retain the proof and verification methods developed for the untimed temporal logics irrespective of whether the time model used is discrete, continuous or dense. In particular, we can use a computation induction principle to prove safety properties. To prove a safety property, we prove that it holds at the first state state transition point, and at all future state transition points and at time points which are obtained by adding positive integer values to the last state transition point. Since, in an explicit clock temporal logic, liveness properties can be expressed as a safety ones under the assumption that time progresses, the computation induction principle can also be used to prove liveness properties. Note that the computation induction principle in ωTL differs from that in RTTL or XCTL where induction is done over time points.

Fourthly, it means that a formula in ωTL is only evaluated at state transition points and at time points which are obtained by adding positive integer values to the last state transition point. Therefore, the meaning of the temporal operators is different from that in RTTL and XCTL. For example, the meaning of $\Diamond\varphi$ (eventually) in ωTL is that φ holds at some future state transition point or at a time point which is an integral number of time units after the last state transition point and the meaning of $\Box\varphi$ (henceforth) is that φ holds at all future state transition points and at all time points obtained by adding positive integer values to the last state transition point.

Common real-time properties can be stated in ωTL in a similar way to that in other explicit clock temporal logics. For example, to state that a program will eventually terminate within k time units is asserted as :

$$init \wedge (T = x) \Rightarrow \Diamond(term \wedge (T \leq x + k))$$

where *init* holds in the initial state and *term* in the termination state. The informal meaning of the formula is that either *term* holds at some future state transition point which occurs within k time units or *term* holds at some time point which is within k time units and which is an integral number of time units after the last state transition. A bounded-invariance property stating that a program does not terminate within k time units is asserted as :

$$init \wedge (T = x) \Rightarrow \Box((T \leq x + k) \Rightarrow \neg term)$$

Its informal meaning is that *term* does not hold at all state transition points which occur within k time units, and at all time points which occur within k time units and which are obtained by adding a positive integer value to the last state transition point. Note that time can be either continuous, discrete or dense.

Another aim of this thesis is to develop ways of defining properties in which bounds on the number of occurrences of events are expressed and in which cumulative durations of events are specified. Let φ be a predicate and $\mathcal{N}(\varphi, n)$ be defined such that n records the number of occurrences of φ since $T = 0$. Then, we state that the number of occurrences of a system failure f in an interval of k time units is less than or equal to two as follows :

$$\mathcal{N}(f, n) \wedge (T = x) \wedge (n = y) \Rightarrow \Box((T \leq x + k) \Rightarrow (n \leq y + 2))$$

The formula uses two free variables x and y to 'freeze' the values of the time and the number of occurrences of a system failure respectively and compares y with the number of occurrences of a system failure when the time is less than or equal to $x + k$. The cumulative duration of events is specified similarly. Let $\mathcal{D}(\varphi, d)$ be defined such that d records the total duration φ is true since $T = 0$. Then, we assert that the total duration of gas leaks within any interval greater than or equal to k time units must be less than or equal to one-twentieth of the duration of the interval as follows:

$$\mathcal{D}(l, d) \wedge (d = x) \wedge (T = u) \Rightarrow \Box((u + k \leq T) \Rightarrow 20(d - x) \leq T - u)$$

where l denotes that the system is in the leak state. The concepts of the number of occurrences and the durations of predicates are both defined inductively in the logic. In Chapter 3, we will see that this is made possible by the definition of the *next* operator as the next time point at which a state transition occurs if such a time point exists or else it is the time point obtained by incrementing the current time by one and by introducing quantification over flexible variables.

A further aim is to develop the proof and verification techniques for real-time systems. To this end, we develop a calculus of the number of occurrences and durations of predicates which we use for proving functional and timing properties. In particular, we develop a proof technique based on the induction over the number of occurrences of predicates to prove safety properties. To prove a safety property, we prove that it holds when the number of occurrences

of a predicate is zero; and on the assumption that it holds when the number of occurrences is n we prove it holds when the number of occurrences is $n+1$. The calculus is presented as derived rules (i.e. theorems) within the logic.

We also develop a proof method for real-time systems. Real-time systems are characterised by a limited set of resources [JG88]. To reason about shared resources, we introduce a shared processor model of computation. The model allows us to state safety and liveness requirements which in conjunction with the specification of a scheduler must satisfy the original requirements. Different scheduling algorithms can thereby be studied and compared. To illustrate these ideas, we develop a proof system for a real-time extension of a CSP-like language.

1.4 Organisation of the Thesis

The rest of the thesis is organised in three parts.

Part 1 consists of Chapters 2 and 3 in which we introduce the temporal framework. Chapter 2 describes the requirements of a logic which can be used in the specification of the requirements, and the design and development of a real-time system. Various design parameters of the logic are also described and discussed informally based on the requirements of the logic.

Chapter 3 introduces a formal temporal framework for describing the requirements and design of real-time systems. The framework is based on an explicit clock temporal logic. The concepts of the number of occurrences and durations of predicates are introduced and a calculus and proof techniques based on them are developed.

Part 2 consists of Chapters 4 and 5 in which we demonstrate the verification of algorithms where both the specification and the implementation are written in the same language. In Chapter 4, we specify and prove correctness of a fault-tolerant broadcast algorithm using the proof technique developed in Chapter 3. An abstract specification is initially developed and is proved to be safe using induction over the number of occurrences of send and receive operations. The algorithm is then described using low-level data structures and it is proved that the implementation satisfies the abstract properties. Therefore, the implementation is safe. The logic developed in Chapter 3 can also be used for specifying and proving real-time properties of digital circuits. This is illustrated by way of examples in Chapter 5. We specify simple digital circuits and illustrate an alternative proof technique which uses the computation induction principle in ωTL to prove safety properties.

Part 3 consists of Chapter 6 in which we develop a proof method where the specification and the implementation languages are different. As an example, a proof system for a real-time extension of a CSP-like language based on a shared processor model of computation is developed. The proof method is independent of the implementation of scheduling algorithms and can equally be applied to message passing or shared-variable languages. Its applicability is demonstrated by way of examples.

In Chapter 7 we summarise the results of the research presented in this thesis. It also considers related work and topics for further research.

Finally, in Appendices A and B we prove soundness of the logic and the derived rules of the calculus developed in Chapter 3.

Chapter 2

Requirements of the Logic and Design Issues

The development of a logic and a proof method based on it entails many design decisions based on their requirements. These include the phase in the product life cycle during which the logic is to be used, the properties we wish to express, methodological considerations and issues such decidability and completeness. The phase of the software life cycle determines to a considerable extent the level of detail contained in a specification. For example, a specification written in a high-level programming language not only contains 'what' the system does but also 'how' it does it. An abstract specification, on the other hand, captures only what the system does or is expected to do.

The kind of properties one wishes to express determines the primitive operators that the logic must contain. A real-time system is characterised by timing constraints between its inputs and its outputs. Therefore, a logic designed for describing real-time systems must at least be able to capture timing constraints. On the other hand, for systems where timing constraints are not important a logic for describing real-time systems would be an overkill in the sense that a simpler logic could be used.

Other issues which are important are the proof techniques : are the proofs in the logic simpler or more difficult than other logics designed to capture the same class of properties. A logic in which proofs are difficult is likely to be less attractive even if it may contain operators which allow properties to be expressed concisely. Completeness and decidability are also important since for a logic which is complete, one can prove that a statement in it is true or false and for a logic which is decidable a general method exists for determining whether a statement in it is true or false.

Once we have decided on the requirements there are still some detailed design decisions to be made. For example, if timing constraints are to be expressed then should time be introduced explicitly or implicitly in the specifications, and should a continuous or a discrete model of time be used? There is also a choice between different operators and therefore the expressive power of the logic.

In this chapter, we consider some of the requirements and design issues of a logic which can be used for the specification and verification of a real-time system. We also make decisions on the temporal framework developed in this thesis.

2.1 Requirements of the Logic

There are many different requirements to be considered in designing a logic. We briefly describe some of them here.

- **Support For Expressing Requirements and Design Specification**

A specification of a system can be written at various levels of abstraction. User requirements are usually specified at a level of abstraction which makes specifications easy to understand and can form a basis for communicating with the user. The formalism in which the specification is written determines to a considerable degree how abstract the specification is. Therefore, a formalism for expressing requirements must be able to express abstract requirements. When the specification has been accepted by the user, design decisions are made which must be proved to meet the requirements. For this, it is desirable that the design decisions are expressed in the same logic as the requirements so that correctness can be shown by logical implication.

- **Application Domain**

Properties of a real-time system are typically classified into that of *bounded-invariance* and *bounded-response*. The definition of each is given by the following informal statements [HMP91a].

- A *bounded-invariance* property states that something does not happen within some time period.
- A *bounded-response* property states that something happens within some time period.

We can view the informal definitions of the bounded-invariance and the bounded-response properties as stating that something does not happen or happens at least once within some time period. In the specification of some real-time systems, this may not be sufficient. We may also wish to state exactly how many times an event must occur. Therefore, it is desirable that a logic for the specification of a real-time system is able to express bounded-response properties which may include the number of times an event must occur.

Real-time properties are specified using both discrete and continuous time. It is therefore also desirable that the logic is expressive enough to incorporate both models of time.

- **Methodological Considerations**

The analysis and the specification of the requirements of a system is an incremental process. It is often the case that complete information about the system may not be available or the application may be incorrectly or partially understood. Therefore, any formalism used for describing requirements must be able to describe partial and incomplete information about the system. It must also be a proof theoretical approach. This would enable users to prove desirable system properties early in the design stages and remove any inconsistencies before an implementation can begin.

- **Formal Semantics**

Existing formalisms for describing requirements of a system can roughly be grouped into that of formal and semi-formal notations. Semi-formal notations such as JSD [Jac83] and Core [Mul76] lack formal semantics and therefore do not have any notion of deducibility and proof. Therefore, one cannot analyse a specification to check for properties such as consistency. If a notation is to be used for the specification and verification of a system it must have a formal semantics which is sound.

- **Proof Techniques**

When a system is being designed, there are properties which have to be proved in order to ascertain that the design decisions being made satisfy the user's requirements. A logic must therefore not only be used for capturing the user's requirements but also in proving properties of the system being designed. It has already been observed that temporal reasoning, even to a person experienced in logic, can be difficult and tedious [MP89b]. It has also previously been observed that the proof and verification techniques for the formalisms

which use a discrete model of time are significantly simpler than those using a continuous one [HMP91b]. This suggests conflicting requirements. On one hand, a continuous model of time is required for expressing the user's requirements which are often specified in continuous time and on the other, a discrete model is preferred since the proof techniques using it are simpler than those for continuous time. It is therefore desirable to be able to develop proof techniques which can be applied irrespective of whether time is discrete, continuous or dense.

- **Modular Logic**

A logic must be modular so that a wide range of applications can be described in it. It must also include only those features which are pertinent to its definition. Other issues such as built-in data types must be dealt with at the level of language design. This allows many varied and different specification languages to be designed around a common logic.

- **Consistency, Completeness and Decidability**

To be useful a logic must be consistent. That is, two valid statements in the theory must not contradict each other and for any two contradictory statements one should not be provable. A complete theory, on the other hand, is one in which given two contradictory statements one can be proved to be true. Another important issue from a practical point such as building automated verification tools is decidability. A logic is said to be decidable iff there is a general method for determining whether a statement in the logic can be proved or disproved.

2.2 Design Issues

In this section, we consider some design parameters of a logic and a proof method for real-time systems.

- **Explicit-Clock and Bounded-Operator Approaches**

As already mentioned in Chapter 1, there are two approaches to introducing time within a temporal logic : the explicit-clock and the bounded-operator approaches. Therefore, one can adopt either of these two approaches.

  - In the *explicit-clock* approach a variable T is introduced to denote time.

T observes axioms of time, in particular, T does not decrease from one state to its successor and it eventually increases.

- In the *bounded-operator* approach new temporal operators having time bounds are introduced. The explicit-clock approach has several advantages over the bounded one.

  Firstly, one can introduce bounded temporal operators within an explicit-clock framework. For example, we can define $\Diamond_{\leq k}\varphi$ as $(T = x) \Rightarrow \Diamond(\varphi \wedge (T \leq x + k))$ and $\Box_{\leq k}\varphi$ as $(T = x) \Rightarrow \Box((T \leq x + k) \Rightarrow \varphi)$. On the other hand, the bounded-operator approach does not include any explicit reference to a conceptual global clock and therefore no reference can be made to it in the specifications.

  Secondly, no new proof rules are required for the verification of a real-time system using an explicit clock temporal logic. We can use the proof methods developed for the untimed temporal logics to reason about timing properties.

  Another advantage of using an explicit clock temporal logic, and the one which we will be exploiting in this thesis, is that the concepts such as the duration of a predicate can be defined syntactically. In the bounded-operator approach, we cannot refer to T: hence the concept of a duration and its calculus have to be defined semantically. The axioms for the durations of predicates have to be proved sound and complete with respect to the semantics. The proof rules in general will be complex because of the interaction between the calculus and the bounded temporal operators. Similar arguments hold for the calculus of the number of occurrences of predicates. We will demonstrate in the next chapter how the calculus is used for proving timing and functional properties.

• Mathematical Models of Time

The underlying semantic model for both the explicit-clock and the bounded-operator approaches consists of sequences or tree structures of (time,state) pairs. One can therefore make various choices regarding the mathematical model of time. Different models allow different sets of valid sequences or tree structures of (time,state) pairs.

- In an *interval* model of time each state has a duration. Therefore, when we say an atomic predicate φ holds it means φ holds throughout some interval. The meaning of $\Box\varphi$ is that φ holds in all subintervals and $\Diamond\varphi$ means that φ holds in some subinterval. The basic operator in an interval temporal logic is the *chop* ($^\frown$) from which $\Box$ and $\Diamond$ can be defined.

The meaning of $\varphi\hat{}\psi$ in an interval is that it can be partitioned into two sub-intervals of which φ holds in the first and ψ in the second.

In a *point* model of time, each state is defined at a point in time. Therefore, when we say an atomic predicate φ holds it means φ holds at a point in time. The basic operators in the logic are *until* ($\mathcal{U}$) and *since* ($\mathcal{S}$). $\varphi\mathcal{U}\psi$ means that φ holds at all points in time until ψ becomes true. $\varphi\mathcal{S}\psi$ is analogous for past instances. $\Diamond\varphi$ can be defined in terms of *until* as $true\mathcal{U}\varphi$, from which $\Box\varphi$ can be defined. Past temporal operators, $\boxminus$ and $\blacklozenge$ are defined using *since*.

- In a *linear* model of time, points or intervals are viewed as a linear sequence. In the point view, this means that time observes the axiom of linearity, i.e. given two time points either one precedes the other or they are equal. In the interval model, given any two intervals either one precedes the other or they overlap.

  In a *branching* model of time, points are viewed as branching sets of different possible futures and therefore do not obey the axiom of linearity. In the linear model of time there is only one future to consider. Therefore, the statement $\Diamond\varphi$ means that φ is true during that one future and $\Box\varphi$ means that φ is always true in that one future. Since only one future is being considered, "φ is sometimes true" is equivalent to "it is not the case that φ is always false". This means that $\Diamond\varphi$ and $\neg\Box\neg\varphi$ are equivalent. On the other hand, in a branching-time logic there are many futures. Therefore, $\Diamond\varphi$ means φ holds sometime during every possible future and $\Box\varphi$ means that φ always holds in all possible futures. $\neg\Box\neg\varphi$ is then interpreted as "it is not the case that $\neg\varphi$ is true in all possible futures" which is not equivalent to "φ is sometime true in all possible futures". Therefore, $\Diamond\varphi$ and $\neg\Box\neg\varphi$ are not equivalent in a branching-time temporal logic.

  The other difference between branching and linear time models is to do with how concurrency and nondeterminism are modelled. In a linear-time temporal logic, the computations of any two concurrent systems are modelled by the nondeterministic interleaving sequences of events. When we state a property of a system, we consider all possible arbitrary interleavings. Nondeterministic events are also interleaved. In a branching-time logic, nondeterministic operations are modelled as different branches, each determining a different future. Therefore, we can differentiate between events which are nondeterministic and those which are not. A detailed comparision between linear and branching-time temporal logics can be found in [Lam80, EH86].

- In a *discrete* model of time, the time associated with each state is not precise. Therefore, the time at which an event occurs and duration of a

predicate cannot be measured accurately.

In a *continuous* model of time, the time of each state is measured with arbitrary precision. Therefore, we can precisely state the time of an occurrence of an event and the durations of predicates. Integers are used for the discrete domain and reals for the continuous one.

A propositional temporal logic which uses a continuous model of time and which restricts timing expressions to $x + c$ where x is a timing variable and c is a constant is undecidable [AH89, AH90]. On the other hand, a temporal logic which uses discrete time and which restricts quantification to the outermost level (as in XCTL) or which restricts timing expressions to $x + c$ where x is a timing variable and c is a constant (as in TPTL) is decidable. A detailed discussion on the decidability of real-time temporal logics can be found in [AH89, HLP90].

In a temporal logic which uses a discrete model of time and which has a *next* operator, one can use a computational induction principle to prove safety properties. To prove a safety property, we prove it at $T = 0$, and on the assumption that it holds at $T = k$, we prove that it holds at $T = k + 1$. In a temporal logic with a continuous model and which does not have a *next* operator an alternative induction principle is required to prove safety properties.

- **Expressive Power of the Logic**

There are various choices one can make regarding syntactic features and therefore the expressive power of the logic.

- One can use only the *future* fragment of the temporal logic or both future and *past* fragments. It was established in [GPSS80] that the future fragment is expressively complete. However, there are some specifications which can be expressed more naturally using the past operators [KVR83]. For example, to state that an event φ is preceded by ψ, we write $\Box(\varphi \Rightarrow \diamondsuit\psi)$ using the past temporal operators which is symmetrical to the formula $\Box(\varphi \Rightarrow \diamondsuit\psi)$ which states that ψ follows an occurrence of φ. While adding the past operators does not increase the expressive power, the advantage outweighs the cost since the proof system and the decision procedures developed for the future fragment can be used by replacing the future temporal operators by their past symmetrical ones.
- The other choice concerns the basic operators of the logic. Here, there are choices such as using *henceforth* and *eventually* ($L(\Box, \diamondsuit)$), or using fixed point schema and *next* ($L(\nu\xi.\chi(\xi), \bigcirc)$), or using *until* and *next*

($L(\mathcal{U}, \bigcirc)$), or only *until* ($L(\mathcal{U})$). There are many properties such as precedence which cannot be expressed using $L(\Box, \Diamond)$. For example, the statement "ψ must be preceded by φ" cannot be expressed using $\Box$ and $\Diamond$. Using the *until* operator, we would assert it as $(\neg\psi)\mathcal{U}\varphi$. It is also easy to see that $\Box$ and $\Diamond$ can be defined in terms of $\mathcal{U}$ but not vice-versa. Using the fixed point and the *next* operator one can define the *until* operator from which others can be defined. It is not possible to define the fixed point operator using *until*. However, the semantics of the fixed point operator is not as straightforward as that of *until*. Also, the compositional semantics of programming languages using the fixed point operator is more difficult [Fis87]. This leaves us with only two choices : $L(\mathcal{U}, \bigcirc)$ or $L(\mathcal{U})$. The *until* operator can be defined in terms of the *next* operator as :

$$\varphi\mathcal{U}\psi \triangleq \psi \vee (\varphi \wedge \bigcirc(\varphi\mathcal{U}\psi))$$

but the *next* operator cannot be defined in terms of *until*. Some properties expressible in terms of *next* therefore cannot be stated using *until*. However, the choice between $L(\mathcal{U}, \bigcirc)$ and $L(\mathcal{U})$ depends on other issues such as the temporal semantics of programs. Consider the following example concerning two programs S_1 and S_2

$S_1 : x := 0; x := 0; x := 1$ and
$S_2 : x := 0; x := 1$

The two programs are identical except for an additional $x := 0$ statement in S_1. Now, assuming that each program step takes one time unit, the semantics of S_1 in $L(\mathcal{U}, \bigcirc)$ is given by

$$\bigcirc(x = 0 \wedge \bigcirc(x = 0 \wedge \bigcirc x = 1))$$

and of S_2 as :

$$\bigcirc(x = 0 \wedge \bigcirc x = 1)$$

This difference in semantics was attributed to the fact that $L(\mathcal{U}, \bigcirc)$ is sensitive to stuttering and therefore the semantics of the two programs are not fully abstract [BKP86]. It is shown in [BKP86] that for a fully abstract semantics the logic must be insensitive to finite stuttering but must be sensitive to infinite stuttering. In [Fis87], a temporal logic EL is proposed in which it takes the next state in the sequence at which a change in state occurs unless no state change ever occurs. Under this

interpretation, the semantics of the two programs are same. In [BKP86], a temporal logic based on reals is given which does not have a *next* operator and therefore does not differentiate between the semantics of S_1 and S_2. However, a logic based on reals is undecidable and the proof methods are more difficult than those based on discrete time.

Another syntactic choice concerns the quantifiers. We can introduce quantification over only global variables, or over both global and flexible variables or no quantification at all. Unrestricted use of quantifiers leads to undecidability [HLP90]. To achieve decidability, one needs to restrict quantification to the outermost level. On the other hand, there are some interesting properties which require nested quantification and cannot be reduced to the outermost quantification. In [Sza86], it is also shown that a first-order temporal logic with $\Box$ (henceforth), $\Diamond$ (eventually) and $\bigcirc$ (next) with the constant "0", a unary successor function (*succ*), and two binary symbols ($+$ and $*$) has no sound axiomatisation which is both complete and finite. The result is strenthened in [Kro90] where it is shown that a temporal logic with equality, $\bigcirc$ (next) and $\Box$ (henceforth) is incomplete in

a) a language which allows quantification over global variables and has at least two binary function symbols,

b) and in a language which allows quantification over global and local variables and has an arbitrary (possibly empty) set of function and predicate symbols.

Yet, another syntactic choice involves restricting timing expressions to the form $x + c$ where x is a timing variable and c is a constant to achieve decidability. However, restricting expressions restricts our ability to write some common real-time requirements.

• Computation Models

Having discussed how states are represented and a formula in the logic interpreted, we consider choices regarding how actions are modelled in a computation.

- In the *interleaving* model, two parallel systems are represented as an interleaving of their actions. Under this choice, the semantics of a program are given by a set of sequences of states generated by executing one action at any time. This view is a very simple one since we need to consider the effects of only one action at any time. However, it is not a natural representation of concurrent systems. In particular, it permits

actions from one process to be executed while other processes may never be executed even though their actions may be enabled (i.e. ready for execution). To exclude this possibility various requirements of fairness are added. Fairness conditions are used for stating that an action that is enabled will eventually be executed. The interleaving model is quite satisfactory for analysing and proving properties of concurrent systems where time is unimportant. However, it allows arbitrary delays between any two actions. In a real-time system, delays between execution of actions are determined by the availability of resources and is not arbitrary. One approach taken in [HMP91a] is to consider an interleaving model in which actions are instantaneous with minimal and maximal delays. However, this does not seem to be fully satisfactory since it is difficult to know the delays in advance of the implementation.

- An alternative model that has been proposed in the literature [SM81, KSR+85, HGR87] is the *maximal parallelism* model. In this model, it is assumed that each process has its own processor. Parallel actions can overlap in time unless prohibited by synchronisation factors. The model is more realistic because there are no unrealistic assumptions about the waiting periods of processors. However, for real-time systems, resources may be limited and therefore the model is not fully satisfactory.

2.3 Discussion

Having discussed various parameters of a logic and a proof method for the specification and verification of a real-time system, we are now ready to introduce and justify the choice of the approach taken in this thesis.

The temporal logic we develop is based on the introduction of a variable denoting an explicit global clock, for the following reasons:

1. We can define bounded temporal operators within an explicit-clock framework.
2. The concepts of a duration and an occurrence of a predicate and their calculus can be defined within the logic instead of semantically. Therefore, we do not need to prove relative completeness of the calculus.
3. Existing proof methods developed for the untimed temporal logics can be used by adding appropriate axioms and proof rules for time.

4. It allows us to leave the choice of whether discrete, continuous, or dense time model is to be used to the application level while still leaving the logic unchanged.

The logic is based on a point view of time. Intervals can be constructed from time points and properties stated using interval schemas. We use a linear-time semantics as opposed to a branching-time semantics because of its simplicity. The price we pay for using a simpler model is that we cannot distinguish between nondeterminism arising from making a nondeterministic choice and that introduced by nondeterministic interleaving to represent concurrency.

We define the *next* operator in the logic to mean the next time point at which a state transition occurs if such a time point exists or else it is defined as the time obtained by incrementing the current time by one. This allows us to achieve an abstract semantics of programming languages while still retaining a *next* operator and the use of a computational induction principle to prove safety properties. It also makes the logic insensitive to whether the underlying time model is discrete, continuous or dense. We can therefore use a time model most appropriate to the application area.

The logic contains both the past and the future fragments. This does not extend the expressive power of the logic but allows us to state many properties naturally. It has already been observed in [MP89a] that the cost of adding the past fragment is modest since it only involves extending the results (model checking, completeness of the logic) obtained for the future fragment. We allow quantification over both global and flexible variables. We will see in Chapter 3 that the quantification over flexible variables is required for defining the calculus of durations and the number of occurrences of predicates. We also allow all forms of arithmetic expressions using timing variables. The result is that the logic is undecidable and incomplete.

The interleaving and the maximal parallelism computation models are both not fully satisfactory for the specification and verification of real-time systems. The interleaving model allows unrealistic delays in waiting times of processors while maximal parallelism assumes no waiting time since all processes have their own processors. A key characteristic of real-time systems is the limited set of resources. To model resources we introduce a shared processor model of computation. Using this model we can separate the scheduling assumptions from the proof system. We can then specify and implement a scheduler which can be put in parallel with an implementation and can be analysed for timing properties. Different scheduling algorithms can be studied and their performance compared.

Chapter 3

The Temporal Framework

The two most important requirements identified in Chapter 2 for a logic used in the specification and verification of a real-time system were that it must be expressive enough to specify a large class of real-time properties and that it should be possible to develop proof techniques for the verification of these systems. In this chapter, we introduce a temporal logic ωTL to meet these requirements. Its main features are :

1. A distinguished variable represents current time. Therefore, existing verification techniques developed for reactive systems can be used. Bounded temporal operators can be defined as abbreviations within the logic.

2. It is independent of whether the time model is discrete, continuous or dense. The logic is therefore expressive enough to be used in the specification of systems where either a discrete, a continuous or a dense model is required.

3. The *next* operator in the logic is defined as the next time point at which a state transition occurs if such a time point exists in the future or else it is the time point obtained by adding one to the current time. The logic is therefore insensitive to stuttering which occurs for a finite interval of time but is sensitive to stuttering which occurs for an infinite time interval.

4. It allows the definition of a duration and the number of occurrences of a predicate to be introduced and their calculus developed as theorems within the logic. The calculus is used for proving safety properties of real-time systems.

The rest of the chapter is organised as follows. We present the syntax of the logic in Section 3.1 and the semantics in Section 3.2. This is followed by the proof theory in Section 3.3. In Section 3.4, a calculus of durations and the number of occurrences of predicates is introduced. Derived rules for the durations and the number of occurrences of predicates are given and a technique is developed for proving the timing properties of a real-time system.

3.1 Syntax

The basic symbols of the language are constants, variables, function and predicate symbols. We use the usual set of propositional connectives: negation ($\neg$), implication ($\Rightarrow$) and the first-order universal quantifier ($\forall$). The modal connectives used are weak next ($\odot$), weak previous ($\oslash$), weak until (U) and weak since (S). Formally, the following are the primitive symbols of the language.

Alphabet

- Constants : A denumerable, possibly infinite set of constants.
- Distinguished variables : T is the only distinguished variable.
- Local variables : A denumerable, possibly infinite set of local variables denoted by VAR.
- Global variables : A denumerable, possibly infinite set of global variables.
- Function symbols : A denumerable, possibly infinite set of function symbols.
- Predicate symbols : A denumerable, possibly infinite set of predicate symbols.
- Logical operators : $\neg$, $\Rightarrow$, $\forall$, $\odot$, $\oslash$, U and S.
- Punctuation symbols : (and).

Formation Rules

- Terms
  1. Variables and constants are terms.
  2. Time is the distinguished variable T.

3. If $t_1, \ldots, t_n$ are terms and F is an n-place function symbol then $F(t_1, \ldots, t_n)$ is a term.
4. If t is a term then so are $\odot t$ and $\oslash t$.
5. Nothing else is a term.

- Formulae

1. If $t_1, \ldots, t_n$ are terms and P is an n-place predicate symbol then $P(t_1, \ldots, t_n)$ is a formula.
2. If φ is a formula then so are $\neg\varphi$, $\odot\varphi$, and $\oslash\varphi$.
3. If φ and ψ are formulae then so are $\varphi \Rightarrow \psi$, $\varphi \mathsf{U} \psi$ and $\varphi \mathsf{S} \psi$.
4. If φ is a formula and x is a global variable, then $\forall x.\varphi$ is also a formula.
5. If φ is a formula then so is (φ).
6. Nothing else is a formula.

3.2 Semantics

We use a time domain with values from a carrier set Γ and defined as the tuple $(\Gamma, <, =, +, *, 0, 1)$, where "$<$" is a precedence relation on Γ, "$=$" the equality relation, "$+$" the addition operator, "$*$" the multiplication operator and constants "0" and "1" which are members of the set Γ. It satisfies the following axioms where τ, τ_1, τ_2 and τ_3 belong to Γ.

T1. $\forall \tau_1, \tau_2, \tau_3.(\tau_1 < \tau_2 \wedge \tau_2 < \tau_3 \Rightarrow \tau_1 < \tau_3)$
T2. $\forall \tau.\neg(\tau < \tau)$
T3. $\forall \tau_1, \tau_2.((\tau_1 = \tau_2) \vee (\tau_1 < \tau_2) \vee (\tau_2 < \tau_1))$
T4. $\forall \tau_1. \exists \tau_2.\tau_1 < \tau_2$

Axiom T1 states that the precedence relation is transitive and Axiom T2 states that it is irreflexive. Axiom T3 states that any two time points are either equal or one precedes the other, i.e. time points are linearly ordered. Axiom T4 asserts that every time point has a successor.

T5. $\forall \tau_1, \tau_2.(\tau_1 + \tau_2 = \tau_2 + \tau_1)$
T6. $\forall \tau_1, \tau_2, \tau_3.((\tau_1 + \tau_2) + \tau_3 = \tau_1 + (\tau_2 + \tau_3))$
T7. $\forall \tau.\tau + 0 = \tau$

T8. $\forall \tau, \tau_1, \tau_2.(\tau + \tau_1 = \tau + \tau_2 \Rightarrow \tau_1 = \tau_2)$
T9. $\forall \tau_1, \tau_2.(\tau_1 + \tau_2 = 0 \Rightarrow \tau_1 = 0 \wedge \tau_2 = 0)$
T10. $\forall \tau_1, \tau_2. \exists \tau.(\tau_1 = \tau_2 + \tau \vee \tau_2 = \tau_1 + \tau)$

Axiom T5 asserts that addition is commutative and T6 states that it is associative. Axiom T7 states that any time point added to the constant "0" returns the same time point. Axiom T8 states that the addition operator is injective. Axiom T9 asserts that time points are positive and T10 asserts that there is an absolute difference between any two time points.

T11. $\forall \tau_1, \tau_2.(\tau_1 * \tau_2 = \tau_2 * \tau_1)$
T12. $\forall \tau_1, \tau_2, \tau_3.((\tau_1 * \tau_2) * \tau_3 = \tau_1 * (\tau_2 * \tau_3))$
T13. $\forall \tau.\tau * 1 = \tau$
T14. $\forall \tau_1, \tau_2.(\tau_1 * \tau_2 = 0 \Rightarrow \tau_1 = 0 \vee \tau_2 = 0)$
T15. $\forall \tau_1, \tau_2, \tau_3.(\tau_1 * (\tau_2 + \tau_3) = \tau_1 * \tau_2 + \tau_1 * \tau_3)$
T16. $0 \neq 1$

Axiom T11 asserts that multiplication is commutative and T12 states that it is associative. Axiom T13 states that any time point multiplied by the constant "1" returns the same time point. Axiom T14 asserts that if the multiplication of two time points results in 0 then at least one of them must be the constant "0". Axiom T15 states that multiplication distributes over addition and T16 states that the constants "0" and "1" are not equal.

Remark 3.2.1 : Note that we have not made any assumptions about whether Γ is discrete, continuous, or dense, in line with our requirements of the logic described in Chapter 2.

■

Let Γ be a time domain satisfying the axioms T1-T16 and $STATE$ be a set of mappings from the set of local variables VAR to the set of values VAL. Each member of the set $STATE$ is called a state and it assigns values to local variables.

$$STATE = VAR \rightarrow VAL$$

A behaviour σ is a mapping from Γ to $STATE$. Given a behaviour σ and a time point $\tau \in \Gamma$, $\sigma(\tau)$ returns the state at the time point τ.

We consider only those behaviours in which there are not infinitely many state changes in a finite interval of time (finite variability : [BKP86]).

Definition 3.2.1 (Validity of a Behaviour) A behaviour σ is valid iff for all time points τ if a state change occurs at τ' and $\tau < \tau'$ then there exists a time point τ'' such that $\tau < \tau''$ and the state at τ'' is not equal to the state at τ and for all time points between τ and τ' the state is equal to that at τ.

$$\forall \tau.((\exists \tau'.(\tau < \tau' \wedge \sigma(\tau') \neq \sigma(\tau))) \Rightarrow \exists \tau''.(\tau < \tau'' \wedge \sigma(\tau) \neq \sigma(\tau'') \wedge \forall \tau'''.(\tau < \tau''' < \tau' \Rightarrow \sigma(\tau''') = \sigma(\tau))))$$

■

In the definition of a valid behaviour and in the rest of the thesis we use the phrase "a state change occurs at τ" or "a state transition occurs at τ" to mean that either there does not exist τ' such that $\tau' < \tau$ (i.e. $\tau = 0$) or there exists τ' such that $\tau' < \tau$ and the state at τ' is not equal to the state at τ and for all τ'' such that $\tau' < \tau'' < \tau$ the state at τ' is equal to the state at τ''.

Let Σ denote the set of all valid behaviours. Define a function ts which takes a valid behaviour as its argument and returns an ordered sequence of time points at which a state transition occurs or after which no state transitions occur and is equal to the previous time point in the sequence plus one.

$$ts : \Sigma \rightarrow (\mathbf{N} \rightarrow \Gamma)$$

$$\begin{aligned} ts(\sigma)(i) &= 0 \text{ if } i = 0 \\ &= \tau' \text{ if } 0 < i \wedge ts(\sigma)(i-1) < \tau' \wedge \sigma(\tau') \neq \sigma(ts(\sigma)(i-1)) \\ &\qquad \wedge \forall \tau.(ts(\sigma)(i-1) < \tau < \tau' \Rightarrow \sigma(\tau) = \sigma(ts(\sigma)(i-1))) \\ &= ts(\sigma)(i-1) + 1 \text{ otherwise} \end{aligned}$$

Definition 3.2.2 (Length of a Sequence) Let s be a sequence:

$$s : s_0, s_1, s_2, \ldots$$

If s is finite, i.e.

$$s : s_0, s_1, s_2, \ldots, s_k$$

then we define the length of s to be $|s| = k + 1$. Otherwise $|s| = \omega$, the first infinite ordinal.

■

A model $M = (I, \sigma, \alpha, i)$ consists of an interpretation function I, a valid behaviour σ, an assignment α and a nonnegative integer i.

- The interpretation function I specifies a nonempty set D and assigns to a constant a fixed element in D, for an n-place function symbol I assigns a function from D^n into D, and for an n-place predicate symbol assigns an n-ary relation over D, i.e. a function from D^n to the truth values **tt** and **ff**. The set of values VAL is a subset of D.
- σ is a valid behaviour of a real-time system.
- The assignment α assigns a value to each global variable.
- i refers to the index into the sequence of time points returned by $ts(\sigma)$.

The meaning of a formula φ and a term t under M is given inductively and is abbreviated as $\varphi|_i^\sigma$ and $t|_i^\sigma$ respectively, with I and α implicit in the definition.

- A constant

  $c|_i^\sigma = I(c)$

  The value of a constant c is given by applying I to c.

- A local variable

  $x|_i^\sigma = \sigma(ts(\sigma)(i))(x)$

  The value of a local variable x is the one assigned at the i^{th} time point in the sequence returned by $ts(\sigma)$.

- A global variable

  $x|_i^\sigma = \alpha(x)$

  The value of a global variable x is the one assigned by α.

- Time (or T)

  $T|_i^\sigma = ts(\sigma)(i)$

  The value of the distinguished variable T is the i^{th} time point in the sequence returned by $ts(\sigma)$.

- An n-ary function

  $F(t_1, \ldots, t_n)|_i^\sigma = I(F)(t_1|_i^\sigma, \ldots, t_n|_i^\sigma)$

  i.e. the value is given by the application of $I(F)$ to the values of the arguments.

- Weak next term

$$\odot t|_i^\sigma = \begin{cases} t|_i^\sigma & \text{if } |ts(\sigma)| = i+1 \\ t|_{i+1}^\sigma & \text{if } |ts(\sigma)| \neq i+1 \end{cases}$$

$\odot t$ evaluates to the value of t at the next time point in the sequence returned by $ts(\sigma)$ if one exists or else it is equal to the value of t at the time point at which $\odot t$ is being evaluated.

- Weak previous term

$$\oslash t|_i^\sigma = \begin{cases} t|_i^\sigma & \text{if } i = 0 \\ t|_{i-1}^\sigma & \text{if } i \neq 0 \end{cases}$$

$\oslash t$ evaluates to the value of t at the previous time point in the sequence returned by $ts(\sigma)$ if one exists or else it is equal to the value of t at the time point at which $\oslash t$ is being evaluated.

- An n-ary predicate

$$P(t_1, \ldots, t_n)|_i^\sigma = I(P)(t_1|_i^\sigma, \ldots, t_n|_i^\sigma)$$

i.e. the value is given by applying $I(P)$ to the values of the arguments.

- Negation

$$\neg\varphi|_i^\sigma = \mathbf{tt} \text{ iff } \varphi|_i^\sigma = \mathbf{ff}$$

$\neg\varphi$ holds iff φ does not hold.

- Implication

$$(\varphi \Rightarrow \psi)|_i^\sigma = \mathbf{tt} \text{ iff } \varphi|_i^\sigma = \mathbf{ff} \text{ or } \psi|_i^\sigma = \mathbf{tt}$$

$\varphi \Rightarrow \psi$ holds iff φ does not hold or ψ holds.

- Universal quantification over global variables

$\forall x.\varphi|_i^\sigma = \mathbf{tt}$ iff for every $d \in D$, $\varphi(d/x)|_i^\sigma = \mathbf{tt}$, where $\varphi(d/x)$ is obtained by substituting d for all free occurrences of x in φ

- Weak next

$$\odot\varphi|_i^\sigma = \mathbf{tt} \text{ iff } |ts(\sigma)| = i+1 \text{ or } \varphi|_{i+1}^\sigma = \mathbf{tt}$$

$\odot\varphi$ holds iff φ holds at the next time point in the sequence returned by $ts(\sigma)$ or it is being evaluated at the last time point in the sequence returned by $ts(\sigma)$.

- Weak until

$$\varphi \mathsf{U} \psi|_i^\sigma = \mathbf{tt} \text{ iff } \begin{array}{l} \text{for every } j,\ i \le j < |ts(\sigma)|,\ \varphi|_j^\sigma = \mathbf{tt} \\ \text{or for some } k,\ i \le k < |ts(\sigma)|,\ \psi|_k^\sigma = \mathbf{tt} \\ \qquad \text{and for every } j,\ i \le j < k,\ \varphi|_j^\sigma = \mathbf{tt} \end{array}$$

$\varphi \mathsf{U} \psi$ holds iff φ holds at all future time points in the sequence returned by $ts(\sigma)$ (including the time point at which $\varphi \mathsf{U} \psi$ is being evaluated) until a time point in the sequence at which ψ holds or φ holds at all future times points in the sequence returned by $ts(\sigma)$.

- Weak previous

$$\oslash\varphi|_i^\sigma = \mathbf{tt} \text{ iff } i = 0 \text{ or } \varphi|_{i-1}^\sigma = \mathbf{tt}$$

$\oslash\varphi$ holds iff φ holds at the previous time point in the sequence returned by $ts(\sigma)$ or it is being evaluated at the first time point in the sequence returned by $ts(\sigma)$.

- Weak since

$$\varphi \mathsf{S} \psi|_i^\sigma = \mathbf{tt} \text{ iff } \begin{array}{l} \text{for every } j,\ 0 \le j \le i,\ \varphi|_j^\sigma = \mathbf{tt} \\ \text{or for some } k,\ 0 \le k \le i,\ \psi|_k^\sigma = \mathbf{tt} \\ \qquad \text{and for every } j,\ k < j \le i,\ \varphi|_j^\sigma = \mathbf{tt} \end{array}$$

$\varphi \mathsf{S} \psi$ holds iff φ held at all previous time points in the sequence returned by $ts(\sigma)$ (including the time point at which $\varphi \mathsf{S} \psi$ is being evaluated) since a time point in the sequence at which ψ held or φ held at all previous times points in the sequence returned by $ts(\sigma)$.

A wff φ is said to be satisfied by $M = (I, \sigma, \alpha, i)$ denoted by $M \models \varphi$ iff $\varphi|_i^\sigma = \mathbf{tt}$. A wff φ is logically valid iff φ is true for every model. This is denoted by $\models \varphi$.

The logic defined in this section is insensitive to stuttering which occurs for a finite time interval but is sensitive to stuttering which occurs for an infinite time interval. This is because a formula in ωTL is evaluated at only those time points in the sequence returned by ts. The function ts is defined such that when applied to a valid behaviour σ returns an ordered sequence of time points at which a state transition occurs or after which no state transitions occur and

is equal to the previous time point in the sequence plus one. In Section 3.5, we will see that alternative definitions of the function ts and $|ts(\sigma)|$ lead to a logic which is either insensitive to any stuttering, or which is sensitive to stuttering which occurs for a finite interval of time only, or which is sensitive to stuttering which occurs for both finite and infinite time intervals.

3.3 Proof Theory

In the previous section, we defined valid formulae in the logic. We now turn our attention to the proof theory. We describe a set of axioms and a set of inference rules which allow us to derive other valid formulae by means of a proof. A formula which is either an axiom or one which is derived using the set of inference rules is called a theorem. If φ is a theorem then we denote it by $\vdash \varphi$, i.e. φ is provable in the logic.

But first, we state some useful abbreviations. In the abbreviations and in the rest of the thesis, the symbol $\triangleq$ is used to mean *syntactically equivalent to.*

PL1. $\varphi \vee \psi \triangleq \neg\varphi \Rightarrow \psi$	PL2. $\varphi \wedge \psi \triangleq \neg(\varphi \Rightarrow \neg\psi)$
PL3. $true \triangleq \varphi \vee \neg\varphi$	PL4. $false \triangleq \neg true$
PL5. $\varphi \Leftrightarrow \psi \triangleq (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$	
DF1. $\Box\varphi \triangleq \varphi \mathsf{U} false$	DP1. $\boxminus\varphi \triangleq \varphi \mathsf{S} false$
DF2. $\Diamond\varphi \triangleq \neg\Box\neg\varphi$	DP2. $\diamondminus\varphi \triangleq \neg\boxminus\neg\varphi$
DF3. $\varphi \mathcal{U} \psi \triangleq \varphi \mathsf{U} \psi \wedge \Diamond\psi$	DP3. $\varphi \mathcal{S} \psi \triangleq \varphi \mathsf{S} \psi \wedge \diamondminus\psi$
DF4. $\oplus\varphi \triangleq \neg \odot \neg\varphi$	DP4. $\ominus\varphi \triangleq \neg \oslash \neg\varphi$
DF5. $\varphi \mathcal{U}^{+} \psi \triangleq \varphi \wedge \varphi \mathcal{U} \psi$	DP5. $\varphi \mathcal{S}^{+} \psi \triangleq \varphi \wedge \varphi \mathcal{S} \psi$
DF6. $\varphi \mathsf{U}^{+} \psi \triangleq \varphi \wedge \varphi \mathsf{U} \psi$	DP6. $\varphi \mathsf{S}^{+} \psi \triangleq \varphi \wedge \varphi \mathsf{S} \psi$

PL1-PL5 are standard definitions for propositional calculus. $\Box\varphi$ (henceforth) is true iff φ holds at all future state transition points and at all time points obtained by adding positive integer values to the last state transition point. $\Diamond\varphi$ (eventually) is true iff φ is true at some future state transition point or at some time point which is an integral number of time units after the last state transition point. $\varphi \mathcal{U} \psi$ (strong until) holds iff ψ holds at some future state transition point or at some time point which is an integral number of

time units after the last state transition point and φ holds until that instant. $\oplus\varphi$ (strong next) holds iff φ holds at the next state transition point if such a time point exists or no state transitions occur and φ is true at the time point obtained by adding one to the current time. $\varphi\mathcal{U}^{+}\psi$ (strict until) holds iff φ and $\varphi\mathcal{U}\psi$ hold. $\varphi\mathsf{U}^{+}\psi$ (strict weak until) holds iff φ and $\varphi\mathsf{U}\psi$ hold. Definitions DP1-DP6 are symmetric with DF1-DF6 for the past fragment.

ωTL is a combination of a 'next-change' and a 'next-time' temporal logic in the sense that it considers *next* as the next time point at which state changes if there exists such a time point in the future or else *next* is defined as the time point obtained by incrementing the current time by one. When we say "henceforth φ is true" it means that φ is true at at all future state transition points and at all time points obtained by adding positive integer values to the last state transition point. The property 'whenever φ is true then ψ is true one time unit later' is expressed in RTTL or XCTL in which *next* is defined as the time point obtained by adding one to the current time as :

$$\varphi \wedge (T = x) \Rightarrow \Box((T = x + 1) \Rightarrow \psi)$$

In ωTL, it is expressed as :

$$\varphi \wedge (T = u) \wedge (u \leq x < \odot T) \Rightarrow \Box((T \leq x + 1) \wedge \odot(T > x + 1) \Rightarrow \psi)$$

In the above formula, u is the time point at which a state change occurred or is equal to the time at the last state transition plus some positive integer value and x is any time point between u and the next state transition point, if one exists or if no state transition occur then x is between u and $u + 1$. Also, note that unlike RTTL and XCTL, time in ωTL need not be discrete. Therefore, we can write formulas such as :

$$\varphi \wedge (T = u) \wedge (u \leq x < \odot T) \Rightarrow \Box((T \leq x + 0.5) \wedge \odot(T > x + 0.5) \Rightarrow \psi)$$

which states that whenever φ is true then ψ is true 0.5 time units later. The following examples illustrate some common specifications written in ωTL.

Example 3.3.1

1. If $T = u$ is a state transition point or a time point which is an integral number of time units after the last state transition point and φ is true at that time point then if a state transition occurs at $T = u + 1$ or it is a time point which is an integral number of time units after the last state transition point then ψ will be true at that time point.

$$\varphi \wedge (T = u) \Rightarrow \Box((T = u + 1) \Rightarrow \psi)$$

2. If φ holds at $T = x$ then ψ will be true at $T = x + 1$. (Note that the formula does not imply that a state transition occurs at $T = x$ or at $T = x + 1$).

$$\varphi \wedge (T = u) \wedge (u \leq x < \odot T) \Rightarrow \Box((T \leq x + 1) \wedge \odot(T > x + 1) \Rightarrow \psi)$$

3. If $T = u$ is a state transition point or it is a time point which is an integral number of time units after the last state transition point and φ is true at that time point then a state transition occurs at $T = u + 1$ or it is a time point which is an integral number of time units after the last state transition point and ψ holds at that time point.

$$\varphi \wedge (T = u) \Rightarrow \Diamond((T = u + 1) \wedge \psi)$$

4. If φ holds at $T = x$ then at $T = x + 1$ a state transition occurs or it is a time point which is an integral number of time units after the last state transition point and ψ holds at that time point. (Note that the formula does not imply that a state transition occurs at $T = x$ but does at $T = x + 1$).

$$\varphi \wedge (T = u) \wedge (u \leq x < \odot T) \Rightarrow \Diamond((T = x + 1) \wedge \odot(T > x + 1) \wedge \psi)$$

■

From now on, we will avoid any reference in the informal description to the fact that the formulas in ωTL are only evaluated at state transition points and at time points which are obtained by adding positive integer values to the last state transition point.

The proof theory of the propositional part of ωTL consists of the following axioms.

Axioms

F1. $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$
F2. $\vdash \odot(\varphi \Rightarrow \psi) \Rightarrow (\odot\varphi \Rightarrow \odot\psi)$
F3. $\vdash \oplus\varphi \Rightarrow \odot\varphi$
F4. $\vdash \varphi \Rightarrow \odot \ominus \varphi$
F5. $\vdash \Box\varphi \Rightarrow \Box \odot \varphi$
F6. $\vdash \Box(\varphi \Rightarrow \odot\varphi) \Rightarrow (\varphi \Rightarrow \Box\varphi)$
F7. $\vdash \varphi \mathsf{U} \psi \Leftrightarrow (\psi \vee (\varphi \wedge \odot(\varphi \mathsf{U} \psi)))$
F8. $\vdash \Box\varphi \Rightarrow \varphi \mathsf{U} \psi$

P1. $\vdash \boxdot(\varphi \Rightarrow \psi) \Rightarrow (\boxdot\varphi \Rightarrow \boxdot\psi)$
P2. $\vdash \oslash(\varphi \Rightarrow \psi) \Rightarrow (\oslash\varphi \Rightarrow \oslash\psi)$
P3. $\vdash \ominus\varphi \Rightarrow \oslash\varphi$
P4. $\vdash \varphi \Rightarrow \oslash \oplus \varphi$
P5. $\vdash \boxdot\varphi \Rightarrow \boxdot \oslash \varphi$
P6. $\vdash \boxdot(\varphi \Rightarrow \oslash\varphi) \Rightarrow (\varphi \Rightarrow \boxdot\varphi)$
P7. $\vdash \varphi \mathsf{S} \psi \Leftrightarrow (\psi \vee (\varphi \wedge \oslash(\varphi \mathsf{S} \psi)))$
P8. $\vdash \diamondsuit \oslash \mathit{false}$

Axiom F1 states that if "henceforth φ implies ψ" then whenever φ holds then so does ψ. Axiom F2 is similar to F1 for the *weak next* operator. Axiom F3 establishes that *strong next* implies *weak next*. Axiom F4 combines the future with the past: it states that if φ is true then $\odot \ominus \varphi$ also holds. Axiom F5 states if "henceforth φ" holds then so does $\Box \odot \varphi$. Axiom F6 states that if "henceforth φ implies $\odot\varphi$" holds then "φ implies $\Box\varphi$" also holds. Axiom F7 defines *weak until* recursively by stating that either ψ is true, or φ and $\odot(\varphi \mathsf{U} \psi)$ hold, or both hold. Axiom F8 states that "henceforth φ" implies that $\varphi \mathsf{U} \psi$ also holds. Axioms P1-P7 apply to the past fragment and are symmetric to F1-F7. Axiom P8 states that the past is bounded.

Inference Rules

R1. If φ is a propositional tautology then $\vdash \varphi$

R2. Modus Ponens

$$\frac{\vdash \varphi, \vdash \varphi \Rightarrow \psi}{\vdash \psi}$$

R3. $\Box$ and $\boxdot$ Introduction

$$\frac{\vdash \varphi}{\vdash \Box\varphi \wedge \boxdot\varphi}$$

Rule R1 states that if φ is a tautology in propositional calculus then it is a theorem in ωTL. Rule R2 is the Modus Ponens. Rule R3 states that if φ is a theorem then $\Box\varphi$ and $\boxdot\varphi$ are also theorems.

Next, we consider axioms and inference rules involving quantifiers over global variables and their interactions with the temporal operators. We define the existential quantifier in terms of the universal quantifier.

PL6. $\exists x.\varphi \triangleq \neg \forall x.\neg\varphi$

The axioms are as follows :

Axioms

F9. $\vdash \forall x. \odot \varphi \Rightarrow \odot \forall x.\varphi$ **P9.** $\vdash \forall x. \oslash \varphi \Rightarrow \oslash \forall x.\varphi$

Axioms F9 and P9 state that the $\forall$ quantifier for global variables commutes with the $\odot$ and $\oslash$ operators respectively.

Inference Rules

R4. If φ is a tautology in predicate calculus then $\vdash \varphi$

R5. $\forall$-Introduction for global variables

$$\frac{\vdash \varphi \Rightarrow \psi}{\vdash \varphi \Rightarrow \forall x.\psi}$$

where x is not free in φ

Rule R4 states that if φ is a tautology in predicate calculus then it is a theorem in ωTL. Rule R5 states that if $\varphi \Rightarrow \psi$ is a theorem then $\varphi \Rightarrow \forall x.\psi$ is also a theorem provided x is not free in φ.

Next, we consider axioms which govern the use of the weak next term and weak previous term operators.

Axioms

F10. $\vdash \odot F(t_1, \ldots, t_n) = F(\odot t_1, \ldots, \odot t_n)$
P10. $\vdash \oslash F(t_1, \ldots, t_n) = F(\oslash t_1, \ldots, \oslash t_n)$
F11. $\vdash \odot P(t_1, \ldots, t_n) = P(\odot t_1, \ldots, \odot t_n)$
P11. $\vdash \oslash P(t_1, \ldots, t_n) = P(\oslash t_1, \ldots, \oslash t_n)$

Axioms F10 and F11 state that to evaluate $\odot F(t_1, \ldots, t_n)$ we can evaluate $F(\odot t_1, \ldots, \odot t_n)$ and to evaluate $\odot P(t_1, \ldots, t_n)$ we can evaluate $P(\odot t_1, \ldots, \odot t_n)$. Axioms P10 and P11 are symmetric to F10 and F11 for the weak previous term operator.

We give now axioms which apply the weak next term and the weak previous term operators over the distinguished variable T.

Axioms

F12. $\vdash \odot \mathit{false} \Leftrightarrow (\odot T) = T$ P12. $\vdash \oslash \mathit{false} \Leftrightarrow T = 0$
F13. $\vdash \neg \odot \mathit{false} \Leftrightarrow (\odot T) > T$ P13. $\vdash \neg \oslash \mathit{false} \Leftrightarrow (\oslash T) < T$

Axiom F12 states that $\odot T$ is equal to T iff $\odot \mathit{false}$ holds. Axiom P12 states that $T = 0$ iff $\oslash \mathit{false}$ holds. Axiom F13 asserts that $(\odot T) > T$ iff $\neg \odot \mathit{false}$ holds and Axiom P13 states that $(\oslash T) < T$ iff $\neg \oslash \mathit{false}$ holds.

Axioms of equality are as follows :

Axioms

PL7. $\vdash t = t$ for any term t
PL8. $\vdash (t_1 = t_2) \Rightarrow (\varphi(t_1, t_1) \Leftrightarrow \varphi(t_1, t_2))$
where φ is a formula which does not contain any temporal operators, and t_1 and t_2 are terms.

Axiom PL7 states that equality is reflexive and Axiom PL8 asserts that for any two terms which are equal one can be substituted for the other in a formula which does not contain any temporal operators.

Remark 3.3.1 : The proof theory presented in this section does not contain an axiom which characterises the *next* operator. This is because such an axiom cannot be expressed in the logic. To see this, let VAR be an infinite set of local variables, then for some local variable x in this set either $x \neq \odot x$ or $(\odot T) = T + 1$ holds. This property cannot be stated in the logic because infinite disjunction is not allowed. ■

To prove the soundness of the proof system, we have to prove the following theorem.

Theorem 3.3.1 (Soundness) *If* $\vdash \varphi$ *then* $\models \varphi$

To prove the above theorem we have to prove that all the axioms are sound. Detailed proofs are given in Appendix A.1.

3.4 A Calculus of Durations and Occurrences

The logic described in the previous section is quite adequate for reasoning about many properties of a real-time system. However, because of the low-level nature of the specification written in it, the proofs are tedious and sometimes difficult to obtain. In this section, we introduce a calculus which allows requirements to be expressed more succinctly and develop a proof technique for using it to prove the safety properties of a real-time system. The calculus is defined as theorems within the logic.

Consider the following example taken from [CHR91].

Example 3.4.1 (Gas Burner) [CHR91] The example concerns the safety requirements of a gas burner. Let $leak(t)$ be a predicate which returns 1 iff *leak* holds at time t and returns 0 iff *leak* does not hold at time t. To avoid dangerous accumulation of gas, the proportion of time in the leak state must not be more than one-twentieth of the elapsed time. The interval over which the system is observed should be at least one minute long - otherwise the requirement would be violated immediately on the start of a leak. The requirement can be stated as :

Req

$$t_2 - t_1 \geq 60 \ secs \Rightarrow 20 \int_{t_1}^{t_2} leak(t)dt \leq t_2 - t_1$$

The next stage is to make certain design decisions regarding the implementation which must be shown to meet the above requirement. For example, a leak should be detectable and stoppable within one second and then it is acceptable to wait for another thirty seconds before risking another leak by switching the gas on again. These two design decisions are formalised as follows :

Des-1

$$\int_{t_1}^{t_2} leak(t)dt = t_2 - t_1 \Rightarrow t_2 - t_1 \leq 1 \ sec$$

and

Des-2

$$leak(t_1) \wedge leak(t_2) \wedge (\exists t.t_1 < t < t_2 \wedge \neg leak(t)) \Rightarrow 30 \ secs \ \leq t_2 - t_1$$

The conjunction of the two formulas implies the original requirement, a fact which should be proved before proceeding with an implementation.

■

In our attempt to prove the requirement from the design decisions, we first code the requirement and the design decisions in the logic developed in the previous section. The requirement is stated in terms of the duration of gas leaks. We therefore require additional concepts and definitions to formalise **Req**.

Definition 3.4.1 (Duration of a Predicate) The duration of a predicate is the length of time for which it is true since $T = 0$.

■

As a next step, we need to formalise the concept of a duration and develop some rules to calculate and relate durations of predicates. To do this, we need to introduce quantification over local (i.e. flexible) variables into the language and give its semantics. The formation rules for the terms and the formulae introduced in Section 3.1 remain the same, but the following rule is added.

Formation Rules

- Formulae
  1. If φ is a formula and x is a local variable, then $\forall x.\varphi$ is also a formula.

The semantics of the universal quantifier is given as :

- Universal quantification over local variables

$\forall x.\varphi|_i^\sigma = \mathbf{tt}$ iff for every $\sigma' \in \Sigma$ such that $\forall i.ts(\sigma')(i) = ts(\sigma)(i)$, $\varphi|_i^{\sigma'} = \mathbf{tt}$, where σ' is obtained from σ by assigning at all $\tau \in \Gamma$ the same values to all local variables except x

The quantifier over local variables $\forall x$ differs from the quantifier over global variables in that it asserts that any value can be substituted for x at state transition points instead of just a single value. However, it obeys the same laws as the universal quantifier over global variables.

We can define the existential quantification over local variables in terms of the universal quantifier.

PL9. $\exists x.\varphi \triangleq \neg\forall x.\neg\varphi$

The universal quantifier over local variables satisfies the following axioms and inference rules.

Axioms

F14. $\vdash \forall x.\, \odot\varphi \Rightarrow \odot\forall x.\varphi$ P14. $\vdash \forall x.\, \oslash\varphi \Rightarrow \oslash\forall x.\varphi$

Axioms F14 and P14 state that the $\forall$ quantifier for local variables commutes with the $\odot$ and $\oslash$ operators respectively.

Inference Rules

R6. $\forall$-Introduction for local variables

$$\frac{\vdash \varphi \Rightarrow \psi}{\vdash \varphi \Rightarrow \forall x.\psi}$$

where x is not free in φ

Rule R6 states that if $\varphi \Rightarrow \psi$ is a theorem then $\varphi \Rightarrow \forall x.\psi$ is also a theorem provided x is not free in φ.

Let $\mathcal{D}(\varphi, d)$ be a formula which states that d is a flexible variable which records the duration of φ. Formally, $\mathcal{D}(\varphi, d)$ is defined as follows :

$$\begin{aligned}\mathcal{D}(\varphi, d) \triangleq \Box\boxdot((T = 0 \Rightarrow d = 0)\\ \wedge(T \neq 0 \Rightarrow ((\oslash\varphi \Leftrightarrow d = (\oslash d) + T - \oslash T)\\ \wedge(\oslash\neg\varphi \Leftrightarrow d = \oslash d))))\end{aligned}$$

The first clause states that the value of d at $T = 0$ is 0. The second clause states that at all future state transition points and time points obtained by adding positive integer values to the last state transition point and where T is not equal to zero, the value of d is equal to $(\oslash d) + T - \oslash T$ iff $\oslash\varphi$ holds and its value is equal to $\oslash d$ iff $\oslash\neg\varphi$ holds.

Example 3.4.2 Let l denote that there is a gas leak. Then, a valid behaviour σ with only those time points at which a state transition occurs is illustrated as follows:

(τ_0, s_0)	(τ_1, s_1)	(τ_2, s_2)	(τ_3, s_3)	(τ_4, s_4)	$\cdots$
l	l	$\neg l$	l	$\neg l$	$\cdots$
$d = 0$	$d = \tau_1$	$d = \tau_2$	$d = \tau_2$	$d = \tau_2 + \tau_4 - \tau_3$	$\cdots$

The value of d which records the duration of gas leaks is equal to zero at $T = \tau_0 = 0$. Its value is equal to τ_1 at $T = \tau_1$ and at $T = \tau_2$ and at $T = \tau_3$ it is equal to τ_2. Its value is equal to $\tau_2 + \tau_4 - \tau_3$ at $T = \tau_4$, and so on.

■

Remark 3.4.1 : Note that the duration of a predicate could be defined inductively because we have defined the *next* operator in the logic as the next time point at which a state transition occurs if such a time point exists in the future or else it is the time point obtained by adding one to the current time.

■

The following derived rules apply over the durations of predicates.

Rule S1 The duration of $false$ is zero.

$$\vdash \mathcal{D}(false, d) \Rightarrow \Box(d = 0) \wedge \boxdot(d = 0)$$

Rule S2 The duration of *true* at any time T is T.

$$\vdash \mathcal{D}(true, d) \Rightarrow \Box(d = T) \wedge \boxdot(d = T)$$

Rule S3 The duration of any predicate is greater than or equal to zero.

$$\vdash \mathcal{D}(\varphi, d) \Rightarrow \Box(d \geq 0) \wedge \boxdot(d \geq 0)$$

Rule S4 The total duration of any two predicates is the sum of the durations of their disjunction and their conjunction.

$$\vdash \mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(\psi, d_2) \wedge \mathcal{D}(\varphi \vee \psi, d_3) \wedge \mathcal{D}(\varphi \wedge \psi, d_4) \\ \Rightarrow \Box(d_1 + d_2 = d_3 + d_4) \wedge \boxdot(d_1 + d_2 = d_3 + d_4)$$

We can now specify **Req** in terms of the duration of leak. Let l be a boolean variable which holds when there is a gas leak and d be a flexible variable which records the duration of leak. Then,

Req

$$\mathcal{D}(l, d) \Rightarrow \Box(((T = u) \wedge (d = x)) \\ \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(d - x) \leq T - u))$$

Req asserts that if the duration of leak d at $T = u$ is equal to x time units then whenever $u + 60 \leq T$ holds then $20(d - x) \leq T - u$ also holds.

The design decisions are asserted as :

Des-1

$$\Box(((T = u) \wedge l) \Rightarrow \Diamond((T \leq u + 1) \wedge \neg l))$$

Des-1 states that whenever l holds, then $\neg l$ will eventually hold within one time unit.

Des-2

$$\Box\neg((T = u) \wedge l \wedge \Diamond(\neg l \wedge \Diamond((T < u + 30) \wedge l)))$$

Des-2 states that it is always not the case that l holds at $T = u$ and eventually $\neg l$ and $\Diamond((T < u + 30) \wedge l)$ hold.

We now attempt to prove the requirement from the design decisions. To prove **Des-1** $\wedge$ **Des-2** $\Rightarrow$ **Req** we need to find an invariant which can be derived from the design decisions and can be used in proving **Req**. Note that both **Des-1** and **Des-2** are coded using the distinguished variable T whereas **Req** is described using the duration of leak. We may therefore try expressing the design decisions also in terms of the duration of leak. **Des-1** and **Des-2** can then be encoded as :

Des-1'

$$\mathcal{D}(l, d) \wedge \Box((l \wedge (d = x)) \Rightarrow \Diamond(\neg l \wedge (d \leq x + 1)))$$

Des-1' states that whenever l holds and its duration is equal to x time units then eventually $\neg l$ will hold and the duration of l will be less than or equal to $x + 1$ time units.

Des-2'

$$\mathcal{D}(\neg l, d_1) \wedge \Box((l \wedge (d_1 = z)) \Rightarrow \Box(l\mathcal{S}^+(\neg l\mathcal{S}^+(l \wedge d_1 = z)) \Rightarrow z + 30 \leq d_1))$$

Des-2' asserts that whenever l holds and the duration d_1 of $\neg l$ is equal to z time units then whenever $l\mathcal{S}^+(\neg l\mathcal{S}^+(l \wedge d_1 = z))$ holds then the duration of $\neg l$ will be greater than or equal to $z + 30$ time units.

We may now try again to prove the requirement from the design decisions. Note that **Des-2'** contains a formula which uses nested *since* operator as a result of coding it in terms of the duration of $\neg l$. This is because **Des-2'** states that if there is a gas leak and the duration of $\neg l$ at that time is equal to z time units then at the next occurrence of a gas leak the duration of $\neg l$ must be at least equal to $z + 30$ time units. Proofs using formulas which contain nested *until* or *since* operators are generally more difficult. As a next step, we code any design decisions which express bounded-response properties into safety ones. This may enable us to find an invariant. **Des-1'** is the only bounded-response property. It can be written as a safety property as follows :

Des-1"

$$\mathcal{D}(l, d) \wedge \Box((l \wedge (d = x)) \Rightarrow (d \leq x + 1)\mathsf{U}\neg l)$$

Des-1" states that whenever l holds and the duration of l is equal to x time units then the duration of l will always be less than or equal to $x + 1$ time units until $\neg l$ holds.

We can also derive the following safety property from **Des-1'**.

$$\mathcal{D}(l, d) \wedge \Box((\neg l \wedge (d = x)) \Rightarrow \Box(l\mathcal{S}^+(\neg l \wedge (d = x)) \Rightarrow d \leq x + 1))$$

It states that whenever $\neg l$ holds and the duration d of l is equal to x time units then whenever $l\mathcal{S}^+(\neg l \wedge (d = x))$ holds then the duration of leak will be less than or equal to $x + 1$ time units.

Note that this has not reduced the complexity of the formulas. Therefore, as the next step we introduce some new concepts. It is not difficult to see that **Des-1** states that the duration of an 'occurrence' of a gas leak must be less than or equal to one time unit and **Des-2** states that the duration of an 'occurrence' of $\neg l$ between any two occurrences of leak must be greater than or equal to thirty time units. To code the design decisions in terms of the number of occurrences of a gas leak we need to formalise the concept of an occurrence of a predicate.

Definition 3.4.2 (Occurrence of a Predicate) An occurrence of a predicate φ is the maximal interval of time by inclusion such that φ holds throughout the interval.

■

Example 3.4.3 Consider the behaviour described in the Example 3.4.2 where l denotes that there is a gas leak.

(τ_0, s_0)	(τ_1, s_1)	(τ_2, s_2)	(τ_3, s_3)	(τ_4, s_4)	$\cdots$
l	l	$\neg l$	l	$\neg l$	$\cdots$

The first occurrence of l is the left-closed right-open interval $[\tau_0, \tau_2)$ and the second occurrence is the interval $[\tau_3, \tau_4)$.

■

Let $\mathcal{N}(\varphi, n)$ be a formula which states that n is a flexible variable which records the number of occurrences of φ since $T = 0$. Formally, $\mathcal{N}(\varphi, n)$ is defined as follows:

$$\begin{aligned}\mathcal{N}(\varphi, n) \triangleq \Box\boxdot((T = 0 \Rightarrow (\varphi \Leftrightarrow n = 1) \wedge (\neg\varphi \Leftrightarrow n = 0)) \\ \wedge (T \neq 0 \Rightarrow (((\neg\varphi \vee (\varphi \wedge \oslash\varphi)) \Leftrightarrow n = \oslash n) \\ \wedge ((\varphi \wedge \oslash\neg\varphi) \Leftrightarrow n = (\oslash n) + 1))))\end{aligned}$$

The first clause states that the value of n at $T = 0$ is zero iff φ does not hold at $T = 0$ and it is equal to one iff φ is true at $T = 0$. The second clause states that at all future state transition points and at time points obtained by adding positive integer values to the time at the last state transition point and where T is not equal to zero, the value of n is equal to $\oslash n$ iff φ does not hold, or if φ is true and $\oslash\varphi$ also holds; and the value of n is equal $(\oslash n) + 1$ iff φ holds and $\oslash\neg\varphi$ also holds.

Example 3.4.4 Consider again the behaviour described in the Example 3.4.2 where l denotes that there is a gas leak.

(τ_0, s_0)	(τ_1, s_1)	(τ_2, s_2)	(τ_3, s_3)	(τ_4, s_4)	$\cdots$
l	l	$\neg l$	l	$\neg l$	$\cdots$
$n = 1$	$n = 1$	$n = 1$	$n = 2$	$n = 2$	$\cdots$

The value of n which records the number of occurrences of l is equal to one at $T = \tau_0 = 0$. It is also equal to one at $T = \tau_1$ and at $T = \tau_2$. Its value is equal to two at $T = \tau_3$ and $T = \tau_4$ and so on.

■

The following derived rules apply over the number of occurrences of predicates.

Rule S5 The number of occurrences of *false* is zero.

$$\vdash \mathcal{N}(false, n) \Rightarrow \Box(n = 0) \wedge \boxdot(n = 0)$$

Rule S6 The number of occurrences of any predicate is greater than or equal to zero.

$$\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box(n \geq 0) \wedge \boxdot(n \geq 0)$$

Rule S7 The difference between the number of occurrences of a predicate and its negation is either zero or one.

$$\begin{aligned} \vdash \mathcal{N}(\varphi, n_1) \wedge \mathcal{N}(\neg\varphi, n_2) \Rightarrow \\ \Box((\varphi \Rightarrow 0 \leq n_1 - n_2 \leq 1) \wedge (\neg\varphi \Rightarrow 0 \leq n_2 - n_1 \leq 1)) \\ \wedge \boxdot((\varphi \Rightarrow 0 \leq n_1 - n_2 \leq 1) \wedge (\neg\varphi \Rightarrow 0 \leq n_2 - n_1 \leq 1)) \end{aligned}$$

The following derived rules relate the duration of a predicate to the number of its occurrences.

Rule S8 If the number of occurrences of a predicate is zero then its duration is also zero.

$$\vdash \mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \Rightarrow \Box((n = 0) \Rightarrow (d = 0)) \wedge \boxdot((n = 0) \Rightarrow (d = 0))$$

Rule S9 The number of occurrences of a predicate is zero iff it was not true at any time in the past.

$$\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box((n = 0) \Leftrightarrow \neg\diamondsuit\varphi) \wedge \boxdot((n = 0) \Leftrightarrow \neg\diamondsuit\varphi)$$

Rule S10 If a predicate was always true in the past then the number of its occurrences is one.

$$\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box(\boxdot\varphi \Rightarrow (n = 1)) \wedge \boxdot(\boxdot\varphi \Rightarrow (n = 1))$$

Rule S11 If the maximum duration of any occurrence of φ is equal to k time units then whenever an occurrence of φ occurs then the duration of φ increases by at most k time units.

$$\begin{aligned}\vdash \mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \Rightarrow \\ (\Box((\varphi \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg\varphi \wedge (n = y) \wedge d \leq x + k)) \Rightarrow \\ \Box((\neg\varphi \wedge (n = y_1) \wedge (d = x_1)) \Rightarrow \Box((n = y_1 + 1) \Rightarrow d \leq x_1 + k)))\end{aligned}$$

Rule S11 allows us to derive a safety property from a liveness one. The proofs of the derived rules of the calculus are given in Appendix B.

Remark 3.4.2 : The induction rules of a logic provide a powerful tool for proving safety properties. Using the calculus, we can do induction on the number of occurrences of a predicate. That is, to prove that a property involving a predicate φ always holds, we prove it holds when the number of its occurrences is zero, assume that it holds when the number of occurrences is n and prove it for $n + 1$. The induction principle is stated as follows:

$$\vdash (R(0) \wedge \forall n.(R(n) \Rightarrow R(n + 1))) \Rightarrow R(k)$$

for any formula R.

■

We can now code the requirement and the design decisions of Example 3.4.1 in terms of the number of occurrences of l.

Req

$$\begin{aligned}\mathcal{D}(l, d) \Rightarrow \Box(((T = u) \wedge (d = x)) \\ \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(d - x) \leq T - u))\end{aligned}$$

The specification of the requirement remains the same.

Des-1

$$\begin{aligned}&\mathcal{N}(l, n) \wedge \mathcal{D}(l, d) \\ &\wedge\Box((l \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg l \wedge (n = y) \wedge (d \leq x + 1)))\end{aligned}$$

Des-1 asserts that whenever l holds and the number of its occurrences is equal to y and its duration is equal to x time units then eventually $\neg l$ will hold and the number of occurrences of l will be equal to y and its duration will be less than or equal to $x + 1$ time units.

Des-2

$$\mathcal{N}(l, n) \wedge \mathcal{D}(\neg l, d_1)$$
$$\wedge \Box((l \wedge (n = y) \wedge (d_1 = z)) \Rightarrow \Box((y < n) \Rightarrow (z + 30 \leq d_1)))$$

Des-2 asserts that whenever l holds and the number of its occurrences is equal to y and the duration of $\neg l$ is equal to z time units then whenever the number of occurrences of l is greater than y then the duration of $\neg l$ will be greater than or equal to $z + 30$ time units.

Note that by using the number of occurrences of l to code the design decisions we have not introduced any nested *since* or *until* operators.

We must prove **Des-1** $\wedge$ **Des-2** $\Rightarrow$ **Req**. But first, we define some abbreviations to make the proofs a little more presentable.

$$p \triangleq ((l \wedge (n = y) \wedge (d_1 = z)) \Rightarrow \Box((y < n) \Rightarrow (z + 30 \leq d_1)))$$
$$\wedge((l \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg l \wedge (n = y) \wedge (d \leq x + 1)))$$
$$q \triangleq ((T = u) \wedge (d = x)) \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(d - x) \leq T - u)$$
$$r \triangleq \mathcal{N}(l, n) \wedge \mathcal{D}(l, d) \wedge \mathcal{D}(\neg l, d_1)$$

Using the abbreviations we have

$$\textbf{Req} \triangleq \mathcal{D}(l, d) \Rightarrow \Box q$$

$$\textbf{Des-1} \wedge \textbf{Des-2} \triangleq r \wedge \Box p$$

Having coded the design decisions in terms of the number of occurrences of l we need to find an invariant. It is not difficult to see that since the duration of each leak is less than or equal to one time unit the duration of l will always be less than or equal to the number of occurrences of l. Also, since after a leak has occurred another leak cannot occur for at least thirty time units, it implies that if the number of occurrences of l is more than one then the duration of $\neg l$ must be at least thirty times the number of occurrences of l minus one. The proof therefore rests on the following two lemmas.

Lemma-1 : The duration d of l is always less than or equal to the number of its occurrences times the maximum duration of each l.

$$\vdash (r \wedge \Box p) \Rightarrow \Box(d \leq n)$$

Lemma-2 : If there is more than one leak, then the duration of $\neg l$ is greater than or equal to thirty times the number of occurrences of l minus one.

$$\vdash (r \wedge \Box p) \Rightarrow \Box((2 \leq n) \Rightarrow 30(n-1) \leq d_1)$$

The proofs of lemmas make use of the rules S8 and S11 of the calculus. Rule S8 states that if the number of occurrences of a predicate is zero then its duration is also zero.

$$\vdash \mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \Rightarrow \Box(n = 0 \Rightarrow d = 0)$$

Rule S11 states that if the maximum duration of each occurrence of φ is equal to k time units then whenever an occurrence of φ occurs then the duration of φ increases by at most k time units.

$$\begin{aligned} \vdash \mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \Rightarrow \\ (\Box((\varphi \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg\varphi \wedge (n = y) \wedge d \leq x + k)) \Rightarrow \\ \Box((\neg\varphi \wedge (n = y_1) \wedge (d = x_1)) \Rightarrow \Box((n = y_1 + 1) \Rightarrow d \leq x_1 + k))) \end{aligned}$$

The proof of **Lemma-1** is as follows :

Proof :

1. $\vdash (r \wedge \Box p) \Rightarrow \Box(d \leq n)$

To prove (1) we use induction on the number of occurrences of l, i.e. n. For $n = 0$, the result follows from S8. Assume for $n = y$ that $d \leq n$. We prove for $n = y + 1$ that $d \leq y + 1$. (2) follows from $r \wedge \Box p$.

2. $\Box((l \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg l \wedge (n = y) \wedge (d \leq x + 1)))$

(3) follows from (2) and S11.

3. $\Box(\neg l \wedge (n = y) \wedge (d = x) \Rightarrow \Box((n = y + 1) \Rightarrow (d \leq x + 1)))$

(4) follows from (3), the induction assumption, and arithmetic.

4. $\Box((n = y + 1) \Rightarrow (d \leq y + 1))$

This proves the induction step and therefore the lemma.

■

The proof of **Lemma-2** is as follows :

Proof :

1. $\vdash (r \wedge \Box p) \Rightarrow \Box((2 \leq n) \Rightarrow 30(n - 1) \leq d_1)$

To prove (1) we use induction on n. For the base case $n = 2$, $30 \leq d_1$ follows trivially from $r \wedge \Box p$. Assume for $(n = y) \wedge (2 \leq n)$ that $30(n - 1) \leq d_1$. We prove for $(n = y + 1) \wedge (2 \leq n)$ that $30y \leq d_1$. (2) follows from $r \wedge \Box p$.

2. $\Box((l \wedge (n = y) \wedge (d_1 = z)) \Rightarrow \Box((y < n) \Rightarrow (z + 30 \leq d_1)))$

(3) follows from (2), the induction assumption, and arithmetic.

3. $\Box((y < n) \Rightarrow (30y \leq d_1))$

This proves the induction step and therefore the lemma.

■

The proof of the main theorem $\vdash (r \wedge \Box p) \Rightarrow (\mathcal{D}(l, d) \Rightarrow \Box q)$ makes use of the rules S2, S3 and S4 of the calculus. Rule S2 states that the duration of *true* at any time T is T.

$$\vdash \mathcal{D}(true, d) \Rightarrow \Box(d = T)$$

Rule S3 states that the duration of any predicate is greater than or equal to zero.

$$\vdash \mathcal{D}(\varphi, d) \Rightarrow \Box(d \geq 0)$$

Rule S4 states that the total duration of any two predicates is the sum of the durations of their disjunction and their conjunction.

$$\vdash \mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(\psi, d_2) \wedge \mathcal{D}(\varphi \vee \psi, d_3) \wedge \mathcal{D}(\varphi \wedge \psi, d_4) \Rightarrow \Box(d_1 + d_2 = d_3 + d_4)$$

The proof is as follows :

Proof :

1. $\vdash (r \wedge \Box p) \Rightarrow (\mathcal{D}(l, d) \Rightarrow \Box q)$

To prove (1), we prove (2), and apply lemma-1 and lemma-2.

2. $(r \wedge \Box((d \leq n) \wedge (2 \leq n \Rightarrow 30(n-1) \leq d_1))) \wedge \mathcal{D}(l, d) \Rightarrow \Box((T = u) \wedge (d = x) \Rightarrow \Box(u + 60 \leq T \Rightarrow 20(d - x) \leq T - u))$

Assume (3).

3. $r \wedge \Box((d \leq n) \wedge ((2 \leq n) \Rightarrow 30(n-1) \leq d_1))$

(4) follows from (3), S2, S4 and arithmetic.

4. $r \wedge \Box((2 \leq n) \Rightarrow (d + 30n - 30 \leq T))$

To prove (2), we prove (5) by assuming (3).

5. $\Box((T = u) \wedge (d = x) \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(d - x) \leq T - u))$

To prove (5), we prove (6) which follows from (3) and arithmetic.

6. $\Box((T = u) \wedge (d = x) \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(n - x) \leq T - u))$

To prove (6), we use induction on n. For $n \leq 2$, (6) holds because of (3) and S3. For $2 \leq n$, we prove (7) which follows from (4) and (6).

7. $\Box((T = u) \wedge (d = x) \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(n - x) \leq d + 30n - 30 + c - u))$

by assuming it holds for $n = y$ and $T = d + 30n - 30 + c$ $(c \geq 0)$ and proving that it holds for $n = y + 1$. The proof is trivial. This proves the theorem. ■

We now summarise the method we followed in proving the gas burner problem.

1. Code the requirements and the design in terms of the formulas in the 'raw' logic. If it is not possible, then develop some new concepts. In the gas burner problem, for example, we defined the concept of a duration of a predicate to express the requirement.
2. Try proving the requirement from the design decisions. The proofs of safety properties usually require finding an invariant.
3. If the design is not expressed in the same concepts as the requirement then code the design decisions such that they are expressed in the same concepts. In the gas burner problem, we coded the design decisions in terms of the duration of the gas leaks. This may result in the use of nested *until* and *since* operators. The proofs in such cases may not be easy.
4. If the desired property cannot be proved from Step 3, try coding any design decisions which are bounded-response properties into safety ones. Try finding an invariant again.
5. If after Step 4, an invariant cannot be found, then develop some new concepts. In the gas burner problem, we developed a concept of an occurrence of a predicate and used the number of gas leaks to code the design decisions. The invariant in the gas burner problem is easily deduced from the design decisions when they are expressed in terms of the number of gas leaks.
6. Prove the invariant from the design decisions. In the gas burner problem, we used induction on the number of gas leaks. In proving the invariant, we also used a rule which allows us to derive a safety property from a liveness one.
7. Prove the desired property from the invariant. In the gas burner problem, this was again done by induction on the number of gas leaks.

3.5 Discussion

In this chapter, we introduced a temporal framework for reasoning about the requirements and the design of a real-time system. It is based on a temporal

logic in which the *next* operator is defined as the next time a state transition occurs if such a time point exists or else it is the time point obtained by adding one to the current time. This allowed us to abstract away from any particular time domain; we can use either a continuous or a discrete model of time depending on the application area. The resulting logic is therefore more expressive than a logic which allows only a discrete time model or which only allows a continuous model. We also developed a calculus of durations and the number of occurrences of predicates and used it to prove timing properties of a gas burner.

In developing the logic we made some detailed design decisions. In particular, the function ts was defined such that it takes as its arguments a valid behaviour and returns an ordered sequence of time points at which a state transition occurs or after which no state transitions occur and is equal to the previous time point in the sequence incremented by one. We also defined $|ts(\sigma)|$ to be the length of the sequence returned by $ts(\sigma)$. These definitions gave rise to a logic which is a 'next-change' logic if there exists a future state transition point and is a 'next-time' logic if no state transitions occur in the future. As an alternative, we can define $|ts(\sigma)|$ as follows :

$$\begin{aligned}|ts(\sigma)| &= 1 \text{ if } \sigma(ts(\sigma)(0)) = \sigma(ts(\sigma)(1)) \\ &= k \text{ if } (1 < k) \wedge \sigma(ts(\sigma)(k-1)) \neq \sigma(ts(\sigma)(k-2)) \\ &\qquad \wedge \forall j \geq k.(\sigma(ts(\sigma)(j)) = \sigma(ts(\sigma)(k-1))) \\ &= \omega \text{ if } \forall i.\, \exists j > i.\sigma(ts(\sigma)(i)) \neq \sigma(ts(\sigma)(j))\end{aligned}$$

The term $|ts(\sigma)|$ is equal to the number of state transitions occurring in a valid behaviour σ. The semantics of the operators remain the same except that i in the model $M = (I, \sigma, \alpha, i)$ refers to the i^{th} state transition point in the behaviour σ and its value is less than the number of state transitions in σ. The proof theory also remains the same except that the interpretation given to the temporal operators is now different. For example, $\Box\varphi$ (henceforth) means that φ holds at all future state transition points and $\Diamond\varphi$ (eventually) means that φ holds at some future state transition point. A logic with $|ts(\sigma)|$ defined as above leads to a 'next-change' logic which is insensitive to any stuttering. To formalise certain requirements precisely in such a logic may require additional formulae. For example, it is not sufficient to formalise the requirement of the gas burner as follows :

Req

$$\begin{aligned}\mathcal{D}(l,d) \Rightarrow \Box(((T = u) \wedge (d = x)) \\ \Rightarrow \Box((u + 60 \leq T) \Rightarrow 20(d - x) \leq T - u))\end{aligned}$$

because it can be satisfied by a gas burner with a permanent leak and if no state change ever occurs. This is because **Req** in a 'next-change' logic means

$$\begin{aligned}\mathcal{D}(l,d) \Rightarrow &\Box(((T=u) \wedge (d=x) \wedge \text{ "state changes" }) \\ &\Rightarrow \Box((u+60 \leq T) \wedge \text{ "state changes" } \Rightarrow 20(d-x) \leq T-u))\end{aligned}$$

If no state change occurs after $T = u$ then **Req** holds trivially. We therefore need to add the following liveness condition to state the requirement precisely.

$$\Box(l \Rightarrow \Diamond\neg l)$$

It asserts that if at any state transition point l holds then eventually there exists a future state transition point where $\neg l$ will hold. Another example of the difference between ωTL and a 'next-change' logic is the following. In ωTL, a property that whenever φ holds then ψ holds one time unit later is expressed as follows:

$$\varphi \wedge (T=u) \wedge (u \leq x < \odot T) \Rightarrow \Box((T \leq x+1) \wedge \odot(T > x+1) \Rightarrow \psi)$$

In a 'next-change' logic, the same property is expressed as:

$$\begin{aligned}\varphi \wedge (T=u) \wedge &((u \leq x < \odot T) \vee (u=x)) \\ &\Rightarrow \Box((T \leq x+1) \wedge \odot(T > x+1) \Rightarrow \psi)\end{aligned}$$

The difference between the two is that $u \leq x < \odot T$ is replaced by $(u \leq x < \odot T) \vee (u = x)$. This is because $\odot T$ at the last state transition point evaluates to T which is equal to u and $u \leq x < \odot T$ evaluates to an empty set. The formula with only $u \leq x < \odot T$ therefore holds trivially at the last state transition point irrespective of whether ψ holds one time unit later or not.

Yet, another alternative is to define the function ts as follows :

$$\begin{aligned}ts(\sigma)(i) &= 0 \text{ if } i = 0 \\ &= \tau' \text{ if } 0 < i \wedge ts(\sigma)(i-1) < \tau' \leq (ts(\sigma)(i-1)+1) \\ &\qquad \wedge \sigma(\tau') \neq \sigma(ts(\sigma)(i-1)) \\ &\qquad \wedge \forall \tau.(ts(\sigma)(i-1) < \tau < \tau' \Rightarrow \sigma(\tau) = \sigma(ts(\sigma)(i-1))) \\ &= ts(\sigma)(i-1)+1 \text{ otherwise}\end{aligned}$$

The function ts defined as above returns an ordered sequence of time points in which the next time point in the sequence is the time point at the next state transition if it occurs within one time unit or else it is the time point obtained by adding one to the current time. $|ts(\sigma)|$ can be defined as the length of the sequence returned by $ts(\sigma)$. The semantics and the proof theory remain the same except that the interpretation given to the temporal operators is different. For example, $\Box\varphi$ (henceforth) means that φ holds at all future state transition points and at time points obtained by adding one to the time points where state transitions occur if the next state transition occurs after one time unit or after which no state transitions occur. $\Diamond\varphi$ (eventually) means that φ holds at some future state transition point or at a time point obtained by adding one to the time point where a state transition occurs if the next state transition occurs after one time unit, or after which no state transitions occur. The logic defined in this way is sensitive to stuttering which occurs for a finite and an infinite time intervals. If, however $|ts(\sigma)|$ is defined so that it returns the length of the subsequence of $ts(\sigma)$ in which the last element is the time point at which the last state transition occurred then the logic which is obtained is sensitive to stuttering which occurs for a finite time interval but is insensitive to stuttering which occurs for an infinite time interval.

Bounded temporal operators can be introduced as abbreviations in ωTL. For example, we can define $\Diamond_{=k}\varphi$ as :

$$\Diamond_{=k}\varphi \triangleq (T = u) \Rightarrow \Diamond((T \leq u + k) \wedge \odot(T > u + k) \wedge \varphi)$$

and $\Box_{=k}\varphi$ is defined as :

$$\Box_{=k}\varphi \triangleq (T = u) \Rightarrow \Box((T \leq u + k) \wedge \odot(T > u + k) \Rightarrow \varphi)$$

Other commonly used abbreviations can be defined as follows: $\Diamond_{\leq k}\varphi$ for $(T = u) \Rightarrow \Diamond((T \leq u + k) \wedge \varphi)$, $\Diamond_{\geq k}\varphi$ for $(T = u) \Rightarrow \Diamond(\odot(T > u + k) \wedge \varphi)$, $\Box_{\leq k}\varphi$ for $(T = u) \Rightarrow \Box((T \leq u + k) \Rightarrow \varphi)$ and $\Box_{\geq k}\varphi$ for $(T = u) \Rightarrow \Box(\odot(T > u + k) \Rightarrow \varphi)$. It is also useful to be able to write bounded temporal operators in terms of the number of occurrences of some predicate where an event is used as a timer instead of the distinguished variable T. For example, we can define $\Diamond^{\varphi}_{=k}\psi$ as an abbreviation for $\mathcal{N}(\varphi, n) \wedge (n = u) \Rightarrow \Diamond((n = u + k) \wedge \psi)$. This formula asserts that ψ holds at k occurrences of φ from the present. Similarly, $\Box^{\varphi}_{=k}\psi$ can be defined as $\mathcal{N}(\varphi, n) \wedge (n = u) \Rightarrow \Box((n = u + k) \Rightarrow \psi)$.

Other abbreviations which are useful in the specification of systems involve relating occurrences of predicates. Recall that an occurrence of a predicate

was defined as a maximal interval in which it is true throughout. We can use this definition and relations between intervals to define occurrence schemas. For example, we can define $\varphi \Delta \psi$ (during) to mean that an occurrence of φ occurs during an occurrence of ψ, i.e. the maximal interval in which φ holds throughout is contained in the maximal interval in which ψ holds throughout. The *during* operator satisfies the properties of intervals. For example, it is both reflexive and transitive, i.e. $\varphi \Rightarrow \varphi \Delta \varphi$ and $\varphi \Delta \psi \wedge \psi \Delta \chi \Rightarrow \varphi \Delta \chi$ are theorems. Other useful operators such as precedes, overlaps can also be defined. An example of the style of specification using these operators and their formal definitions is given in [Nai92].

In Section 3.4, we introduced a calculus of durations and the number of occurrences of predicates and developed a proof technique for proving timing properties of a real-time system. The calculus was defined by extending the logic to include quantification over local variables. An alternative method of defining the calculus is to introduce higher-order functions. For example, we can define a function which takes as its argument a predicate and returns the length of time it is true since $T = 0$. Its semantics is given as follows :

- Duration of a predicate

$$\mathcal{D}(\varphi)|_i^\sigma = \begin{cases} 0 & \text{if } i = 0 \\ \mathcal{D}(\varphi)|_{i-1}^\sigma + T|_i^\sigma - T|_{i-1}^\sigma & \text{if } i \neq 0 \wedge \varphi|_{i-1}^\sigma \\ \mathcal{D}(\varphi)|_{i-1}^\sigma & \text{if } i \neq 0 \wedge \neg\varphi|_{i-1}^\sigma \end{cases}$$

The following axiom holds for the duration of a predicate.

$$\begin{aligned} \vdash & (\oslash \mathit{false} \Rightarrow \mathcal{D}(\varphi) = 0) \wedge (\ominus\varphi \Leftrightarrow \mathcal{D}(\varphi) = (\oslash\mathcal{D}(\varphi)) + T - \oslash T) \\ & \wedge (\ominus\neg\varphi \Leftrightarrow \mathcal{D}(\varphi) = \oslash\mathcal{D}(\varphi)) \end{aligned}$$

The soundness of the axiom follows from the semantic definition. The derived rules which apply over the durations of predicates can easily be deduced. For example, the duration of *false* is zero

$$\vdash \mathcal{D}(\mathit{false}) = 0$$

and the duration of any predicate is greater than or equal to zero

$$\vdash \mathcal{D}(\varphi) \geq 0$$

and so on. A higher-order function $\mathcal{N}(\varphi)$ which returns the number of occurrences of a predicate can be defined similarly and rules given for it.

The gas burner problem was originally solved using a calculus of durations in [CHR91]. The concept of a duration of a predicate in the Duration Calculus and in ωTL differ in two respects. Firstly, in the Duration Calculus the duration of a state φ in a closed interval $[b, e]$ is denoted by $\int\varphi$ and is defined by an integral $\int_b^e \varphi(t)dt$. A state is defined using boolean variables which are mapped to either 1 or 0 where 1 denotes that the system is in the state and 0 denotes that the system is not in the state. In ωTL, a duration of a predicate is defined as the length of time it is true since $T = 0$. Its formal definition is given inductively using the *next* operator. Secondly, the duration of a predicate in the Duration Calculus is only defined for a state and not for any arbitrary temporal formula. Therefore, $\int\Diamond\varphi$ is not a valid term in it. In ωTL, a duration is defined for any temporal formula, therefore it is valid to write $\mathcal{D}(\Diamond\varphi, d)$ to mean that d is a flexible variable which records the duration of $\Diamond\varphi$.

Another difference between the Duration Calculus and ωTL is that the Duration Calculus is a first-order logic whereas ωTL is not. In the Duration Calculus, a notation is introduced which converts a state φ to a duration predicate $\lceil\varphi\rceil$. $\lceil\varphi\rceil$ is defined as :

$$\lceil\varphi\rceil \triangleq (\int\varphi = \int 1) \wedge \int 1 > 0$$

A formula in the Duration Calculus is either a duration predicate or one formed using the usual propositional connectives ($\neg$, $\vee$, $\Rightarrow$, $\wedge$, $\Leftrightarrow$), the first-order universal quantifier ($\forall$), and the temporal operators chop ($\hat{}$), always ($\Box$), and eventually ($\Diamond$). Therefore, it is not valid to write a second-order formula such as $\Box(l \Rightarrow \int 1 \leq 1)$ or $\Box(l \Rightarrow \int true \leq 1)$ to state that an interval in which l holds throughout must be less than or equal to one time unit. The correct formula is $\Box(\lceil l\rceil \Rightarrow \int 1 \leq 1)$. In ωTL, the duration of a predicate is defined using quantification over local variables. Therefore, it is not a first-order logic.

The requirement and the design decisions of the gas burner in the Duration Calculus are asserted as :

Req

$$\int 1 \geq 60 \Rightarrow 20 \int l \leq \int 1$$

The informal meaning of the requirement is that in any interval greater than or equal to sixty time units the duration of l is less than or equal to one-twentieth of the duration of the interval.

Des-1

$$\Box(\lceil l \rceil \Rightarrow \int 1 \leq 1)$$

Des-1 states that any interval in which l holds throughout must be less than or equal to one time unit.

Des-2

$$\Box((\Diamond\lceil l \rceil)^\frown(\Diamond\lceil \neg l \rceil)^\frown(\Diamond\lceil l \rceil) \Rightarrow \int 1 \geq 30)$$

Des-2 states that any interval which can be subdivided into three subintervals such that l holds sometime in the first and in the third and $\neg l$ holds sometime in the second then its duration must be greater than or equal to thirty time units.

The proof in the Duration Calculus that the design decisions meet the requirement depends on the following induction rule over structure of a formula

$$\frac{\vdash R(\lceil\ \ \rceil), R(X) \vdash R(X \vee X^\frown\lceil\varphi\rceil \vee X^\frown\lceil\neg\varphi\rceil)}{R(true)}$$

where X is a formula letter and φ is a state. It may be possible to introduce such an induction rule in ωTL. However, the proofs using such an induction rule in ωTL will in general be more difficult than using induction over the number of occurrences of a predicate since one has to find a predicate containing a formula letter to do induction over its structure.

The concept of a duration and the number of occurrences of a predicate as defined in Section 3.4 can be introduced in an explicit clock temporal logic with discrete time such as RTTL and XCTL by extending them with quantification over local variables. However, unlike ωTL duration can only be measured with finite precision in these logics since a discrete time domain is used. In an explicit-clock or a bounded-operator temporal logic where a continuous time domain is used and which does not have a *next* operator, the duration and the number of occurrences of a predicate have to be introduced semantically as higher-order functions. Axioms which state that the duration of *false* is zero and the duration of any predicate is greater than or equal to zero have to be introduced and these have to be proved sound with respect to the semantics. The axioms in general will be complicated because the interaction between

the durations and the number of occurrences of predicates with the bounded temporal operators have to be axiomatised.

The other important issue apart from just being able to specify durations and the number of occurrences of predicates is how do we do proofs. The proofs of safety properties usually require some induction principle. In a temporal logic where discrete time is used, induction can be done over time points. In a temporal logic where a continuous time domain is used, induction can be done over structure of a formula as in the Duration Calculus. In ωTL, we introduced the concept of an occurrence and the number of occurrences of a predicate to do induction over it. An alternative induction principle in ωTL involves doing induction over the number of state transitions and the time points obtained by adding positive integer values to the last state transition point. Its use is illustrated in Chapter 5.

Chapter 4

Example : A Fault-Tolerant Broadcast Protocol

In Chapter 3, we introduced a temporal framework for specifying and proving properties of a real-time system. We now turn our attention to the problem of verification: showing that an implementation meets its specification. The specification and the implementation languages may be the same, in which case the proof techniques developed for the logic can be used. If, however the specification and the implementation languages are not the same then a proof method is developed. In this chapter, we specify and prove correctness of a fault-tolerant broadcast algorithm using the calculus developed in Chapter 3. We consider as an example the algorithm developed by Chang and Maxemchuk [CM84] which guarantees delivery of broadcast messages inspite of failures that may cause messages to be lost. The protocol also guarantees that messages are received in the same order at all sites. The algorithm is described in the following section.

4.1 The Specification Statement

Consider a broadcast network which connects several sites. Messages in such a network are available to all operational receiver sites; however because of transmission or buffer errors some or all receivers may lose a message. We may therefore want to implement a protocol which guarantees the delivery of messages in the presence of such failures. An obvious way to implement a reliable protocol is to retransmit the broadcast message until an acknowledgement is received from each of the receiver sites. This is known as a positive acknowledgement protocol. The disadvantage of such a protocol is that the

number of acknowledgements transmitted is at least as large as the number of receiver sites. It also does not guarantee that the messages are received in the same sequence at all the destination sites. In a system with a single destination and many sources the sequencing of messages is trivial. There is also only one acknowledgement. When there are many destinations and a single source, sequencing is enforced by assigning a sequence number to each message. Explicit acknowledgement is therefore not required since lost messages are detected if a message with a higher sequence is received than expected. Retransmission can then requested.

The basis of the Chang and Maxemchuk algorithm is to make a multiple source and destination network appear as a combination of two systems: one with a single destination and the other with a single source. A system with many destinations is made to look like a single destination by having only one of the receivers send the source an acknowledgement. The acknowledgement (containing a sequence number called the timestamp) is also sent to the remaining destinations, as in a system with a single source and many destinations. The remaining destinations use the timestamp to detect a missing message; they then request the missing acknowledgement and the message. They do not transmit any messages if no messages are lost.

The destinations are arranged in a cycle. After each broadcast message the next destination in the cycle takes its turn as the site responsible for acknowledging a message. A site can only perform this role if it has received all the messages that have previously been acknowledged. Therefore, all the messages that have been acknowledged since the last time the site had the responsibility of acknowledging have been received by all operational destinations and will no longer need to be retransmitted.

4.2 Abstract Specification

In order to prove that the algorithm works we develop an abstract specification and prove that it is safe. The safety property we prove asserts that if the broadcast has been completed then all operational destinations must have received the message. In the next section, we give an implementation of the algorithm in terms of queues and sequences. We show that the implementation satisfies the abstract specification and therefore is safe.

Let I be a set of site identifiers, $S \subset I$ a set of source identifiers and $D \subset I$ a set of destination identifiers.

I : a set of site identifiers
$S \subset I$: a set of source identifiers
$D \subset I$: a set of destination identifiers

Let M be a set of messages. Each message has a type associated with it. We assume for now, that there are only two types of messages $BCAST$ and ACK.

M : a set of messages
$Types = \{BCAST, ACK\}$: types of messages

In addition, a message of the $BCAST$ type has a sequence number and the source identifier. An ACK message has the sequence number of the message being acknowledged, the identifier of the source of the message and a timestamp. We define the following functions which return various fields of a message. *type* is a function which returns the type of a message, *src* returns its source and *seq* returns its sequence number.

$type : M \rightarrow Types$: message type
$src : M \rightarrow S$: message source
$seq : M \rightarrow \mathbb{N}$: message sequence number

Other functions of interest are : *ts* which returns the timestamp of a message, *tokensite* which maps timestamps to the destination responsible for sending the message with the timestamp and *responsible* which returns the site responsible for sending a message.

$ts : M \rightarrow \mathbb{N}$: message timestamp
$tokensite : \mathbb{N} \rightarrow D$: timestamp responsibility
$responsible : M \rightarrow D$: site responsible for a message

The functions *responsible*, *tokensite* and *ts* are related as follows. If $ts(m)$ is the timestamp of an acknowledgement message m then $tokensite(ts(m))$ is the site responsible for sending m and therefore is equal to $responsible(m)$.

$$tokensite(ts(m)) = responsible(m)$$

The safety property we are interested in proving is stated as follows.

A message b has been broadcast successfully (i.e. all the destination sites have received b) if the following three conditions hold.

1. The source of b has sent b.
2. The source of b has received an acknowledgement a of the receipt of b by the tokensite.
3. The source has received an acknowledgement a' for another message (possibly sent by another source) such that the timestamp of a' is equal to or greater than $|D|$ plus the timestamp a.

The notation $|D|$ denotes the cardinality of the set D.

We formalise the safety property as follows. Let the predicate $acks(b, a)$ hold iff a is an acknowledgement of b.

$$acks(b, a) \triangleq src(a) = src(b) \wedge seq(a) = seq(b)$$

Let $send(s, m)$ and $rec(s, m)$ be predicates such that $send(s, m)$ holds iff a site s sends a message m and $rec(s, m)$ holds iff a site s receives a message m.

$send(s, m)$: a site s sends a message m
$rec(s, m)$: a site s receives a message m

We further define predicates $sent(s, m)$ and $recvd(s, m)$ to indicate that a message was previously sent and received respectively. $sent(s, m)$ and $recvd(s, m)$ are defined in terms of $send(s, m)$ and $rec(s, m)$.

$$sent(s, m) \triangleq \neg send(s, m) \wedge \Diamond send(s, m)$$

$$recvd(s, m) \triangleq \neg rec(s, m) \wedge \Diamond rec(s, m)$$

From now on, as abbreviations we will use a, a', a'' to denote messages of the type ACK and b, b', b'' for messages of the type $BCAST$. Therefore, $send(s, a)$ and $rec(s, a)$ are defined as :

$$send(s, b) \triangleq \exists m.(send(s, m) \wedge type(m) = BCAST \wedge m = b)$$

$$rec(s, a) \triangleq \exists m.(rec(s, m) \wedge type(m) = ACK \wedge m = a)$$

We can now formally state the safety property.

Theorem 4.2.1 (Safety of the Network)

$$\vdash (sent(s,b) \wedge (src(b) = s) \wedge \exists a, a'.(acks(b,a) \wedge recvd(s,a) \wedge recvd(s,a') \wedge (ts(a) + |D| \leq ts(a')))) \Rightarrow \forall s'.(s' \in D \Rightarrow recvd(s',b))$$

The algorithm is specified in terms of seven properties. Our goal is to prove that the safety theorem can be deduced from these properties.

The properties are either global to the network or local to a site. The first property is the only global one; it is stated in terms of the number of occurrences of send and receive operations. Local properties on the other hand hold before and after sending and receiving of messages.

Let $\mathcal{N}(\varphi, n)$ be a formula which states that n is a flexible variable which records the number of occurrences of φ. It was defined in Section 3.4 as follows :

$$\mathcal{N}(\varphi,n) \triangleq \Box\boxdot((T = 0 \Rightarrow (\varphi \Leftrightarrow n = 1) \wedge (\neg\varphi \Leftrightarrow n = 0)) \wedge (T \neq 0 \Rightarrow (((\neg\varphi \vee (\varphi \wedge \circleddash\varphi)) \Leftrightarrow n = \circleddash n) \wedge ((\varphi \wedge \circleddash\neg\varphi) \Leftrightarrow n = (\circleddash n) + 1))))$$

Define $p \triangleq \exists s, m.send(s,m)$ and $q \triangleq \exists s, m.rec(s,m)$. Then, the properties of the network are stated as follows:

1. *Network Safety*

A. A message cannot be received when none has been sent.

$$\mathcal{N}(p,n_1) \wedge \mathcal{N}(\neg p,n_2) \wedge \mathcal{N}(q,n_3) \Rightarrow \Box(((n_1 = 0) \vee (n_2 = 0)) \Rightarrow (n_3 = 0))$$

B. A message cannot be received unless it has been sent.

$$\mathcal{N}(\neg p,n_2) \Rightarrow \Box((n_2 = u + 1) \wedge (rec(s,m) \vee recvd(s,m)) \Rightarrow \boxdot((n_2 = u) \Rightarrow \exists s'.(send(s',m) \vee sent(s',m))))$$

C. A message which has been received cannot be 'un-received'.

$$\mathcal{N}(\neg q,n_4) \Rightarrow \Box((n_4 = u) \wedge (rec(s,m) \vee recvd(s,m)) \Rightarrow \Box((n_4 = u + 1) \Rightarrow recvd(s,m)))$$

D. A message which has been sent cannot be 'un-sent'.

$$\mathcal{N}(\neg p, n_2) \Rightarrow \Box((n_2 = u) \wedge (send(s,m) \vee sent(s,m)) \\ \Rightarrow \Box((n_2 = u+1) \Rightarrow sent(s,m)))$$

2. *Tokensite Coverage* : Any sequence of $|D|$ timestamps should map to the entire set of destinations.

$$\Box(((send(s,a) \vee sent(s,a)) \wedge (|D| \leq ts(a)+1)) \\ \Rightarrow \{tokensite(t') : ts(a)+1-|D| \leq t' \leq ts(a)\} = D)$$

3. *Source Safety* : A broadcast message can only be sent by its source or by a site which has previously received it. The second condition allows rebroadcast.

$$\Box((send(s,b) \vee sent(s,b)) \Rightarrow ((src(b) = s) \vee recvd(s,b)))$$

4. *Sequence Uniqueness* : A source cannot send two distinct broadcast messages with the same sequence number.

$$\Box(((send(s,b) \vee sent(s,b)) \wedge (send(s,b') \vee sent(s,b')) \\ \wedge (s = src(b) = src(b')) \wedge (seq(b) = seq(b'))) \Rightarrow b = b')$$

5. *Destination Safety* : An acknowlegement message a can only be sent by a site s if s is responsible for it or it has previously received a. The second condition allows rebroadcast.

$$\Box((send(s,a) \vee sent(s,a)) \Rightarrow (responsible(a) = s) \vee recvd(s,a))$$

6. *Timestamp Uniqueness*: A destination site cannot send two distinct acknowledgement messages with the same timestamp.

$$\Box(((send(s,a) \vee sent(s,a)) \wedge (send(s,a') \vee sent(s,a')) \wedge (ts(a) = ts(a')) \\ \wedge (responsible(a) = responsible(a') = s)) \Rightarrow a = a')$$

7. *Tokensite Safety* : A destination site cannot send an acknowledgement message a unless it has received a broadcast message matching the acknowledgement and for every timestamp t less than or equal to $ts(a)$ the destination has seen both an acknowledgement and a matching broadcast.

$$\Box((send(s,a) \vee sent(s,a)) \Rightarrow \forall t \leq ts(a).\, \exists a', b.(acks(b,a') \wedge (ts(a') = t) \wedge recvd(s,b) \wedge (((responsible(a') = s) \wedge (send(s,a') \vee sent(s,a')) \vee recvd(s,a'))))$$

Before we can prove Theorem 4.2.1, we prove five lemmas. The first lemma states that if a site receives a broadcast message, then the source of the message must have previously sent it. The second lemma is similar for acknowledgement messages. A site responsible for an acknowledgement message must have sent it before it can be received by any other site. Both lemmas are proved using induction on the number of occurrences of $\neg p$.

The proofs make use of the rule S9 in the calculus of occurrences of predicates. It asserts that the number of occurrences of a predicate is zero iff it is not the case that it held sometime in the past.

$$\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box((n = 0) \Leftrightarrow \neg\Diamond\varphi) \quad .$$

Lemma 4.2.1 Source origin : A broadcast message b must be sent by its source before it can be received by any site.

$$\vdash (rec(s,b) \vee recvd(s,b)) \Rightarrow sent(src(b), b)$$

Proof :

Use induction on the number of occurrences of $\neg p$. For the base case, (1) follows from the Network safety condition and assuming $\mathcal{N}(p, n_1) \wedge \mathcal{N}(\neg p, n_2) \wedge \mathcal{N}(q, n_3)$ holds.

1. $\Box((n_2 = 0) \Rightarrow (n_3 = 0))$

(2) follows from (1), $n_2 = 0$, Modus Ponens (R2) and S9.

2. $\neg\Diamond\, \exists s, m.rec(s,m)$

(3) follows from (2) and the definition of $recvd(s,m)$.

3. $\neg\, \exists s, m.recvd(s,m) \wedge \neg\, \exists s, m.rec(s,m)$

(4) follows from (3).

4. $(rec(s,b) \vee recvd(s,b)) \Rightarrow sent(src(b), b)$

The induction case is proved by assuming that the lemma holds when $n_2 = y$ and proving it for $n_2 = y + 1$, i.e.

5. $\vdash \Box((n_2 = y) \Rightarrow \varphi) \Rightarrow \Box((n_2 = y + 1) \Rightarrow \varphi)$

where $\varphi \triangleq rec(s,b) \vee recvd(s,b) \Rightarrow sent(src(b),b)$. Assume that $\Box((n_2 = y) \Rightarrow \varphi)$ and $n_2 = y+1 \wedge (rec(s,b) \vee recvd(s,b))$ hold and prove $send(src(b),b)$ holds. (6) follows from the assumption and Network safety.

6. $\boxdot((n_2 = y) \Rightarrow \exists s'.(send(s',b) \vee sent(s',b)))$

(7) follows from (6) and Source safety.

7. $\boxdot((n_2 = y) \Rightarrow \exists s'.(recvd(s',b) \vee (send(src(b),b) \vee sent(src(b),b))))$

(8) follows from (7) and the induction hypothesis.

8. $\boxdot((n_2 = y) \Rightarrow \exists s'.(sent(src(b),b) \vee (send(src(b),b) \vee sent(src(b),b))))$

(9) follows from (8).

9. $\boxdot((n_2 = y) \Rightarrow (send(src(b),b) \vee sent(src(b),b)))$

(10) follows from (9) and Network safety.

10. $\boxdot\Box((n_2 = y + 1) \Rightarrow sent(src(b),b))$

(11) follows from (10), $\vdash \boxdot\varphi \Rightarrow \varphi$ and $\vdash \Box\varphi \Rightarrow \varphi$.

11. $(n_2 = y + 1) \Rightarrow sent(src(b),b)$

(12) follows from (11), assumption and Modus Ponens (R2).

12. $sent(src(b),b)$

This proves the lemma. ∎

Lemma 4.2.2 Timestamp origin : An acknowledgement message a must be sent by the site responsible for it before it can be received by any site.

$\vdash (rec(s,a) \vee recvd(s,a)) \Rightarrow sent(responsible(a),a)$

Proof :

Similar to the proof for Lemma 4.2.1 ∎

The next two lemmas state the uniqueness of sequence numbers for broadcast messages and timestamps for acknowledgement messages.

Lemma 4.2.3 A destination site cannot receive two distinct broadcast messages with the same sequence number from the same source.

$$\vdash (rec(s,b) \vee recvd(s,b)) \wedge sent(src(b), b') \wedge seq(b) = seq(b') \wedge src(b') = src(b) \Rightarrow (b = b')$$

Proof :

To prove the lemma, we assume (1) and prove $b = b'$

1. $(rec(s,b) \vee recvd(s,b)) \wedge sent(src(b), b') \wedge seq(b) = seq(b') \wedge src(b') = src(b)$

(2) follows from (1) and Lemma 4.2.1

2. $sent(src(b), b)$

(3) follows from (1), (2) and Sequence Uniqueness.

3. $b = b'$

This proves the lemma. ■

Lemma 4.2.4 A destination site cannot receive two distinct acknowledgement messages with the same timestamp from the same source.

$$\vdash (rec(s,a) \vee recvd(s,a)) \wedge sent(responsible(a), a') \wedge ts(a) = ts(a') \Rightarrow a = a'$$

Proof :

Similar to the proof for Lemma 4.2.3 ■

Lemma 4.2.5 If a source has sent a message b and received an acknowledgement a' for it, then a destination which sends an acknowledgement a with a timestamp greater than or equal to a' must have received b.

$$\vdash (sent(src(b), b) \wedge recvd(src(b), a') \wedge acks(b, a') \wedge (send(s,a) \vee sent(s,a))$$

$$\wedge ts(a') \leq ts(a)) \Rightarrow recvd(s, b)$$

Proof :

To prove the lemma, we assume (1) and prove $recvd(s, b)$.

1. $sent(src(b), b) \wedge recvd(src(b), a') \wedge acks(b, a') \wedge (send(s, a) \vee sent(s, a))$
   $\wedge ts(a') \leq ts(a)$

(2) follows from (1) and Tokensite safety.

2. $\exists a''.(ts(a'') = ts(a') \wedge (recvd(s, a'') \vee ((send(s, a'') \vee sent(s, a''))$
   $\wedge (responsible(a'') = s))))$

(3) follows from (2) and Lemma 4.2.2

3. $sent(responsible(a''), a'') \vee send(responsible(a''), a'')$

(4) follows from (1) and Lemma 4.2.2

4. $sent(responsible(a'), a')$

(5) follows from (2), (3), (4), the definition of $sent(s, a)$ and Lemma 4.2.4

5. $a'' = a'$

(6) follows from (1), (5) and Tokensite safety.

6. $\exists b'.(acks(b', a') \wedge recvd(s, b'))$

(7) follows from (6) and the definition of $acks(b, a)$.

7. $src(a') = src(b') \wedge seq(a') = seq(b')$

(8) follows from (1) and the definition of $acks(b, a)$.

8. $src(b) = src(a') \wedge seq(b) = seq(a')$

(9) follows from (7), (8) and $\vdash (t_1 = t_2) \wedge (t_2 = t_3) \Rightarrow (t_1 = t_3)$.

9. $src(b') = src(b) \wedge seq(b') = seq(b)$

(10) follows from (1), (9) and Sequence uniqueness.

10. $b = b'$

(11) follows from (6) and (10).

11. $recvd(s, b)$

This proves the lemma. ■

We can now prove the safety theorem.

Theorem 4.2.1

$$\vdash (sent(s,b) \wedge (src(b) = s) \wedge \exists a, a'.(acks(b,a) \wedge recvd(s,a) \wedge recvd(s,a') \wedge (ts(a) + |D| \leq ts(a')))) \Rightarrow \forall s' \in D.recvd(s', b)$$

Proof :

To prove the theorem we assume (1) and prove $\forall s' \in D.recvd(s', b)$.

1. $sent(s,b) \wedge (src(b) = s) \wedge \exists a, a'.(acks(b,a) \wedge recvd(s,a) \wedge recvd(s,a') \wedge (ts(a) + |D| \leq ts(a')))$

(2) follows from (1) and Lemma 4.2.2

2. $sent(tokensite(ts(a')), a')$

(3) follows from (2) and Tokensite safety.

3. $\forall t \leq ts(a').\exists a''.((ts(a'') = t) \wedge (recvd(tokensite(ts(a')), a'') \vee (sent(tokensite(ts(a')), a'') \wedge responsible(a'') = tokensite(ts(a')))))$

(4) follows from (3) and Lemma 4.2.2

4. $\forall t \leq ts(a').\exists a''.((ts(a'') = t) \wedge sent(tokensite(ts(a'')), a''))$

(5) follows from (4) and arithmetic.

5. $|D| \leq ts(a') \Rightarrow (\forall t.(ts(a') - |D| < t \leq ts(a') \Rightarrow \exists a''.(ts(a'') = t \wedge sent(tokensite(ts(a'')), a'')))$

(6) follows (5) and Tokensite coverage.

6. $\forall d \in D.(\exists a''.sent(d, a''))$

(7) follows from (6) and Lemma 4.2.5

7. $\forall d \in D.recvd(d, b)$

This proves the theorem. ■

4.3 Verification of the Implementation

Having developed an abstract specification of the algorithm and proved its correctness we are now ready to describe the detailed implementation of the Chang and Maxemchuk algorithm in terms of sequences and other low level data structures. We proceed by stating the algorithm and proving the safety properties of the implementation. We then relate the implementation to the specification and prove its correctness by showing that the implementation satisfies the specification.

4.3.1 The Implementation

The algorithm is stated in terms of two data structures : $SrcCtl(s)$ for sources and $DesCtl(s)$ for destinations. The algorithm of the source is as follows. When a broadcast message b is sent, the source stores b until it receives an acknowledgement message from the tokensite. It then puts the message and the acknowledgement in a queue. The source also records the highest timestamp of the acknowledgement messages it receives. Let $SrcCtl(s)$ be a tuple (m, q, n, t) where m is the current message being transmitted, q is the queue of broadcast messages and their acknowledgements, n is the sequence number of the next message to be transmitted and t is the highest timestamp of the acknowledgement messages. The source algorithm is formally stated as follows.

Let $lsend(s, m)$ and $lrec(s, m)$ be predicates such that $lsend(s, m)$ holds iff a site s sends a message m and $lrec(s, m)$ hold iff a site s receives a message m. These are implementation versions of $send(s, m)$ and $rec(s, m)$.

$lsend(s, m)$	: a site s sends a message m
$lrec(s, m)$	: a site s receives a message m

The source algorithm can now be formalised in terms of the number of occurrences of $\neg \exists m.lsend(s,m)$ and $\neg \exists m.lrec(s,m)$.

Let *init* be an abbreviation for $\oslash false$ and *nullm* be an empty message. Then, initially the value of m is *nullm*, q is empty, and n and t are both zero.

$$SrcAl_a \triangleq \Box(init \vee s \notin S \Rightarrow SrcCtl(s) = (nullm, <>, 0, 0))$$

A site can send a message b if m is empty. Following the send operation, b is stored in m and the sequence number of the next message is incremented. Let p' be defined as $\exists m.lsend(s,m)$ and q' as $\exists m.lrec(s,m)$. Then,

$$\begin{aligned} SrcAl_b \triangleq & \mathcal{N}(\neg p', n_2) \wedge \mathcal{N}(\neg q', n_4) \Rightarrow \Box((s \in S \\ & \wedge SrcCtl(s) = (nullm, q, n, t) \wedge (n_2 = u) \wedge (n_4 = v) \wedge lsend(s,b) \\ & \wedge src(b) = s \wedge seq(b) = n) \Rightarrow \Box((n_2 = u+1) \wedge (n_4 = v) \\ & \Rightarrow SrcCtl(s) = (b, q, n+1, t))) \end{aligned}$$

If a site receives an acknowledgement a for a message b it has broadcast then it puts the acknowledgement and the message in the queue q, clears m and updates the value of t if the timestamp of a is greater than t.

$$\begin{aligned} SrcAl_c \triangleq & \mathcal{N}(\neg p', n_2) \wedge \mathcal{N}(\neg q', n_4) \Rightarrow \Box((s \in S \wedge SrcCtl(s) = (m, q, n, t) \\ & \wedge (n_2 = u) \wedge (n_4 = v) \wedge lrec(s,a) \wedge acks(m,a)) \\ & \Rightarrow \Box((n_2 = u) \wedge (n_4 = v+1) \Rightarrow Srcdata)) \end{aligned}$$

where *Srcdata* is defined as :

$$Srcdata \triangleq SrcCtl(s) = (nullm, <(m,a)> q, n, max(t, ts(a)))$$

If the acknowledgement is not for the message it has sent then the value of t is updated with the timestamp of a if it is greater.

$$\begin{aligned} SrcAl_d \triangleq & \mathcal{N}(\neg p', n_2) \wedge \mathcal{N}(\neg q', n_4) \Rightarrow \Box((s \in S \wedge SrcCtl(s) = (m, q, n, t) \\ & \wedge (n_2 = u) \wedge (n_4 = v) \wedge lrec(s,a)) \wedge \neg acks(m,a)) \\ & \Rightarrow \Box((n_2 = u) \wedge (n_4 = v+1) \\ & \Rightarrow SrcCtl(s) = (m, q, n, max(t, ts(a))))) \end{aligned}$$

The source algorithm is asserted as:

$$SrcAl \triangleq SrcAl_a \wedge SrcAl_b \wedge SrcAl_c \wedge SrcAl_d$$

We next describe the destination algorithm. When a destination receives a broadcast message b it checks to see if the message has been received before. It does this by checking the sequence number of b with the next sequence number it expects from the source of b. If the sequence number is greater than or equal to the number it expects then b is stored in a queue to be processed. If the destination receives an acknowledgement it checks to see if it has already been received by checking if it is in the queue of the received acknowledgements and that its timestamp is greater than or equal to the timestamp of the next acknowledgement to be processed. This suggests a data structure $DesCtl(s) = (q_c, q_b, q_t, nts, nscq)$ where q_c is a queue of acknowledgements to be processed, q_b is a queue of broadcast messages to be processed, q_t is a queue of broadcast message and acknowledgement pairs which have been processed, nts is the timestamp of the next acknowledgement it expects and $nseq$ is a list of sequence numbers of the next broadcast message it expects from each source. The processing of the received messages at a site s is specified using a recursive function $process_s$ defined below.

$$\begin{aligned}
&process_s((q_c, q_b, q_t, nts, nseq)) \\
&\quad = process_s((q_c', q_b', q_t', nts', nseq')) \text{ if } q_c = q < a > \\
&\qquad\qquad \wedge \exists\, b \in q_b.acks(b, a) \\
&\qquad\qquad \wedge nts = ts(a) \\
&\qquad\qquad \wedge nts' = nts + 1 \\
&\qquad\qquad \wedge nseq'[src(b)] = seq(b) \\
&\qquad\qquad \wedge nseq'[s \neq src(b)] = nseq[s] \\
&\qquad\qquad \wedge q_c' = q \\
&\qquad\qquad \wedge q_b' = q_b \backslash b \\
&\qquad\qquad \wedge q_t' =< (a, b) > q_t \\
&\quad = process_s((< a >, q_b, q_t, nts', nseq)) \text{ if } q_c =<> \\
&\qquad\qquad \wedge tokensite(nts) = s \\
&\qquad\qquad \wedge ts(a) = nts \\
&\qquad\qquad \wedge \exists\, b \in q_b.acks(b, a) \\
&\qquad\qquad \wedge seq(b) = nseq[src(b)] \\
&\qquad\qquad \wedge nts' = nts + 1 \\
&\quad = (q_c, q_b, q_t, nts, nseq) \qquad\qquad \text{otherwise}
\end{aligned}$$

In the definition of $process_s$, we have used the notation <> to mean an empty queue, $q < a >$ to mean that the element a is appended to the front of the queue q, $< a > q$ to mean that the element a is appended to the tail of the queue q, $q \backslash a$ to mean that a is removed from the queue q, $b \in q$ to mean that b

is in the queue q and $nseq[s]$ to mean the element corresponding to the index s in the sequence $nseq$. $b \notin q$ is used later to mean that b is not in the queue q.

A site processes a broadcast message b by removing its acknowledgement at the head of q_c and removing b from q_b. It then appends the pair to the queue q_t. The counter of the next expected timestamp nts and the next sequence number expected from the source of b are incremented. If the site is the tokensite then it generates an acknowledgement for a broadcast message in q_b and increments nts.

We define $DesCtl(s)$ in terms of $process_s$ and the number of occurrences of $\neg \exists m.lrec(s,m)$. Initially, the values of q_c, q_b, q_t are all null and the values of nts and elements of $nseq$ are zero.

$$DesAl_a \triangleq \Box(init \vee s \notin D \Rightarrow DesCtl(s) = (<>, <>, , <>, 0, (0^{|D|})))$$

In the above formula, we have used an abbreviation m^n to denote a list of n elements each of value m.

If a site receives an acknowledgement a which it has not received before, it appends a to q_c to be processed.

$$\begin{aligned} DesAl_b \triangleq & \mathcal{N}(\neg q', n_4) \Rightarrow \Box((s \in D \wedge DesCtl(s) = (q_c, q_b, , q_t, nts, nseq) \\ & \wedge (n_4 = u) \wedge lrec(s,a) \wedge nts \leq ts(a) \wedge a \notin q_c) \Rightarrow \Box((n_4 = u+1) \\ & \Rightarrow DesCtl(s) = process_s(< a > q_c, q_b, q_t, nts, nseq))) \end{aligned}$$

If a site receives a broadcast message which it has not received before then it is appended to q_b to be processed.

$$\begin{aligned} DesAl_c \triangleq & \mathcal{N}(\neg q', n_4) \Rightarrow \Box((s \in D \wedge DesCtl(s) = (q_c, q_b, , q_t, nts, nseq) \\ & \wedge (n_4 = u) \wedge lrec(s,b) \wedge nseq(src(b)) \leq seq(b) \wedge b \notin q_b) \Rightarrow \\ & \Box((n_4 = u+1) \\ & \Rightarrow DesCtl(s) = process_s(q_c, < b > q_b, q_t, nts, nseq))) \end{aligned}$$

The destination algorithm is stated as:

$$DesAl \triangleq DesAl_a \wedge DesAl_b \wedge DesAl_c$$

Note that any messages which have already been received are ignored.

We now specify the behaviour of a site s. A site is either idle, or it makes a request to send a message or is in an error state. A site is in an error state iff it has sent a message which it did not request to send, for example, by faulty transmission. The behaviour is captured as follows:

Initially, a site is not in an error state.

$$SAl_a \triangleq \mathcal{N}(p', n_1) \Rightarrow \Box((n_1 = 0) \Rightarrow \neg error(s))$$

If a site is in an error state or it sends a message it did not request to send then it will always remain in the error state. Let $rsend(s, m)$ denote a request by a site s to send a message m. Then,

$$\begin{aligned} SAl_b \triangleq \mathcal{N}(\neg p', n_2) \Rightarrow \Box(((n_2 = y) \wedge (error(s) \vee (lsend(s, m') \\ \wedge \neg rsend(s, m')))) \Rightarrow \Box((n_2 = y + 1) \Rightarrow error(s))) \end{aligned}$$

If a site is in an error state then it must previously must have sent a message it did not request to send.

$$\begin{aligned} SAl_c \triangleq \mathcal{N}(p', n_1) \Rightarrow \Box((n_1 = y + 1) \wedge \neg error(s) \\ \Rightarrow \boxdot((n_1 = y) \Rightarrow \neg error(s))) \end{aligned}$$

If a site requests to send a broadcast message b then either it is a source of b and b is a new message or it is rebroadcasting the current message. If it is not a source of b then it must have previously received b and seen an acknowledgement for it.

$$\begin{aligned} SAl_d \triangleq \Box(rsend(s, b) \Rightarrow (SrcCtl(s) = (m, q, n, t) \wedge src(b) = s \\ \wedge((m = nullm \vee seq(b) = n) \vee (m = b \wedge \exists a.(b, a) \in q))) \\ \vee(DesCtl(s) = (q_c, q_b, q_t, nts, nseq) \wedge \exists a.(b, a) \in q_t)) \end{aligned}$$

If a site requests to send an acknowledgement then it must have received a broadcast message for it.

$$SAl_e \triangleq \Box(rsend(s, a) \Rightarrow DesCtl(s) = (q_c, q_b, q_t, nts, nseq) \wedge \exists b.(b, a) \in q_t)$$

The behaviour of a site is stated as:

$$SAl \triangleq SAl_a \wedge SAl_b \wedge SAl_c \wedge SAl_d \wedge SAl_e$$

4.3.2 Implementation Conditions and Safety Properties

We now define the implementation versions of $lsent(s,m)$ and $lrecvd(s,m)$ in terms of $lsend(s,m)$ and $lrec(s,m)$ and prove that the implementation also satisfies the safety conditions.

The predicates $lsent(s,m)$ and $lrecvd(s,m)$ indicate that a message has been sent and received respectively. They are defined as follows:

$$lsent(s,m) \triangleq \neg lsend(s,m) \wedge \Diamond lsend(s,m)$$

$$lrecvd(s,m) \triangleq \neg lrec(s,m) \wedge \Diamond lrec(s,m)$$

The predicates $lsent(s,m)$ and $lrecvd(s,m)$ satisfy the following properties.

A message which has been sent cannot be 'un-sent'.

$$\begin{aligned}\mathcal{N}(p',n_1) \Rightarrow \Box((n_1 = y) \wedge (lsent(s,m) \vee lsend(s,m)) \\ \Rightarrow \Box((n_1 = y+1) \Rightarrow lsent(s,m)))\end{aligned}$$

An *lsend* operation must previously been executed for a message to be sent.

$$\begin{aligned}\mathcal{N}(p',n_1) \Rightarrow \Box((n_1 = y+1) \wedge lsent(s,m) \\ \Rightarrow \boxdot((n_1 = y) \Rightarrow (lsent(s,m) \vee lsend(s,m))))\end{aligned}$$

A message which has been received cannot be 'un-received'.

$$\begin{aligned}\mathcal{N}(q',n_3) \Rightarrow \Box((n_3 = y) \wedge (lrecvd(s,m) \vee lrec(s,m)) \\ \Rightarrow \Box((n_3 = y+1) \Rightarrow lrecvd(s,m)))\end{aligned}$$

An *lrec* operation must previously been executed for a message to be received.

$$\begin{aligned}\mathcal{N}(q',n_3) \Rightarrow \Box((n_3 = y+1) \wedge lrecvd(s,m) \\ \Rightarrow \boxdot((n_3 = y) \Rightarrow (lrecvd(s,m) \vee lrec(s,m))))\end{aligned}$$

We now prove the implementation versions of the safety properties.

Lemma 4.3.1 *Implementation Source Safety* : If a site is not in an error state and it sends a broadcast message b then it must be the source of b or have previously received b.

$$\vdash (lsend(s,b) \vee lsent(s,b)) \wedge \neg error(s) \Rightarrow (src(b) = s \vee lrecvd(s,b))$$

Proof :

Use induction on the number of occurrences of p'. (1) follows from S9 and by assuming $\mathcal{N}(p', n_1)$ holds.

1. $\Box((n_1 = 0) \Leftrightarrow \neg\Diamond \exists m.lsend(s,m))$

(2) follows from (1), $n_1 = 0$ and Modus Ponens (R2).

2. $\neg\Diamond \exists m.lsend(s,m)$

(3) follows from (2).

3. $(lsend(s,b) \vee lsent(s,b)) \wedge \neg error(s) \Rightarrow (src(b) = s \vee lrecvd(s,b))$

The induction case is proved by assuming that the lemma holds when $n_1 = y$ and proving it for $n_1 = y + 1$, i.e.

4. $\vdash \Box((n_1 = y) \Rightarrow \varphi) \Rightarrow \Box((n_1 = y + 1) \Rightarrow \varphi)$

where

$$\varphi \triangleq (lsend(s,b) \vee lsent(s,b)) \wedge \neg error(s) \Rightarrow (src(b) = s \vee lrecvd(s,b))$$

Assume that $\Box((n_1 = y) \Rightarrow \varphi)$ and $n_1 = y + 1 \wedge (lsend(s,b) \vee lsent(s,b)) \wedge \neg error(s)$ hold and prove $src(b) = s \vee lrecvd(s,b)$ holds. (5) follows from the assumption and Implementation condition.

5. $lsend(s,b) \vee \boxdot((n_1 = y) \Rightarrow lsend(s,b) \vee lsent(s,b))$

(6) follows from the assumption and the definition of a site.

6. $\boxdot((n_1 = y) \Rightarrow \neg error(s))$

(7) follows from (5), (6) and the definition of a site.

7. $rsend(s,b) \vee \boxdot((n_1 = y) \Rightarrow rsend(s,b) \vee lsent(s,b))$

(8) follows from (7) and the definition of a site.

8. $src(b) = s \vee (DecCtl(s) = (q_c, q_b, q_t, nseq, nts) \wedge \exists a.(b,a) \in q_t)$
   $\vee \boxdot((n_1 = y) \Rightarrow (src(b) = s$
   $\vee (DecCtl(s) = (q_c, q_b, q_t, nseq, nts) \wedge \exists a.(b,a) \in q_t))$
   $\vee lsent(s,b)))$

(9) follows from (8) and the definition of $DesCtl(s)$.

9. $src(b) = s \vee lrecvd(s,b) \vee$
   $\vee \boxdot((n_1 = y) \Rightarrow (src(b) = s \vee lrecvd(s,b) \vee lsent(s,b)))$

(10) follows from (9) and the induction hypothesis.

10. $src(b) = s \vee lrecvd(s,b)$
    $\vee \boxdot((n_1 = y) \Rightarrow (src(b) = s \vee lrecvd(s,b)))$

(11) follows from (10), the definition of a source and Implementation condition.

11. $src(b) = s \vee lrecvd(s,b)$
    $\vee \boxdot\Box((n_1 = y + 1) \Rightarrow (src(b) = s \vee lrecvd(s,b)))$

(12) follows from (11).

12. $src(b) = s \vee lrecvd(s,b)$
    $\vee((n_1 = y + 1) \Rightarrow (src(b) = s \vee lrecvd(s,b)))$

(13) follows from the assumption and (12).

13. $src(b) = s \vee lrecvd(s,b)$

This proves the lemma. ∎

Lemma 4.3.2 *Implementation Sequence Number Uniqueness* : A site which is not in an error state cannot send two distinct broadcast messages with the same sequence number.

$\vdash (\neg error(s) \wedge (lsend(s,b) \vee lsent(s,b)) \wedge (lsend(s,b') \vee lsent(s,b'))$
$\wedge s = src(b) = src(b')) \Rightarrow b = b'$

Proof :

Use induction on the number of occurrences of p'. (1) follows from S9.

1. $\Box((n_2 = 0) \Leftrightarrow \neg\Diamond \exists m.lsend(s,m))$

(2) follows from (1), $n_1 = 0$ and Modus Ponens (R2).

2. $\neg\Diamond \exists m.lsend(s,m)$

(3) follows from (2).

3. $\neg error(s) \wedge (lsend(s,b) \vee lsent(s,b)) \wedge (lsend(s,b') \vee lsent(s,b'))$
$\wedge s = src(b) = src(b') \Rightarrow b = b'$

The induction case is proved by assuming that the lemma holds when $n_1 = y$ and proving it for $n_1 = y + 1$.

4. $\vdash \Box((n_1 = y) \Rightarrow \varphi) \Rightarrow \Box((n_1 = y + 1) \Rightarrow \varphi)$

where

$\varphi \triangleq \neg error(s) \wedge (lsend(s,b) \vee lsent(s,b))$
$\wedge (lsend(s,b') \vee lsent(s,b')) \wedge s = src(b) = src(b') \Rightarrow b = b'$

Assume that $\Box(n_1 = y \Rightarrow \varphi)$ and $n_1 = y + 1 \wedge \neg error(s) \wedge (lsend(s,b) \vee lsent(s,b)) \wedge (lsend(s,b') \vee lsent(s,b')) \wedge s = src(b) = src(b')$ hold and prove $b = b'$ holds. (5) follows from the assumption and the implementation conditions.

5. $(lsend(s,b) \wedge lsend(s,b')) \vee \boxdot(n_1 = y \Rightarrow ((lsend(s,b) \vee lsent(s,b))$
$\wedge (lsend(s,b') \vee lsent(s,b')))$

(6) follows from the assumption and the definition of a site.

6. $\boxdot((n_1 = y) \Rightarrow \neg error(s))$

(7) follows from the definition of a source and a sequence number.

7. $\boxdot((n_1 = y) \Rightarrow src(b) = src(b') \wedge seq(b) = seq(b'))$

(8) follows from (5), (6), (7) and the induction hypothesis.

8. $(lsend(s,b) \wedge lsend(s,b')) \vee \boxdot((n_1 = y) \Rightarrow b = b')$

(9) follows from (8) and the definition of a message.

9. $(lsend(s,b) \wedge lsend(s,b')) \vee \boxdot\Box((n_1 = y + 1) \Rightarrow b = b')$

(10) follows from (9).

10. $(lsend(s,b) \wedge lsend(s,b')) \vee ((n_1 = y + 1) \Rightarrow b = b')$

(11) follows from the assumption and (10).

11.$(lsend(s,b) \wedge lsend(s,b')) \vee b = b'$

(12) follows from (11) and the definition of $SrcCtl(s)$.

12. $b = b'$

This proves the lemma. ■

Lemma 4.3.3 *Implementation Destination Safety* : If a site is not in an error state and sends an acknowledgement a then it is either responsible for a or has previously received a.

$$\vdash (lsend(s,a) \vee lsent(s,a)) \wedge \neg error(s) \Rightarrow (responsible(a) = s \vee lrecvd(s,a))$$

Proof :

Slight variation of the proof for Lemma 4.3.1 ■

Lemma 4.3.4 *Implementation Timestamp Uniqueness* : A site which is not in an error state cannot send two distinct acknowledgement messages with the same timestamp.

$$\vdash ((lsend(s,a) \vee lsent(s,a)) \wedge (lsend(s,a) \vee lsent(s,a')) \wedge \neg error(s) \wedge ts(a) = ts(a') \wedge responsible(a) = s) \Rightarrow a = a'$$

Proof :

Slight variation of the proof for Lemma 4.3.2 ■

Lemma 4.3.5 *Implementation Tokensite Safety* : A site which is not in an error state cannot send an acknowledgement message a unless it has received a broadcast message matching the acknowledgement and for every timestamp t less than or equal to $ts(a)$ the destination has seen both an acknowledgement and a matching broadcast.

$$\vdash (lsend(s,a) \vee lsent(s,a)) \wedge \neg error(s) \Rightarrow \forall t \leq ts(a).\, \exists a', b.(acks(b,a') \wedge ts(a') = t \wedge lrecvd(s,b) \wedge((responsible(a') = s \wedge lsent(s,a')) \vee lrecvd(s,a')))$$

Proof :

Slight variation of the proof for Lemma 4.3.1 ■

4.4 Relating Implementation to the Specification

Having described the implementation and proved its safety properties, we must now prove that it meets the specification. This we do by stating two global properties that the implementation must satisfy and by relating $send(s,m)$ and $rec(s,m)$ to $lsend(s,m)$ and $lrec(s,m)$ respectively. We then prove that the safety properties of the implementation imply the safety properties of the abstract specification. Thus, the implementation satisfies the specification, i.e. the safety theorem.

The first safety property asserts that a site only sends a message it has requested to be sent.

Global Safety Property 1

$$\Box(lsend(s,m) \Rightarrow rsend(s,m))$$

Using this property we can prove that the sites never enter an error state.

Lemma 4.4.1 : Sites never enter the error state.

$$\vdash \neg error(s)$$

Proof :

Use induction on the number of occurrences of p'. (1) follows from the definition of a site and assuming $\mathcal{N}(p', n_1)$.

1. $\Box((n_1 = 0) \Rightarrow \neg error(s))$

(2) follows from (1), $n_1 = 0$ and Modus Ponens (R2).

2. $\neg error(s)$

The induction case is proved by assuming that the lemma holds when $n_1 = y$ and proving it for $n_1 = y + 1$, i.e.

3. $\vdash \Box((n_1 = y) \Rightarrow \neg error(s)) \Rightarrow \Box((n_1 = y + 1) \Rightarrow \neg error(s))$

Assume $\Box((n_1 = y) \Rightarrow \neg error(s))$ and prove $\Box((n_1 = y + 1) \Rightarrow \neg error(s))$. (4) follows from the assumption, the definition of a site, Global property 1 and Modus Ponens (R2).

4. $\Box((n_1 = y + 1) \Rightarrow \neg error(s))$

This proves the lemma. ■

Now, if we 'implement' $send(s,m)$ and $rec(s,m)$ by $lsend(s,m)$ and $lrec(s,m)$ then $send(s,m) \Leftrightarrow lsend(s,m)$ and $rec(s,m) \Leftrightarrow lrec(s,m)$. Using this relation, the definitions of $sent(s,m)$, $recvd(s,m)$, $lsent(s,m)$, $lrecvd(s,m)$ and Lemma 4.4.1 it is easy to prove that the implementation safety properties imply the abstract specification ones (i.e. properties 3-7 of Section 4.2). Therefore, all that remains to be done is to prove the network safety properties of Section 4.2.

In order to prove the network safety properties we assert the second global safety property.

Global Safety Property 2

$$\Box((\exists s.lrec(s,m) \vee \exists s.lrecvd(s,m)) \Rightarrow \exists s'.lsent(s',m))$$

The statement asserts that a message can only be received if it has been sent by a site.

Using this property and the definitions of $lsend(s,m)$ and $lrec(s,m)$ the proofs are as follows:

The proof of the Network safety 1A is as follows :

$\vdash \mathcal{N}(p, n_1) \wedge \mathcal{N}(\neg p, n_2) \wedge \mathcal{N}(q, n_3) \Rightarrow \Box(((n_1 = 0) \vee (n_2 = 0)) \Rightarrow (n_3 = 0))$

Proof :

To prove the safety property we assume (1), $\mathcal{N}(p, n_1) \wedge \mathcal{N}(\neg p, n_2) \wedge \mathcal{N}(q, n_3)$ and prove $n_3 = 0$.

1. $n_1 = 0 \vee n_2 = 0$

(2) follows from (1) and S9.

2. $\neg \Diamond \exists m.lsend(s, m) \vee \boxdot \exists m.lsend(s, m)$

(3) follows from (2) and the definition of $lsent(s, m)$.

3. $\boxdot \neg \exists m.lsent(s, m)$

(4) follows from (3) and the Global safety property 2.

4. $\boxdot \neg \exists s.lrec(s, m)$

(5) follows from (4) and S9.

5. $n_3 = 0$

This proves the safety property. ∎

The proof of the Network safety 1B is as follows :

$$\vdash \mathcal{N}(\neg p, n_2) \Rightarrow \Box((n_2 = y + 1) \wedge (rec(s, m) \vee recvd(s, m)) \Rightarrow \boxdot((n_2 = y) \Rightarrow \exists s'.(send(s', m) \vee sent(s', m))))$$

Proof :

To prove the property, we prove (1) which follows from the definitions of $send(s, m)$ and $rec(s, m)$.

1. $\Box((n_2 = y + 1) \wedge (lrec(s, m) \vee lrecvd(s, m)) \Rightarrow \boxdot((n_2 = y) \Rightarrow \exists s'.(lsent(s', m) \vee lsend(s', m))))$

(2) follows from (1) and Global safety property 2.

2. $\Box((n_2 = y + 1) \wedge \exists s'.lsent(s', m) \Rightarrow \boxdot((n_2 = y) \Rightarrow \exists s'.(lsent(s', m) \vee lsend(s', m))))$

(2) is satified by the implementation conditions. This proves the safety property. ∎

The proofs of the Network safety properties 1C and 1D are slight variations of the proof for the Network safety 1B. This completes the proofs to show that the implementation satisfies the specification.

4.5 Discussion

In this chapter, we specified an algorithm and proved the correctness of its implementation. The proofs used induction on the number of occurrences of predicates. The analysis given here can easily be extended to reason about the algorithm with timing assumptions, for example, by including the maximum duration a transmitter waits for an acknowledgement. The algorithm can also be specified and proved using the 'raw' logic. For example, the property that a message cannot be 'un-received' is stated in it as:

$$\Box(rec(s,m) \Rightarrow \Box(\neg rec(s,m)\mathcal{S}^{+}rec(s,m) \Rightarrow recvd(s,m)))$$
$$\wedge\Box(recvd(s,m) \Rightarrow \Box recvd(s,m))$$

However, the proofs in the 'raw' logic are generally more difficult because of the need to reason about nested *until* and *since* operators.

The example was originally specified and proved in [YR92] using modal primitive recursive (m.p.r) functions. To state using an m.p.r function that initially no site has received any message one writes

$$StartState() \Rightarrow \neg recvd(s,m)$$

where $StartState()$ is a modal boolean function which holds only in the initial start state. We can state that an assertion is true after an input by using the $after$ m.p.r function modifier. For example, to state that a message cannot be 'un-received' we write

$$recvd(s,m) \Rightarrow after(< \chi >, recvd(s,m))$$

where χ is an input symbol. The proofs are done using induction on inputs. To prove a property φ, we prove $StartState() \Rightarrow \varphi$ for the base case, and $\varphi \Rightarrow after(< \chi >, \varphi)$ for the induction step.

A major difference between our treatment and that of using m.p.r functions is that in the latter actions are considered instantaneous whereas we do not make any such assumptions. This is reflected in our specification. For example, the source safety property is stated as :

$$sent(s,b) \Rightarrow ((src(b) = s) \vee recvd(s,b))$$

using m.p.r functions. But in our analysis, actions may have duration and therefore the same property is stated as :

$$send(s,b) \vee sent(s,b) \Rightarrow ((src(b) = s) \vee recvd(s,b))$$

This allows the specification to be easily extended to include timing constraints on the maximum duration of the send operation or the minimum duration a transmitter must wait before it can retransmit a message.

Chapter 5

Example : Digital Circuits

The calculus defined in Chapter 3 is useful in the specification and verification of real-time properties where the duration of an occurrence of an event is specified or when we can assert properties by stating what is true before and after an occurrence of an event. For example, in the gas burner problem, the maximum duration of an occurrence of a gas leak and the minimum duration for which a leak cannot occur after a gas leak has occurred were given. Using these relations we could specify the design decisions and the invariants in terms of the number of occurrences of leak. Similarly, the fault-tolerant broadcast algorithm of Chapter 4 was specified and verified using the number of occurrences of send and receive operations because we could state the properties before and after these events. The calculus also allowed us to specify and prove properties without using nested *until* and *since* operators. However, in certain cases it may not be straightforward to specify real-time properties in terms of the number of occurrences of a predicate. Consider, the following example :

> " The output Y of a delay circuit after δ time units is the current value of the input X."

This property cannot be stated in terms of the number of occurrences of either X or Y. We therefore have to state it in terms of the 'raw' logic and prove timing properties using an alternative induction principle. The induction principles we use are stated as follows :

C-IND

$$\frac{\vdash \varphi \Rightarrow \odot\varphi}{\vdash \varphi \Rightarrow \Box\varphi}$$

and

DC-IND

$$\frac{\vdash \varphi \Rightarrow (\psi \wedge \odot\varphi)}{\vdash \varphi \Rightarrow \Box\psi}$$

The rule C-IND (computation induction) states that if $\varphi \Rightarrow \odot\varphi$ is a theorem then so is $\varphi \Rightarrow \Box\varphi$. The rule DC-IND (derived computation induction) states that if $\varphi \Rightarrow (\psi \wedge \odot\varphi)$ is a theorem then $\varphi \Rightarrow \Box\psi$ is also a theorem. The proofs of these theorems are similar to the ones given for TL [MP82] and can be applied irrespective of the time domain that is used. To illustate their use in proving safety properties, we apply the logic in the specification and verification of timing properties of digital circuits. We begin in Section 5.1 by specifying some simple and basic elements such as delay circuits, invertors, and adders from which others can be constructed. In Section 5.2, we consider a specific kind of circuit which is delay-insensitive. Delay-insensitive circuits are those which are insensitive to any variations in delays in gates and wires [HCS92]. Therefore, parts of a circuit can be replaced by faster or slower ones without affecting its behaviour. We prove for a simple oscillator that the circuit after transformation is a valid simplification of the one it replaces using the computation induction principles.

5.1 Combinational Elements

In this section, we describe some basic elements which can be used in the description of more complex ones.

Rising and Falling Signals

A rising signal (i.e. from *false* to *true*) is described by $\uparrow X$:

$$\uparrow X \triangleq X \wedge \oslash\neg X$$

A falling signal (i.e. from *true* to *false*) is described by $\downarrow X$:

$$\downarrow X \triangleq \neg X \wedge \ominus X$$

Remark 5.1.1 : Note that $\uparrow X$ and $\downarrow X$ can be defined as above because the *next* operator in the logic is defined as the next time point at which a state transition occurs if such a time point exists in the future or else it is the time point obtained by adding one to the current time.

■

These operators can be extended to include timing information, for example, to state the minimum duration a signal must remain stable before and after a transition. $\uparrow^{u,v} X$ states that a signal X is stable for at least u time units before and v time units after the transition from *false* to *true*.

$$\uparrow^{u,v} X \triangleq \uparrow X \wedge (T = x) \wedge \boxdot(((\odot T) > x - u) \Rightarrow \neg X) \wedge \Box((T \leq x + v) \Rightarrow X)$$

$\downarrow^{u,v} X$ states that a signal X is stable for at least u time units before and v time units after the transition from *true* to *false*.

$$\downarrow^{u,v} X \triangleq \downarrow X \wedge (T = x) \wedge \boxdot(((\odot T) > x - u) \Rightarrow X) \wedge \Box((T \leq x + v) \Rightarrow \neg X)$$

A negative pulse can be described as follows :

$$\downarrow\uparrow X \triangleq \neg X \mathcal{U} \uparrow X \wedge \neg X \mathcal{S} \downarrow X$$

Delay

Delays in gates illustrated in Figure 5.1 can be modelled as follows :

The value of the output Y at time $x + \delta$ is equal to the value of the input X at time x where δ is the delay associated with the gate.

$$X \ \ del^{\delta} \ \ Y \triangleq \forall u, x, c. \Box(((T = u) \wedge (X = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq x + \delta) \wedge \odot(T > x + \delta) \Rightarrow Y = c))$$

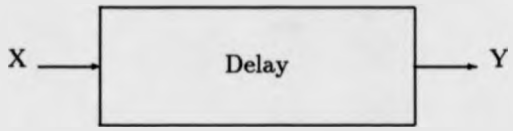

Figure 5.1: A delay element

In the definition of the delay operator $X = c = 0$ iff X does not hold and $X = c = 1$ iff X holds.

Remark 5.1.2 Note that we cannot describe the delay element in the logic by the following statement.

$$\forall u, x, c. \Box(((T = u) \wedge (X = c)) \Rightarrow \Box((T = u + \delta) \Rightarrow Y = c))$$

■

A wire W can be described as a gate with a delay of δ_ω time units as follows:

$$W \triangleq X \ \ del^{\delta_\omega} \ \ Y$$

The delay operator satisfies the following property.

$$\vdash X \ \ del^{\delta_1} \ \ Y \wedge Y \ \ del^{\delta_2} \ \ Z \Rightarrow X \ \ del^{\delta_1+\delta_2} \ \ Z$$

Proof :

To prove the property, we assume (1) and prove $X \ \ del^{\delta_1+\delta_2} \ \ Z$.

1. $X \ \ del^{\delta_1} \ \ Y \wedge Y \ \ del^{\delta_2} \ \ Z$

(2) follows from (1).

2. $\forall u, x, c. \Box(((T = u) \wedge (X = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq x + \delta_1) \wedge \odot(T > x + \delta_1) \Rightarrow Y = c))$

(3) follows from (1).

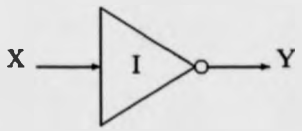

Figure 5.2: An invertor

3. $\forall u, x, c.\Box(((T = u) \wedge (Y = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq x + \delta_2) \wedge \odot(T > x + \delta_2) \Rightarrow Z = c))$

(4) follows from (2), (3) and arithmetic.

4. $\forall u, x, c.\Box(((T = u) \wedge (X = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq u + \delta_1 + \delta_2) \wedge \odot(T > x + \delta_1 + \delta_2) \Rightarrow Z = c))$

(5) follows from (4) and the definition of del^{δ}.

5. $X \;\; del^{\delta_1+\delta_2} \;\; Z$

This proves the property. ■

Invertor

For an invertor (Figure 5.2), the value of the output Y at time $x + \delta_i$ is the negation of the value of the input X at time x.

$$I \triangleq \forall u, x, c.\Box(((T = u) \wedge (X = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq x + \delta_i) \wedge \odot(T > x + \delta_i) \Rightarrow Y = c \bmod 1))$$

Inverting C-Gate

An inverting C-gate (Figure 5.3) has two inputs X and Y and one output Z. If X and Y are equal then the value of Z after a delay of δ_g time units is its current value. If X and Y are not equal then the value of Z after a delay of δ_g time units is the current value of Y.

It is specified as follows :

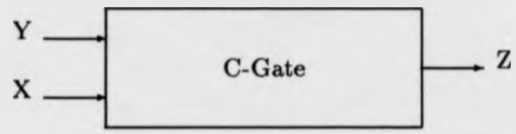

Figure 5.3: An inverting C-gate

$$G \triangleq \forall u, x, c.\Box((((T = u) \wedge \neg(X \Leftrightarrow Y) \wedge (Y = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)) \wedge (((T = u) \wedge (X \Leftrightarrow Y) \wedge (Z = c) \wedge (u \leq x < \odot T)) \Rightarrow \Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)))$$

Adder

An adder performs addition on numbers which are implemented by bit-vectors. For example, if X and Y are n-bit vectors denoting input parameters and Z is an n-bit output vector then an adder with a delay of δ_a time units can be specified as follows :

$$\forall u, w, x, y.\Box(((T = u) \wedge (X = x) \wedge (Y = y) \wedge (u \leq w < \odot T)) \Rightarrow \Box((T \leq w + \delta_a) \wedge \odot(T > w + \delta_a) \Rightarrow (nval(Z) = (nval(x) + nval(y)) \bmod 2^n)))$$

In the above formula, x and y are n-bit vectors and $nval$ converts a binary number to a decimal one.

An adder with a carry-in bit C_i and a carry-out bit C_o is described as :

$$\forall u, w, x, y, c.\Box(((T = u) \wedge (X = x) \wedge (Y = y) \wedge C_i = c \wedge (u \leq w < \odot T)) \Rightarrow \Box((T \leq w + \delta_a) \wedge \odot(T > w + \delta_a) \Rightarrow (nval(< C_o > \hat{} Z) = nval(x) + nval(y) + c)))$$

An adder which requires inputs to be stable for δ_a time units is specified as follows :

$$\forall u,w,x,y,c.\Box(((T=u)\wedge(u\leq w<\odot T)$$
$$\wedge((X=x\wedge Y=y\wedge C_i=c)\mathcal{U}^+((T\leq w+\delta_a)\wedge\odot(T>w+\delta_a))))$$
$$\Rightarrow\Box((T\leq w+\delta_a)\wedge\odot(T>w+\delta_a)$$
$$\Rightarrow(nval(<C_o>\hat{}Z)=nval(x)+nval(y)+c)))$$

Latch

A latch is a memory device used for storing a single bit of data. It has two inputs S and R and two outputs Q and $\overline{Q}$ which are complements of each other. If S is equal to 0 and R is equal to 1 and the inputs have been stable for at least δ_l time units, then Q is equal to 0 and $\overline{Q}$ is equal to 1 after a delay of δ_l time units. If S is equal to 1 and R is equal to 0 then Q is equal to 1 and $\overline{Q}$ is equal to 0 after a delay of δ_l time units. The latch is specified as follows:

$$Latch \triangleq \forall u,x.Reset\wedge Set$$

where *Reset* and *Set* are defined as :

$$Reset \triangleq \Box(((T=u)\wedge(u\leq x<\odot T)$$
$$\wedge((S=0)\wedge(R=1))\mathcal{U}^+((T\leq x+\delta_l)\wedge((\odot T)>x+\delta_l)))$$
$$\Rightarrow\Box((T\leq x+\delta_l)\wedge\odot(T>x+\delta_l)\Rightarrow((Q=0)\wedge(\overline{Q}=1))))$$

$$Set \triangleq \Box(((T=u)\wedge(u\leq x<\odot T)$$
$$\wedge((S=1\wedge R=0))\mathcal{U}^+((T\leq x+\delta_l)\wedge((\odot T)>x+\delta_l)))$$
$$\Rightarrow\Box((T\leq x+\delta_l)\wedge\odot(T>x+\delta_l)\Rightarrow((Q=1)\wedge(\overline{Q}=0))))$$

5.2 Delay-insensitive Circuits

Consider the oscillating circuit illustrated in Figure 5.4 constructed using an inverting C-gate, an invertor and a delay wire. Its behaviour is described by a conjunction of the formulas describing the behaviours of its components.

$$OC \triangleq G\wedge I\wedge W$$

The oscillator can be replaced by a new circuit in which G is replaced by G' and X and Z are connected by an "infinitely fast" wire provided the following conditions hold [HCS92].

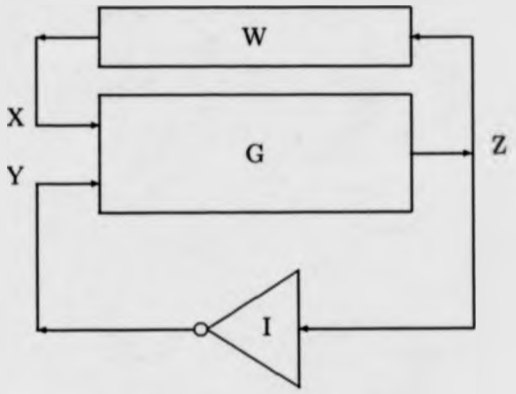

Figure 5.4: An oscillator

1. The delay associated with the wire is less than that of the invertor, i.e. $\delta_w < \delta_i$
2. The delay associated with the invertor is less than that of the inverting C-gate, i.e. $\delta_i < \delta_g$
3. $init \triangleq \Box((0 \leq T < \delta_w \Rightarrow \neg X) \wedge (0 \leq T < \delta_i \Rightarrow Y) \wedge (0 \leq T < \delta_g \Rightarrow Z))$ holds.

The new circuit is shown in Figure 5.5. The behaviour of G' is described as follows :

$$\begin{aligned} G' \triangleq \forall u, x, c. \Box((((T = u) \wedge \neg(Z \Leftrightarrow Y) \wedge (Y = c) \wedge (u \leq x < \odot T)) \\ \Rightarrow \Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)) \\ \wedge(((T = u) \wedge (Z \Leftrightarrow Y) \wedge (Z = c) \wedge (u \leq x < \odot T)) \\ \Rightarrow \Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c))) \end{aligned}$$

If the value of Z is different from that of Y then after a delay of δ_g time units Z takes the current value of Y or else it retains its value.

The transformed circuit OC' is described as :

$$OC' \triangleq G' \wedge I$$

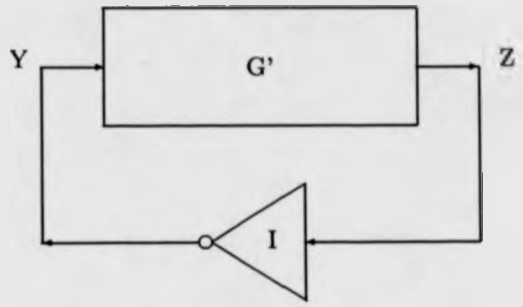

Figure 5.5: A transformed oscillator

We now prove the following theorem to show that OC' is a correct transformation under the initialisation and the timing assumptions.

Theorem 5.2.1 (Correctness of Circuit Transformation)

$$\vdash (OC \wedge \delta_w < \delta_i < \delta_g \wedge init) \Rightarrow OC'$$

To prove Theorem 5.2.1 we prove

$$\vdash (OC \wedge \delta_w < \delta_i < \delta_g \wedge init) \Rightarrow \forall u, x, c. \Box G''$$

where $G'' \triangleq G''_1 \wedge G''_2$ and

$$G''_1 \triangleq ((T = u) \wedge (u \leq x < \odot T) \wedge \neg(Z \Leftrightarrow Y) \wedge (Y = c)) \\ \Rightarrow \Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)$$

and

$$G''_2 \triangleq ((T = u) \wedge (u \leq x < \odot T) \wedge (Z \Leftrightarrow Y) \wedge (Z = c)) \\ \Rightarrow \Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)$$

Proof :

To prove the theorem, we prove 1i and 1ii and apply the derived computation induction principle.

1i. $\vdash (OC \wedge \delta_w < \delta_i < \delta_g \wedge init) \Rightarrow G''$
1ii $\vdash (OC \wedge \delta_w < \delta_i < \delta_g \wedge init) \Rightarrow \odot(OC \wedge \delta_w < \delta_i < \delta_g \wedge init)$

The proof of 1ii is straightforward. To prove 1i we assume $OC \wedge \delta_w < \delta_i < \delta_g \wedge init$ holds and prove G_1'' and G_2''. First, we prove G_2''. We assume (2) and prove $\Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)$.

2. $OC \wedge \delta_w < \delta_i < \delta_g \wedge init \wedge (T = u) \wedge (u \leq x < \odot T)$
$\wedge (Z \Leftrightarrow Y) \wedge (Z = c)$

(3) follows from the assumption.

3. $\Box((T \leq x + \delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)$

This proves G_2''. To prove G_1'', we assume (4) and prove $\Box((T \leq x+\delta_g) \wedge \odot(T > x + \delta_g) \Rightarrow Z = c)$

4. $OC \wedge \delta_w < \delta_i < \delta_g \wedge init \wedge (T = u) \wedge (u \leq x < \odot T)$
$\wedge \neg(Z \Leftrightarrow Y) \wedge (Y = c)$

The conclusion follows iff $\Box\varphi$ holds where $\varphi \triangleq \neg(Z \Leftrightarrow Y) \Rightarrow \neg(X \Leftrightarrow Y)$ holds. To prove this, we use the computation induction principle, i.e. we prove φ holds when $T = 0$ and whenever φ holds then $\odot\varphi$ also holds. φ holds at $T = 0$ follows from the assumptions. To prove that if φ holds then $\odot\varphi$ also holds we prove (5).

5. $((\neg(Z \Leftrightarrow Y) \Rightarrow \neg(X \Leftrightarrow Y)) \wedge \odot\neg(Z \Leftrightarrow Y)) \Rightarrow \odot(\neg(X \Leftrightarrow Y))$

(5) holds because when $(\neg(Z \Leftrightarrow Y) \Rightarrow \neg(X \Leftrightarrow Y)) \wedge \odot\neg(Z \Leftrightarrow Y)$ holds then $\odot\neg(X \Leftrightarrow Y)$ also holds if $\delta_w < \delta_i$ holds. This proves the theorem.

■

Note that the only timing assumption that is required to prove the theorem is $\delta_w < \delta_i$. The assumptions $\delta_w < \delta_g$ and $\delta_i < \delta_g$ are not necessary.

5.3 Discussion

In this chapter, we specified some digital circuits using the logic defined in Chapter 3. We further specified a delay-insensitive oscillator. The oscillator was transformed by replacing a wire in it which has a delay by one without a delay. It was then proved that the transformed circuit was a valid simplification of the original circuit under some timing and initial conditions using a computation induction principle.

Different variants of temporal logic have been used in specifying digital circuits. Malachi and Owicki [MO81], for example, use an untimed linear-time temporal logic to specify self-timed circuits. A similar logic is used by Bochmann [Boc82] to specify and verify properties of an arbiter, a device for controlling access to shared resources. An interval temporal logic with discrete time is used in [HMM83] to specify a variety of circuits including delay circuits, adders, latches, etc. some of which are described in this chapter. A wire with a delay of δ_w time units (Figure 5.1) is specified in it as follows :

$$\Box((len = \delta_w) \Rightarrow (X \rightarrow Y))$$

where *len* returns the duration of an interval and $X \rightarrow Y$ holds for an interval iff the value of X at the beginning of the interval is equal to the value of Y at the end of the interval.

An extension of the interval temporal logic with Duration Calculus [HCS92] is used for specifying and verifying real-time properties of circuits. A wire with a delay of δ_w time units shown in Figure 5.1 is specified in it as follows :

$$\Box(\lceil X = c\rceil\hat{}\lceil\delta_w\rceil \Rightarrow \lceil\delta_w\rceil\hat{}\lceil Y = c\rceil)$$

The formula asserts that if a time interval can be partitioned into two subintervals of which $X = c$ holds throughout the first and the duration of the second subinterval is δ_w time units then it can also be partitioned into two subintervals such that the duration of the first subinterval is δ_w time units and $Y = c$ holds throughout the second subinterval. $\lceil\delta_w\rceil$ is used in the formula as a shorthand for $\int 1 = \delta_w$. Note that the formula $\lceil X = c\rceil\hat{}\lceil\delta_w\rceil \Rightarrow \lceil\delta_w\rceil\hat{}\lceil Y = c\rceil$ holds trivially for an interval which is exactly δ_w time units irrespective of whether $X = c$ holds at the beginning of the interval and $Y = c$ holds at the end of the interval or not. This is because $\lceil X = c\rceil$ is only defined for a time interval greater than 0 time units, i.e it is not defined for a point interval. Therefore, an interval in which $\lceil X = c\rceil\hat{}\lceil\delta_w\rceil$ holds must be greater than δ_w time units.

A detailed discussion on delay-insensitive circuits and the proof that the oscillator shown in Figure 5.5 is a correct transformation of the one shown in Figure 5.4 under the initialisation and the timing assumptions using the Duration Calculus is also given in [HCS92].

Chapter 6

Proof System for a Real-Time Language

In this chapter, we present a compositional proof system in which the specification and the implementation languages are different. Such a proof system typically assumes

1. a particular model of computation,
2. a mathematical model of time,
3. a specification language,
4. an implementation language,
5. and gives algorithms or proof rules for reasoning about safety and liveness properties [HMP91b].

Our motivation for developing a proof system is two-fold. Firstly, as an application of the specification language developed in Chapter 3 and secondly, to develop a general proof system independent of the implementation issues such as the type of communication, resource constraints, etc. The proof system is also independent of the time domain that is used. We follow the following general outline to describe the proof system. We present a description and the syntax of the programming language in Section 6.1. A computation model and a specification-oriented semantics for the language is then given in Section 6.2. We define a correctness relation **sat** which relates the implementation and the specification languages. A proof system is then presented in Section 6.3. This is followed by examples of the specification of process scheduling and the application of the proof system to two examples - a watchdog timer network and

Variables	*var*	::=	$x \mid x_1 \mid x_2 \mid \ldots \mid x_m$
Channels	*chan*	::=	$c \mid c_1 \mid c_2 \mid \ldots \mid c_n$
Expression	e	::=	$v \mid var \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	b	::=	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Statement	S	::=	$\mathbf{skip} \mid var := e \mid chan?var \mid chan!e$ $\mid \mathbf{delay}\ e \mid GC \mid \star GC \mid S_1; S_2$
Guarded Command	GC	::=	$[]_{i=1}^{m} b_i \rightarrow S_i$ $\mid []_{i=1}^{m} b_i; chan_i?var_i \rightarrow S_i [] b; \mathbf{delay}\ e \rightarrow S$
Network	N	::=	$S \mid N_1 \parallel N_2$

Table 6.1: Syntax of the programming language

a stop-and-wait protocol. As an example, we consider a CSP-like language with synchronous communication.

6.1 Programming Language

6.1.1 Syntax

The programming language we consider is an extension of a CSP-like language (c.f.[Hoa78]). CSP is a language for describing processes which are distributed and communicate by synchronous message passing on unidirectional channels. A primitive process is one of the following : *skip*, *assignment*, *input*, *output*, and *delay*. These processes are primitive in the sense that they are indivisible. Input and output are synchronous, i.e. they take place simultaneously. Primitive processes can be combined to form more complex processes using one of the following combinators: guarded command, iteration, and sequential composition. Processes can be combined using parallel composition to form a network.

Let VAR be a nonempty set of program variables, $CHAN$ a nonempty set of channels, and VAL be a domain of values consisting of integers, reals etc. Let $m \in \mathbb{N}$, $n \in \mathbb{N}$ such that $1 \leq m$ and $1 \leq n$ with $x, x_1, \ldots, x_m \in VAR$, $c, c_1, \ldots, c_n \in CHAN$ and $v \in VAL$. The syntax of the programming language is described in Table 6.1 using BNF with some informal description of the meaning of each statement given below. In the description, we use the terms 'process', 'statement' and 'command' interchangeably.

- **skip** is the null statement. The execution of a skip command leaves the values of the program variables unchanged.
- $x := e$ is an assignment statement where x is a program variable and e is an expression. The execution of an assignment statement updates the value of the variable x with the value of the expression e.
- $c?x$ is an input statement. It is used for receiving a value on a channel c and storing it in the variable x. If the corresponding sending process is not ready to send, then the communication is blocked.
- $c!e$ is an output statement. It is used for sending a value of an expression e on a channel c. The communication is synchronous; therefore if the corresponding receiving process is not ready to receive, then the communication is blocked.
- **delay** e is a delay statement. The execution of the delay statement terminates after e time units and leaves the values of the program variables unchanged. If the value of e is less than some value k_e (time required to evaluate e), the delay statement terminates immediately.
- The execution of the guarded command ($[]_{i=1}^{m} b_i \rightarrow S_i$) tests the boolean guards (i.e. b_i, $1 \leq i \leq m$) and selects the statement corresponding to the guard which has evaluated to true for execution. If two or more guards evaluate to true then the choice is nondeterministic. The statement terminates if none of the guards evaluate to true. The guarded command ($[]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [] b;$ **delay** $e \rightarrow S$) is executed as follows. The boolean parts of the guards (i.e. b_i, $1 \leq i \leq m$ and b) are evaluated. If none of them evaluate to true then the command terminates. Otherwise, the input statement of one of the guards which has evaluated to true is executed and the corresponding statement is selected for execution. If the boolean part of the delay guard also evaluates to true and none of the input statements is executed within e time units then S is executed. If b evaluates to true and the value of e is less than some positive value k_e (time required to evaluate e) then S is executed.
- $\star GC$ is an iteration command where $GC \triangleq []_{i=1}^{m} b_i \rightarrow S_i | []_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [] b;$ **delay** $e \rightarrow S$. GC is executed repeatedly until none of the boolean parts of the guards evaluate to true.
- $S_1; S_2$ is sequential composition of two statements S_1 and S_2. S_1 is executed and if it terminates then S_2 is executed.
- $N_1 \parallel N_2$ is parallel composition of two network of processes N_1 and N_2. Both N_1 and N_2 are executed in parallel.

In the rest of the chapter, we will assume that a network process (i.e. a network which is also a process) is labelled using a name $p, p_1, \ldots, p_n \in Pname$ $(1 \leq n)$ where $Pname$ is a set of process names and processors are labelled using a processor name $q, q_1, \ldots, q_m \in Prname$ $(1 \leq m)$ where $Prname$ is a set of processor names.

6.1.2 Syntactic Restrictions

The statements described in the previous section can be combined to form complex statements using the combinators. However, not all statements are syntactically correct. We therefore state some restrictions on constructing syntactically well-formed statements.

Definition 6.1.1 (Channels Occurring in a Statement) Let $outc(S)$ and $inc(S)$ be defined such that $c \in outc(S)$ iff a channel c appears as an output channel in the statement S and $c \in inc(S)$ iff a channel c appears as an input channel in the statement S. ■

$outc(S)$ and $inc(S)$ are defined as follows where $\bigcup_{i=1}^{m} P_i$ is an abbreviation for $P_1 \cup P_2 \ldots \cup P_m$.

1. $outc(\textbf{skip}) = outc(x := e) = outc(c?x) = outc(\textbf{delay}\ e) = \emptyset$, $outc(c!e) = \{c\}$, $outc([\!]_{i=1}^{m} b_i \rightarrow S_i) = \bigcup_{i=1}^{m}(outc(S_i))$, $outc([\!]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [\!] b; \textbf{delay}\ e \rightarrow S) = \bigcup_{i=1}^{m} outc(S_i) \cup outc(S)$, $outc(\star GC) = outc(GC)$, $outc(S_1; S_2) = outc(S_1) \cup outc(S_2)$.

2. $inc(\textbf{skip}) = inc(x := e) = inc(c!x) = inc(\textbf{delay}\ e) = \emptyset$, $inc(c?e) = \{c\}$, $inc([\!]_{i=1}^{m} b_i \rightarrow S_i) = \bigcup_{i=1}^{m}(inc(S_i))$, $inc([\!]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [\!] b; \textbf{delay}\ e \rightarrow S) = \bigcup_{i=1}^{m} inc(S_i) \cup inc(S) \cup (\bigcup_{i=1}^{m}\{c_i\})$, $inc(\star GC) = inc(GC)$, $inc(S_1; S_2) = inc(S_1) \cup inc(S_2)$.

The following restrictions apply to the channels appearing in a statement.

- For a statement $S_1; S_2$, $inc(S_1) \cap outc(S_2) = \emptyset \wedge outc(S_1) \cap inc(S_2) = \emptyset$.

Definition 6.1.2 (Channels Occurring in a Network) Let $oc(N)$ and $ic(N)$ be defined such that $c \in oc(N)$ iff a channel c appears as an output channel in the network N and $c \in ic(N)$ iff a channel c appears as an input channel in the network N. ■

$oc(N)$ and $ic(N)$ are defined as follows :

1. $oc(S) = outc(S)$, $oc(N_1 \parallel N_2) = oc(N_1) \cup oc(N_2)$.
2. $ic(S) = inc(S)$, $ic(N_1 \parallel N_2) = ic(N_1) \cup ic(N_2)$.

The following restrictions apply to the channels appearing in a network.

- For a network $N_1 \parallel N_2$, $ic(N_1) \cap ic(N_2) = \emptyset \wedge oc(N_1) \cap oc(N_2) = \emptyset$.

Definition 6.1.3 (Variables Occurring in an Expression) Let $xvar(e)$ be defined such that $x \in xvar(e)$ iff x occurs in the expression e. ■

$xvar(e)$ is defined as follows:

1. For a variable x, $xvar(x) = \{x\}$.
2. For a value v, $xvar(v) = \emptyset$.
3. For an expression, $xvar(e_1 + e_2) = xvar(e_1 - e_2) = xvar(e_1 \times e_2) = xvar(e_1) \cup xvar(e_2)$.
4. For a boolean expression, $xvar(e_1 = e_2) = xvar(e_1 < e_2) = xvar(e_1) \cup xvar(e_2)$, $xvar(\neg b) = xvar(b)$ and $xvar(b_1 \vee b_2) = xvar(b_1) \cup xvar(b_2)$.

Definition 6.1.4 (Variables Occurring in a Statement) Let $var(S)$ be defined such that $x \in var(S)$ iff a variable x occurs in the statement S. ■

$var(S)$ is defined inductively on the grammar defined in Table 6.1 in terms of $xvar(e)$.

1. $var(\textbf{skip}) = \emptyset$, $var(x := e) = \{x\} \cup xvar(e)$, $var(c?x) = \{x\}$, $var(c!e) = var(\textbf{delay}\ \ e) = xvar(e)$, $var([]_{i=1}^{m} b_i \rightarrow S_i) = \bigcup_{i=1}^{m}(xvar(b_i) \cup var(S_i))$, $var([]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [] b; \textbf{delay}\ e \rightarrow S) = \bigcup_{i=1}^{m}(xvar(b_i) \cup \{x_i\} \cup var(S_i)) \cup xvar(b) \cup xvar(e) \cup var(S)$, $var(\star GC) = var(GC)$, $var(S_1; S_2) = var(S_1) \cup var(S_2)$.

Definition 6.1.5 (Variables Occurring in a Network) Let $nvar(N)$ be defined such that $x \in nvar(N)$ iff a variable x occurs in the network N. ■

$nvar(N)$ is defined as follows:

1. $nvar(S) = var(S)$, $nvar(N_1 \parallel N_2) = nvar(N_1) \cup nvar(N_2)$.

The following restrictions apply to the variables occurring in a network.

- For a network $N_1 \parallel N_2$, $nvar(N_1) \cap nvar(N_2) = \emptyset$.

That is, the sets of program variables of any two parallel network of processes are disjoint.

Definition 6.1.6 (Process Labels Occurring in a Network) Let $pnset(N)$ be defined such that $p \in pnset(N)$ iff a network labelled with p occurs in a network N. ■

$pnset(N)$ is defined inductively on the grammar defined in Table 6.1.

1. $pnset(p : S) = \{p\}$, $pnset(N_1 \parallel N_2) = pnset(N_1) \cup pnset(N_2)$.

The following restrictions apply to the process labels occurring in a network.

- For a network $N_1 \parallel N_2$, $pnset(N_1) \cap pnset(N_2) = \emptyset$.

That is, the sets of process labels of any two parallel network of processes are disjoint.

6.2 Denotational Semantics

6.2.1 Computational Model

The basic computational model is described in Section 3.2. In addition, we use the following variables to describe the behaviour of real-time systems.

We assume that $c \in CHAN$, $x \in VAR$, $p \in Pname$ and $q \in Prname$.

$q.p.c?$	:	a network p on a processor q is waiting to receive a value on a channel c.
$q.p.c!$	:	a network p on a processor q is waiting to send a value on a channel c.
$q.p.c.a$	:	a network p on a processor q is sending or receiving a value a on a channel c.
$q.p.?x$	:	a network p on a processor q accesses a variable x.
$q.p.rdy$	:	a network p is ready to execute on a processor q.
$q.p.run$	:	a network p is running on a processor q.
$q.p.done$	:	a network p has finished executing on a processor q.

The behaviours we consider satisfy the following nonlogical axioms.

1. $q.p.c? \Rightarrow q.p.rdy \wedge \neg q.p.c!$
2. $q.p.c! \Rightarrow q.p.rdy \wedge \neg q.p.c?$
3. $q.p.c.a \Rightarrow q.p.run$
4. $q.p.?x \Rightarrow q.p.run$
5. $q_{i_1}.p_{i_2}.c.a \wedge q_{j_1}.p_{j_2}.c.b \Rightarrow a = b$
6. $q_{i_1}.p_{i_2}.?x \wedge q_{j_1}.p_{j_2}.?x \Rightarrow i_1 = j_1 \wedge i_2 = j_2$.
7. $q.p.rdy \Rightarrow \neg q.p.run \wedge \neg q.p.done$
8. $q.p.run \Rightarrow \neg q.p.rdy \wedge \neg q.p.done$
9. $q.p.done \Rightarrow \neg q.p.rdy \wedge \neg q.p.run \wedge \Box q.p.done$

Axiom 1 states that a network which is waiting to receive is also ready to run; it cannot at the same time be waiting to send. Axiom 2 is similar to Axiom

1 for a network which is waiting to send. Axiom 3 says that a network which is sending or receiving a value is also running. Axiom 4 is similar to Axiom 3 for a network accessing a variable. Axiom 5 states that only one value can be transmitted on any channel at any time. Axiom 6 states that only one network can access any variable at any time. Axiom 7 says that a network which is ready to run is not running and has not finished running. If a network is running then it is not ready to run and has not finished running(Axiom 8). Axiom 9 states that if a network has finished running then it is not waiting to run, and it is not running and it is always the case that it has finished running in the future.

Note that Axiom 6 is not required for the programming language described in the previous section because shared variables are ruled out by syntactic restrictions. It has been included so that the proof system developed in this chapter can also be applied to a language with shared variables.

6.2.2 Extension to the Specification Language

In this section, we extend the specification language with two operators - *chop* and *iterated chop*. These operators are required for describing sequential composition and iteration of processes.

First, we define the concatenation of valid behaviours. Recall that a behaviour was defined in Chapter 3 as a mapping from Γ to $STATE$ where Γ is a time domain and $STATE$ is a set of mappings from the set of local variables to the set of values. A valid behaviour satisfies the finite variability condition, i.e. there cannot be infinitely many state changes in a finite interval of time.

Definition 6.2.1 (Concatenation of Valid Behaviours) Let σ_1 and σ_2 be valid behaviours. Consider the least time point τ such that for all finite sets $qset \subseteq Prname$, $pset \subseteq Pname$ and $xset \subseteq VAR$

$$\forall \tau'.(\tau \leq \tau' \Rightarrow \sigma_1(\tau) = \sigma_1(\tau')) \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset} \sigma_1(\tau)(q.p.done))$$
$$\wedge \bigwedge_{x \in xset} \sigma_1(\tau)(x) = \sigma_2(0)(x)$$

Then σ is defined as $\sigma_1\sigma_2$ (concatenation) iff

$$\forall \tau'.(\tau' < \tau \Rightarrow \sigma(\tau') = \sigma_1(\tau')) \wedge \forall \tau'.(\tau \leq \tau' \Rightarrow \sigma(\tau') = \sigma_2(\tau' - \tau))$$

If no such least time point exists, then $\sigma = \sigma_1$.

■

The formation rules introduced for terms and formulae introduced in Chapter 3 remain the same, but the following rule is added.

Formation Rules

- Formulae

  1. If φ and ψ are formulae then so are $\varphi \mathbf{C} \psi$ and $\varphi \mathbf{C}^* \psi$.

We now give the semantics of *chop* and *iterated chop*.

- Chop

  $\varphi \mathbf{C} \psi|_i^\sigma = \mathbf{tt}$ iff for some σ_1 and $\sigma_2, \varphi|_i^{\sigma_1} = \mathbf{tt}$ and $\psi|_0^{\sigma_2} = \mathbf{tt}$ and $\sigma_1 \sigma_2 = \sigma$

  $\varphi \mathbf{C} \psi$ holds for a behaviour σ iff σ can be partitioned into two sub-behaviours σ_1 and σ_2 such that σ_1 satisfies φ, σ_2 satisfies ψ and they can be concatenated to form σ.

- Iterated chop

  $\varphi \mathbf{C}^* \psi|_i^\sigma = \mathbf{tt}$ iff for some $\sigma_1, \sigma_2, \ldots, \sigma_k, \varphi|_i^{\sigma_1} = \mathbf{tt}$ and for every j, $1 < j < k, \varphi|_0^{\sigma_j} = \mathbf{tt}$ and $\psi|_0^{\sigma_k} = \mathbf{tt}$ and $\sigma_1 \sigma_2 \ldots \sigma_k = \sigma$
  or for some $\sigma_1, \sigma_2, \ldots, \varphi|_i^{\sigma_1} = \mathbf{tt}$ and for every j, $1 < j, \varphi|_0^{\sigma_j} = \mathbf{tt}$ and $\sigma_1 \sigma_2 \ldots = \sigma$

  $\varphi \mathbf{C}^* \psi$ holds for a behaviour σ iff σ can be partitioned into sub-behaviours $\sigma_1, \sigma_2, \ldots, \sigma_k$ such that ψ holds for the last sub-behaviour, and φ holds for all preceding ones and they can be concatenated to form σ, or σ can be partitioned into an infinite number of sub-behaviours each of which satisfies φ and they can be concatenated to form σ.

6.2.3 Temporal Semantics

The semantics of a network N is given by a semantic function $\mathcal{M} : [\![N]\!] \rightarrow \mathcal{F}$ where $\mathcal{F}$ is the set of all temporal formulas over valid real-time behaviours.

We use the following abbreviations to make the semantic definitions a little more presentable. In the abbreviations and in the rest of the chapter we use $\bigwedge_{i=1}^{n} \varphi_i$ or $\bigwedge_{i \in \{1..n\}} \varphi_i$ as an abbreviation for a finite conjunction of formulas, i.e.

$$\bigwedge_{i=1}^{n} \varphi_i \triangleq \bigwedge_{i\in\{1..n\}} \varphi_i \triangleq \varphi_1 \wedge \varphi_2 \ldots \wedge \varphi_n$$

Similarly, $\bigvee_{i=1}^{n} \varphi_i$ or $\bigvee_{i\in\{1..n\}} \varphi_i$ is an abbreviation for a finite disjunction of formulas, i.e.

$$\bigvee_{i=1}^{n} \varphi_i \triangleq \bigvee_{i\in\{1..n\}} \varphi_i \triangleq \varphi_1 \vee \varphi_2 \ldots \vee \varphi_n$$

We also use $\exists\, x : \varphi$ as an alternative syntax for $\exists\, x.\varphi$.

$$\begin{aligned}
&no_inact(qset, pset, cset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset}(\bigwedge_{c\in cset} \neg q.p.c?))\\
&no_outact(qset, pset, cset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset}(\bigwedge_{c\in cset} \neg q.p.c!))\\
&no_comact(qset, pset, cset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset}(\bigwedge_{c\in cset} \neg \exists\, a : q.p.c.a))\\
&no_rdy(qset, pset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset} \neg q.p.rdy)\\
&no_run(qset, pset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset} \neg q.p.run)\\
&no_done(qset, pset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset} \neg q.p.done)\\
&no_vacc(qset, pset, xset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset}(\bigwedge_{x\in xset} \neg q.p.?x))\\
&vacc(qset, pset, xset) \triangleq \bigwedge_{q\in qset}(\bigwedge_{p\in pset}(\bigwedge_{x\in xset} q.p.?x))
\end{aligned}$$

From now on, we will use the abbreviations Pn for $Pname$, Pr for $Prname$ and $\mathcal{M}[\![S]\!]_p$ for $\mathcal{M}[\![p : S]\!]$.

First, we state two properties which are common to all statements. The first is that a program does not wait to communicate or communicate on any channels not occurring in it. For all finite sets $qset \subseteq Pr$ and $cset \subseteq CHAN$

$$\begin{aligned}
\mathcal{M}[\![S]\!]_p \triangleq \Box(&no_comact(qset, \{p\}, cset) \wedge no_inact(qset, \{p\}, cset)\\
&\wedge no_outact(qset, \{p\}, cset))
\end{aligned}$$

provided $cset \cap inc(S) = \emptyset$ and $cset \cap outc(S) = \emptyset$.

The second property states that a program does not access any variables not occurring in it. For all finite sets $qset \subseteq Pr$ and $xset \subseteq VAR$

$$\mathcal{M}[\![S]\!]_p \triangleq \Box no\_vacc(qset, \{p\}, xset)$$

provided $xset \cap var(S) = \emptyset$.

In the semantic definitions we use the following bounded temporal operators as abbreviations.

$$\varphi \mathcal{U}_{=k} \psi \triangleq (T = u) \Rightarrow \Box((T < u + k) \Rightarrow \varphi) \\ \wedge \Diamond((T \leq u + k) \wedge \odot(T > u + k) \wedge \psi)$$

$\varphi \mathcal{U}_{=k} \psi$ holds iff ψ holds k time units from now and until that instant φ holds.

$$\varphi \mathcal{U}_{<k} \psi \triangleq (T = u) \Rightarrow \varphi \mathcal{U}((T < u + k) \wedge \psi)$$

$\varphi \mathcal{U}_{<k} \psi$ holds iff ψ holds within k time units from now and φ holds until that instant.

Skip

A skip statement will initially wait for execution until it is run on one of the processors. If executed it will eventually terminate after k_s $(0 < k_s)$ time units.

$$\mathcal{M}[\![\textbf{skip}]\!]_p \triangleq (\bigwedge_{q \in Pr} q.p.rdy) \mathsf{U} \bigvee_{q \in Pr} (q.p.run \mathcal{U}_{=k_s} q.p.done)$$

Assignment

An assignment statement $x := e$ initially waits for execution. If executed it will eventually terminate after k_a $(0 < k_a)$ time units with x assigned the value of e.

$$\mathcal{M}[\![x := e]\!]_p \triangleq \\ (\bigwedge_{q \in Pr} q.p.rdy) \mathsf{U} \bigvee_{q \in Pr} \exists a.((e = a) \wedge (acc \mathcal{U}_{=k_a}((x = a) \wedge q.p.done)))$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(x := e))$.

Delay

The delay statement **delay** e is described by an initial waiting period followed by a period equal to k_s $(0 < k_s)$ time units during which it evaluates e. The processor runs for a further period equal to the value of $e - k_s$ time units. If the value of e is less than k_s the statement terminates immediately.

$$\mathcal{M}[\![\textbf{delay } e]\!]_p \triangleq \\ (\bigwedge_{q \in Pr} q.p.rdy) \mathsf{U} \bigvee_{q \in Pr} \exists a.(e = a \wedge (acc \wedge q.p.run) \mathcal{U}_{=max(a,k_s)} q.p.done)$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(\textbf{delay } e))$.

Output

An output statement $c!e$ is described by a behaviour with an initial waiting period when the communication is on offer and then by a period when the communication is taking place. The communication terminates after k_c ($0 < k_c$) time units.

$$\mathcal{M}[\![c!e]\!]_p \triangleq (\bigwedge_{q \in Pr} q.p.c!) \mathsf{U} \bigvee_{q \in Pr} \exists a.((e = a) \wedge (acc \wedge q.p.c.a)\mathcal{U}_{=k_c} q.p.done)$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(c!e))$.

Input

The input statement $c?x$ is described similarly. The final value of x is the value it receives by the way of communication.

$$\mathcal{M}[\![c?x]\!]_p \triangleq (\bigwedge_{q \in Pr} q.p.c?) \mathsf{U} \bigvee_{q \in Pr} \exists a.((acc \wedge q.p.c.a)\mathcal{U}_{=k_c}(x = a \wedge q.p.done))$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(c?x))$.

Sequential Composition

The meaning of sequential composition of two statements S_1 and S_2 is given by the set of behaviours which are formed by concatenating the behaviours of S_1 and S_2.

$$\mathcal{M}[\![S_1; S_2]\!]_p \triangleq \mathcal{M}[\![S_1]\!]_p \mathbf{C} \mathcal{M}[\![S_2]\!]_p$$

Guarded Command

Let b_{GC} be defined as :

$$b_{GC} \triangleq \begin{cases} \bigvee_{i=1}^{m} b_i & \text{if } GC \triangleq [\!]_{i=1}^{m} b_i \rightarrow S_i \\ \bigvee_{i=1}^{m} b_i \vee b & \text{if } GC \triangleq [\!]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [\!] b; \textbf{delay } e \rightarrow S \end{cases}$$

First, consider the guarded command without delay.

$$GC \triangleq [\!]_{i=1}^{m} b_i \rightarrow S_i$$

There are two possibilities: either none of the boolean guards evaluates to true in which case the guarded command terminates after k_g $(0 < k_g)$ time units (equal to the time required to evaluate the boolean guards), or one or more boolean guards evaluate to true and one of the commands for which the boolean guard evaluated to true is executed.

Let $bvar_{GC}$ be defined as :

$$bvar_{GC} \triangleq \begin{cases} \bigcup_{i=1}^{m} xvar(b_i) & \text{if } GC \triangleq [\!]_{i=1}^{m} b_i \rightarrow S_i \\ \bigcup_{i=1}^{m} xvar(b_i) \cup xvar(b) & \text{if } GC \triangleq [\!]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i [\!] b; \textbf{delay } e \rightarrow S \end{cases}$$

We also define $accb$ as $vacc(\{q\}, \{p\}, bvar_{GC})$.

Then, the semantics of the guarded command without delay is as follows :

$$\mathcal{M}[\![[\!]_{i=1}^{m} b_i \rightarrow S_i]\!]_p \triangleq \bigwedge_{q \in Pr}(q.p.rdy) \mathsf{U}((\neg b_{GC} \wedge (\bigvee_{q \in Pr} accb\mathcal{U}_{=k_g} q.p.done)) \vee (b_{GC} \wedge (\bigvee_{q \in Pr} accb\mathcal{U}_{=k_g} \bigvee_{i=1}^{m}(b_i \wedge \mathcal{M}[\![S_i]\!]_p))))$$

Next, we consider the guarded command with delay.

$$GC \triangleq [[\!]_{i=1}^{m} b_i; c_i?x_i \rightarrow S_i \; [\!] b; \textbf{delay } e \rightarrow S]$$

The semantics is defined by considering the following possibilities :

- None of the boolean parts of the guards evaluates to true and the guarded command terminates.
- At least one b_i evaluates to true. If b also evaluates to true then one of the input commands for which b_i is true must start executing within e time units or immediately if the value of e is less than k_e time units. If the execution of an input command is started within the required time then the corresponding S_i is executed; otherwise, S is executed. If b evaluates to false then there is no deadline by which the input commands must be performed.

This leads to the following semantics :

$$\mathcal{M}[\![[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]\!]b; \textbf{delay } e \rightarrow S]\!]_p \triangleq \bigwedge_{q \in P_r}(q.p.rdy)\mathcal{U}((\neg b_{GC} \wedge \bigvee_{q \in P_r}(accb\mathcal{U}_{=k_g} q.p.done)) \vee no\_tmout \vee tmout)$$

where *no_tmout* and *tmout* are defined as follows :

$$no\_tmout \triangleq b_{GC} \wedge \neg b \wedge (\bigvee_{q \in P_r} accb\mathcal{U}_{=k_g}(\bigvee_{i=1}^{n}(b_i \wedge \mathcal{M}[\![c_i?x_i]\!]_p \mathbf{C} \mathcal{M}[\![S_i]\!]_p)))$$

$$tmout \triangleq b_{GC} \wedge b \wedge (\bigvee_{q \in P_r} \exists a.(e = a \wedge (accb\mathcal{U}_{=k_g}((\bigvee_{i=1}^{m} b_i \wedge (q.p.c_i?\mathcal{U}_{<max(a,k_s)} q.p.run) \wedge \mathcal{M}[\![c_i?x_i]\!]_p \mathbf{C} \mathcal{M}[\![S_i]\!]_p) \vee (b \wedge \mathcal{M}[\![\textbf{delay } e]\!]_p \mathbf{C} \mathcal{M}[\![S]\!]_p)))))$$

Iteration

The meaning of an iteration command $\star GC$ is given by considering three possibilities:

- GC is executed a finite number of times and on the last iteration all the boolean parts of the guards evaluate to false and GC terminates.
- GC is executed a finite number of times and on the last iteration it does not terminate.
- GC is executed an infinite number of times, all of which terminate.

The three cases are described by using the *iterated chop* operator.

$$\mathcal{M}[\![\star GC]\!]_p \triangleq (((\bigwedge_{q \in P_r} q.p.rdy)\mathcal{U}(b_{GC} \wedge \bigvee_{q \in P_r} q.p.run)) \wedge \mathcal{M}[\![GC]\!]_p) \mathbf{C}^*(((\bigwedge_{q \in P_r} q.p.rdy)\mathcal{U}(\neg b_{GC} \wedge \bigvee_{q \in P_r} q.p.run)) \wedge \mathcal{M}[\![GC]\!]_p)$$

Parallel Composition with Synchronous Communication

The semantics of parallel composition is given as :

$$\mathcal{M}[\![[\!]_{i=1}^{n} p_i : S_i]\!] \triangleq \bigwedge_{i=1}^{n} \mathcal{M}[\![p_i : S_i]\!] \wedge Sync$$

where *Sync* asserts that the communication between processes is synchronous and is defined as :

$$Sync \triangleq \Box(q_{i_1}.p_{i_2}.c.a \Rightarrow \exists j_1, j_2.(i_1 \neq j_1 \wedge i_2 \neq j_2 \wedge q_{j_1}.p_{j_2}.c.a \wedge \forall k_1, k_2.(k_1 \neq j_1 \neq i_1 \wedge k_2 \neq j_2 \neq i_2 \Rightarrow \neg q_{k_1}.p_{k_2}.c.a)))$$

for $1 \leq i_2 \leq n$, $1 \leq j_2 \leq n$ and $1 \leq k_2 \leq n$.

6.3 The Temporal Proof System

Let N be a network, and φ a temporal formula. We define what it means for a network N to satisfy a temporal formula φ.

Definition 6.3.1 (Satisfaction) A network N satisfies a temporal formula φ iff $\models \mathcal{M}[N] \Rightarrow \varphi$. This fact is denoted by N **sat** φ. ∎

We now give a compositional proof system for the language defined in Section 6.1 using the satisfaction relation defined above.

We axiomatise the well-formedness properties.

Axiom 6.3.1 (Well-formedness)

$$N \textbf{ sat } WF$$

where $WF \triangleq \Box(Wf \wedge Comval \wedge Accval)$

Wf, $Comval$ and $Accval$ are axiom schemas applied to all finite sets $qset \subseteq Prname$, $pset \subseteq Pname$, $cset \subseteq CHAN$ and $xset \subseteq VAR$ defined as follows:

$$\begin{aligned} Wf \triangleq\ & \bigwedge_{q \in qset}(\bigwedge_{p \in pset}(\bigwedge_{c \in cset} q.p.c? \Rightarrow q.p.rdy \wedge \neg q.p.c!)) \\ & \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset}(\bigwedge_{c \in cset} q.p.c! \Rightarrow q.p.rdy \wedge \neg q.p.c?)) \\ & \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset}(\bigwedge_{c \in cset} q.p.c.a \Rightarrow q.p.run)) \\ & \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset}(\bigwedge_{x \in xset} q.p.?x \Rightarrow q.p.run)) \\ & \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset} q.p.rdy \Rightarrow \neg q.p.run \wedge \neg q.p.done) \\ & \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset} q.p.run \Rightarrow \neg q.p.rdy \wedge \neg q.p.done) \\ & \wedge \bigwedge_{q \in qset}(\bigwedge_{p \in pset} q.p.done \Rightarrow \neg q.p.rdy \wedge \neg q.p.run \wedge \Box q.p.done) \end{aligned}$$

$$Comval \triangleq \bigwedge_{q_{i_1}.q_{j_1} \in qset}(\bigwedge_{p_{i_2}.p_{j_2} \in pset}(\bigwedge_{c \in cset} q_{i_1}.p_{i_2}.c.a \wedge q_{j_1}.p_{j_2}.c.b \Rightarrow a = b))$$

$$Accval \triangleq \bigwedge_{q_{i_1}.q_{j_1} \in qset}(\bigwedge_{p_{i_2}.p_{j_2} \in pset}(\bigwedge_{x \in xset}(q_{i_1}.p_{i_2}.?x \wedge q_{j_1}.p_{j_2}.?x \Rightarrow i_1 = j_1 \wedge i_2 = j_2)))$$

The inference rules for consequence and conjunction are as follows:

Rule 6.3.1 (Rule of Consequence)

$$\frac{N \textbf{ sat } \varphi_1, \varphi_1 \Rightarrow \varphi_2}{N \textbf{ sat } \varphi_2}$$

Rule 6.3.2 (Rule of Conjunction)

$$\frac{N \textbf{ sat } \varphi_1, N \textbf{ sat } \varphi_2}{N \textbf{ sat } \varphi_1 \wedge \varphi_2}$$

The next axiom states that a process does not communicate or wait to communicate on any channels not occurring in it. For all finite sets $qset \subseteq Pr$ and $cset \subseteq CHAN$

Axiom 6.3.2 (Communication Invariance)

$$(p : S) \textbf{ sat } \Box(no\_comact(qset, \{p\}, cset) \wedge no\_inact(qset, \{p\}, cset) \wedge no\_outact(qset, \{p\}, cset))$$

provided $cset \cap inc(S) = \emptyset$ and $cset \cap outc(S) = \emptyset$.

The following axiom states that a process does not access any variables not occurring in it. For all finite sets $qset \subseteq Pr$ and $xset \subseteq VAR$

Axiom 6.3.3 (Variable Invariance)

$$(p : S) \textbf{ sat } \Box no\_vacc(qset, \{p\}, xset)$$

provided $xset \cap var(S) = \emptyset$.

We now give the axioms for the primitive statements.

The skip axiom states a **skip** statement is ready to be executed until it is executed. If executed it terminates after k_s $(0 < k_s)$ time units.

Axiom 6.3.4 (Skip)

$$(p : \textbf{skip}) \quad \textbf{sat} \quad (\bigwedge_{q \in Pr} q.p.rdy) \mathcal{U} \bigvee_{q \in Pr} (q.p.run \mathcal{U}_{=k_s} q.p.done)$$

The assignment axiom states that an execution of an assignment statement $x := e$ if executed will eventually terminate after k_a $(0 < k_a)$ time units with x having the value of e.

Axiom 6.3.5 (Assignment)

$$(p : x := e) \quad \textbf{sat} \quad (\bigwedge_{q \in Pr} q.p.rdy) \mathcal{U} \bigvee_{q \in Pr} \exists a.((e = a) \wedge (acc \mathcal{U}_{=k_a}((x = a) \wedge q.p.done)))$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(x := e))$.

The delay axiom states a delay statement **delay** e if executed will terminate after e or k_e $(0 < k_e)$ time units, whichever is greater.

Axiom 6.3.6 (Delay)

$$(p : \textbf{delay } e) \quad \textbf{sat} \quad (\bigwedge_{q \in Pr} q.p.rdy) \mathcal{U} \bigvee_{q \in Pr} \exists a.((e = a) \wedge (acc \wedge q.p.run) \mathcal{U}_{=max(a,k_e)} q.p.done)$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(\textbf{delay } e))$.

The output axiom states that an output statement $c!e$ if executed sends the value of e and terminates after k_c $(0 < k_c)$ time units.

Axiom 6.3.7 (Output)

$$(p : c!e) \quad \textbf{sat} \quad (\bigwedge_{q \in Pr} q.p.c!) \mathcal{U} \bigvee_{q \in Pr} \exists a.(e = a \wedge (acc \wedge q.p.c.a) \mathcal{U}_{=k_c} q.p.done)$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(c!e))$.

The input axiom states that an execution of an input statement $c?x$ receives a value which is stored in x and terminates after k_c $(0 < k_c)$ time units.

Axiom 6.3.8 (Input)

$$(p : c?x) \ \textbf{sat} \ (\bigwedge_{q \in P_r} q.p.c?) \mathcal{U} \bigvee_{q \in P_r} \exists a.((acc \wedge q.p.c.a) \mathcal{U}_{=k_c}((x = a) \wedge q.p.done))$$

where $acc \triangleq vacc(\{q\}, \{p\}, var(c?x))$.

We now consider the complex statements.

The rule for sequential composition is

Rule 6.3.3 (Sequential Composition)

$$\frac{(p : S_1) \ \textbf{sat} \ \varphi_1, (p : S_2) \ \textbf{sat} \ \varphi_2}{(p : S_1; S_2) \ \textbf{sat} \ \varphi_1 \mathbf{C} \varphi_2}$$

Rule 6.3.4 (Guarded Command without Delay)

$$\frac{(p : S_i) \ \textbf{sat} \ \varphi_i \ \textit{for} \ i = 1, \ldots, m}{(p : [\!]_{i=1}^m b_i \rightarrow S_i) \ \textbf{sat} \ gc}$$

where gc is defined as :

$$gc \triangleq \bigwedge_{q \in P_r} (q.p.rdy) \mathcal{U}((\neg b_{GC} \wedge (\bigvee_{q \in P_r} accb \mathcal{U}_{=k_g} q.p.done)) \vee (b_{GC} \wedge (\bigvee_{q \in P_r} accb \mathcal{U}_{=k_g} \bigvee_{i=1}^m (b_i \wedge \varphi_i))))$$

and $accb$ is defined in the previous section.

Rule 6.3.5 (Guarded Command with Delay)

$$\frac{(p : \textbf{delay} \ e; S) \ \textbf{sat} \ \varphi, (p : c_i?x_i; S_i) \ \textbf{sat} \ \varphi_i \ \textit{for} \ i = 1, \ldots, m}{p : [\!]_{i=1}^m b_i; c_i?x_i \rightarrow S_i [\!] b; \textbf{delay} \ e \rightarrow S \ \textbf{sat} \ gcd}$$

where *gcd* is defined as :

$$\begin{aligned} gcd \triangleq \bigwedge_{q \in P_r}(q.p.rdy)\mathsf{U}((\neg b_{GC} \wedge \bigvee_{q \in P_r}(accb\mathcal{U}_{=k_s} q.p.done)) \\ \vee(b_{GC} \wedge \neg b \wedge (\bigvee_{q \in P_r} accb\mathcal{U}_{=k_s}(\bigvee_{i=1}^{m}(b_i \wedge \varphi_i)))) \\ \vee(b_{GC} \wedge b \wedge (\bigvee_{q \in P_r} \exists a.(e = a \wedge (accb\mathcal{U}_{=k_s}((\bigvee_{i=1}^{m}(b_i \\ \wedge(q.p.c_i?\mathcal{U}_{<max(a,k_s)} q.p.run) \wedge \varphi_i) \vee (b \wedge \varphi)))))))) \end{aligned}$$

The inference rule for iteration is

Rule 6.3.6 (Iteration)

$$\frac{(p : GC) \ \ \textbf{sat} \ \ \varphi}{(p : \star GC) \ \ \textbf{sat} \ \ iter}$$

where *iter* is defined as :

$$\begin{aligned} iter \triangleq (((\bigwedge_{q \in P_r} q.p.rdy)\mathsf{U}(b_{GC} \wedge \bigvee_{q \in P_r} q.p.run)) \wedge \varphi) \\ \mathbf{C}^{\star}(((\bigwedge_{q \in P_r} q.p.rdy)\mathsf{U}(\neg b_{GC} \wedge \bigvee_{q \in P_r} q.p.\check{run})) \wedge \varphi) \end{aligned}$$

The inference rule for parallel composition is as follows:

Rule 6.3.7 (Parallel Composition with Synchronous Communication)

$$\frac{(p_i \triangleq S_i) \ \ \textbf{sat} \ \ \varphi_i \ \textit{for} \ i = 1, \ldots, n}{\|_{i=1}^{n} (p_i : S_i) \ \ \textbf{sat} \ \ \bigwedge_{i=1}^{n} \varphi_i \wedge Sync}$$

Sync is defined in the previous section.

We now consider the soundness and relative completeness of the proof system.

To prove soundness of the proof system, we have to prove the following theorem.

Theorem 6.3.1 (Soundness) For all networks N, if N **sat** φ then $\models \mathcal{M}[\![N]\!] \Rightarrow \varphi$.

Soundness of the axioms and the inference rules follow directly from the definition of semantics.

■

Next, we prove relative completeness of the proof system. That is, every valid specification can be derived in the proof system if any valid formula in the logic can be proved.

Theorem 6.3.2 (Relative Completeness) If the formula φ is a valid specification for a network N then N **sat** φ is provable in the proof system given in Section 6.3.

Proof :

Assume that φ is a valid specification of N. Then, by the definition of the semantics of a network, $\mathcal{M}[\![N]\!] \Rightarrow \varphi$ is semantically valid for all valid behaviours, i.e. it is a theorem. Since N **sat** $\mathcal{M}[\![N]\!]$ and $\mathcal{M}[\![N]\!] \Rightarrow \varphi$ is a theorem then by the rule of consequence N **sat** φ. This proves the theorem.

■

6.4 Process Scheduling

The proof system developed in the previous section separates the properties attributable to programs from those attributable to the implementation issues including scheduling. There are various scheduling properties one can specify. We specify some of them here.

Safety

Two network processes cannot run on the same processor at the same time. For all finite $qset \subseteq Prname$, $pset \subseteq Pname$

$$\Box \bigwedge_{q \in qset} \bigwedge_{p_i, p_j \in pset} (q.p_i.run \wedge q.p_j.run \Rightarrow (i = j))$$

A network process can only run and finish on one processor at any time.

$$\Box \bigwedge_{q_i, q_j \in qset} \bigwedge_{p \in pset} ((q_i.p.run \wedge q_j.p.run \Rightarrow (i = j)) \wedge (q_i.p.done \wedge q_j.p.done \Rightarrow (i = j)))$$

The following property states that a network process which is ready on a processor cannot be running or finished running on some other processor. It also states similar properties for a network process which is running or finished running.

$$\Box \bigwedge_{q_i,q_j \in qset} \bigwedge_{p \in pset}((q_i.p.rdy \Rightarrow (\neg q_j.p.run \wedge \neg q_j.p.done)) \\ \wedge(q_i.p.run \Rightarrow (\neg q_j.p.rdy \wedge \neg q_j.p.done)) \\ \wedge(q_i.p.done \Rightarrow (\neg q_j.p.rdy \wedge \neg q_j.p.run)))$$

Maximal Parallelism

Two conditions are required to specify maximal parallelism. Firstly, every network process has its own processor. Therefore, a network process can only wait to run iff its communication is blocked. Secondly, communication can only be blocked iff one of the processes is not ready to communicate.

For all finite $qset \subseteq Prname$, $pset \subseteq Pname$, and for some finite $cset \subseteq CHAN$

$$no\_pwait \triangleq \bigwedge_{q \in qset} \bigwedge_{p \in pset} q.p.rdy \Rightarrow \bigvee_{c \in cset}(q.p.c! \vee q.p.c?)$$

no_pwait states that if a network process is ready to execute then it must either be waiting to send or waiting to receive on some channel.

For all finite $qset \subseteq Prname$, $pset \subseteq Pname$, and $cset \subseteq CHAN$

$$no\_cwait \triangleq \bigwedge_{q_{i_1},q_{j_1} \in qset} \bigwedge_{p_{i_2},p_{j_2} \in pset}(\bigwedge_{c \in cset} \neg(q_{i_1}.p_{i_2}.c! \wedge q_{j_1}.p_{j_2}.c?))$$

no_cwait states that no pair of network processes can be waiting to send and receive on the same channel at any time.

Then, the condition for maximal parallelism is stated as :

$$\Box(no\_pwait \wedge no\_cwait)$$

First Ready First Run

The following property states that a network process which is ready to run first is executed first.

$$\Box(\bigwedge_{q\in qset}\bigwedge_{p_i,p_j\in pset,i\neq j}((q.p_i.rdy \wedge q.p_j.rdy)\mathcal{S}(q.p_i.rdy \wedge \neg q.p_j.rdy) \Rightarrow \neg(q.p_i.rdy\,\mathcal{U}(q.p_j.run \wedge q.p_i.rdy))))$$

Priority

A network process p_i with a higher priority to run than p_j $(i < j)$ is executed first is specified as :

$$\Box(\bigwedge_{q\in qset}\bigwedge_{p_i,p_j\in pset,i<j} q.p_i.rdy \wedge q.p_j.rdy \Rightarrow \neg(q.p_i.rdy\,\mathcal{U}(q.p_j.run \wedge q.p_i.rdy)))$$

Upper bound on ready time

The following specifies upper bound on time a network process waits for execution before it is run :

$$\Box(\bigwedge_{q\in qset}\bigwedge_{p\in pset} q.p.rdy \Rightarrow q.p.rdy\,\mathcal{U}_{<k}\neg q.p.rdy) \text{ for } k > 0$$

Upper bound on run time

The following specifies upper bound on time a network process can run on a processor.

$$\Box(\bigwedge_{q\in qset}\bigwedge_{p\in pset} q.p.run \Rightarrow q.p.run\,\mathcal{U}_{<k}(q.p.rdy \vee q.p.done)) \text{ for } k > 0$$

6.5 Example : A Watchdog Timer Network

To illustrate the use of the proof system, we consider the watchdog timer network shown in Figure 6.1. A slightly different version is described and specified in [HW89] using MTL. The network consists of n processes $P_1, \ldots, P_n$ and a 'watchdog' process W. The function of the watchdog process is to periodically check that the processes are active. This is achieved as follows. Processes P_i periodically (say 10 time units) send a reset signal on a channel c_i to the watchdog process. W assumes P_i is active iff P_i sends a signal within every 10 time units. If there is no reset signal from one of the processes, the watchdog timer sends an alarm signal on the channel al within k time units. For simplicity, we assume that each process has its own processor and therefore omit any references to processor names.

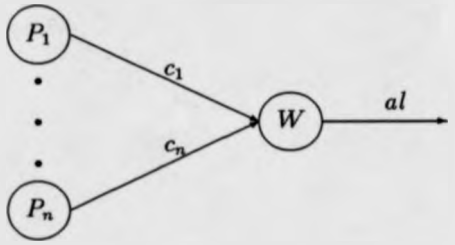

Figure 6.1: A watchdog timer network

First, we specify the watchdog process. We define *comm*, specifying that W has communicated with P_i on channel c_i within the last 10 time units.

$$comm \triangleq \bigwedge_{i=1}^{n}((T = u) \wedge (u \geq 10) \wedge (u \leq x < \odot T) \Rightarrow \Diamond(W.c_i \wedge \odot(T > x - 10)))$$

If no communication has taken place on any channel c_i within the last 10 time units then an error has occurred.

$$err \triangleq \bigvee_{i=1}^{n}((T = u) \wedge (u \geq 10) \wedge (u \leq x < \odot T) \Rightarrow \Box(\odot(T > x - 10) \Rightarrow \neg W.c_i))$$

The watchdog process then sends a signal on al within k time units.

$$alarm \triangleq ((T = u) \wedge (u \leq x < \odot T)) \Rightarrow \Diamond(W.al \wedge (T \leq x + k))$$

The W process is then specified as :

$$W \quad \textbf{sat} \quad (comm \wedge \neg W.al) \mathsf{U} (err \wedge alarm)$$

We specify P_i is active as follows :

$$P_i \quad \textbf{sat} \quad \Box(((T = u) \wedge (u \geq 10) \wedge (u \leq x < \odot T)) \Rightarrow \Diamond(P_i.c_i \wedge \odot(T > x - 10)))$$

That is, P_i sends a signal on c_i every 10 time units.

Given the specifications for P_i and W, we prove that if processes P_i $(1 \leq i \leq n)$ are active then W does not send any signal on al, i.e.

$$P_1 \parallel \ldots \parallel P_n \parallel W \quad \textbf{sat} \quad \Box\neg W.al$$

Proof :

(1) follows from parallel composition rule and the communication invariance axiom.

1. $\bigwedge_{i=1}^{n}(P_i.c_i \Leftrightarrow W.c_i)$

(2) follows from (1) and the specification of P_i.

2. $\Box((T = u) \wedge (u \geq 10) \wedge (u \leq x < \odot T) \Rightarrow \Diamond(W.c_i \wedge \odot(T > x - 10)))$

(3) follows from (2) and the definition of err.

3. $\Box\neg err$

(4) follows from (3) and the specification of W.

4. $\Box(comm \wedge \neg W.al)$

(5) follows from (4) and $\vdash \Box(\varphi \wedge \psi) \Rightarrow \Box\varphi \wedge \Box\psi$.

5. $\Box\neg W.al$

This proves the theorem. ■

6.6 Example : A Stop-and-Wait Protocol

In this section, we consider a more detailed example of the application of the proof system. We specify the requirements of a stop-and-wait protocol similar to the one described in [Sch90], and prove that its implementation meets the requirements.

The protocol consists of two processes P and Q, which communicate across two wires w_1 and w_2 as shown in Figure 6.2. The process P accepts an input message on the channel in and transmits it along w_1 within one time unit. Q accepts the message and within one time unit transmits it on its output channel out. Q also sends an acknowledgement to P on the channel w_2 within one time unit of outputting. P after receiving an acknowledgement accepts another input within one time unit. We assume that the wires are reliable and there is no delay in transmission along wires. The case where there is a delay is easy to incorporate by treating wires as processes.

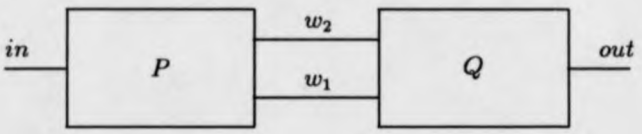

Figure 6.2: A stop-and-wait protocol

6.6.1 Abstract Requirements

The protocol must satisfy the following requirement : that if a message is input then the output must be ready within two time units. Formally, the protocol must satisfy the following specification :

$$Spec \triangleq \Box((T = u) \wedge P.in \Rightarrow \Diamond((T \leq u + 2) \wedge Q.out))$$

We give the specification of each component and prove that the protocol meets the above requirement.

The process P satisfies the following specification : it performs the events $P.in$, $P.w_1$ and $P.w_2$ in rotation; after accepting an input message on in channel it must output it on w_1 within one time unit, and after accepting an acknowledgement on w_2 it must accept another input within one time unit.

$$\begin{aligned} Spec_p \triangleq \Box(&((T = u) \wedge P.in \Rightarrow \Diamond((T \leq u + 1) \wedge P.w_1)) \\ &\wedge((T = u) \wedge P.w_2 \Rightarrow \Diamond((T \leq u + 1) \wedge P.in))) \end{aligned}$$

The receiving process Q will output the message it receives on w_1 within one time unit. It also sends an acknowledgement within one time unit of sending an output. The events $Q.w_1$, $Q.out$, $Q.w_2$ are therefore performed in strict rotation. Hence, the process Q satisfies the following specification :

$$\begin{aligned} Spec_q \triangleq \Box(&((T = u) \wedge Q.w_1 \Rightarrow \Diamond((T \leq u + 1) \wedge Q.out)) \\ &\wedge((T = u) \wedge Q.out \Rightarrow \Diamond((T \leq u + 1) \wedge Q.w_2))) \end{aligned}$$

The protocol is a combination of processes P and Q. We now prove that it satisfies the requirements.

$\vdash Spec_p \wedge Spec_q \Rightarrow Spec$

Proof :

We assume (1) and prove that $Spec$ holds.

1. $Spec_p \wedge Spec_q$

(2) follows from (1).

2. $\Box((T = u) \wedge P.in \Rightarrow \Diamond((T \leq u + 1) \wedge P.w_1))$

(3) follows from (1).

3. $\Box((T = u) \wedge Q.w_1 \Rightarrow \Diamond((T \leq u + 1) \wedge Q.out))$

(4) follows from (2), (3) and the parallel composition rule.

4. $\Box((T = u) \wedge P.in \Rightarrow \Diamond((T \leq u + 2) \wedge Q.out))$

This proves the theorem. ■

6.6.2 Implementation

We now move to the implementation stage. We propose implementations of P and Q and use the proof system to demonstrate that the implementations meet their specifications.

The protocol consists of two components P and Q communicating across the wires w_1 and w_2. The process P accepts an input and transmits it along w_1. After P has completed this transmission it waits for an acknowledgement on w_2 before repeating the cycle. This suggests the following implementation :

$$P \triangleq \star[true \rightarrow in?x; w_1!x; w_2?y]$$

The receiving process Q satisfies similar conditions. It accepts an input from w_1 and outputs it along out. After this transmission it sends an acknowledgement on w_2 before repeating the behaviour. The proposed implementation of Q is as follows :

$$Q \triangleq \star[true \rightarrow w_1?z; out!z; w_2!u]$$

We prove that P satisfies $Spec_p$ assuming the maximal parallelism model, i.e. each process has its own processor and there is no unnecessary waiting time. We also assume that there are no shared variables.

We apply proof rules for each subprocess. Thus,

$$(P : in?x) \textbf{ sat } spec_{p1}$$

where $spec_{p1}$ is defined as :

$$P.in?\mathsf{U}\,\exists a.(P.in.a\,\mathcal{U}_{=k_c}((x = a) \wedge P.done))$$

Similarly,

$$(P : w_1!x) \textbf{ sat } spec_{p2}$$

where $spec_{p2}$ is defined as :

$$P.w_1!\mathsf{U}\,\exists a.((x = a) \wedge P.w_1.a\,\mathcal{U}_{=k_c}P.done)$$

and

$$(P : w_2?y) \textbf{ sat } spec_{p3}$$

where $spec_{p3}$ is defined as :

$$P.w_2?\mathsf{U}\,\exists a.(P.w_2.a\,\mathcal{U}_{=k_c}((y = a) \wedge P.done))$$

Using the rule for sequential composition, we have

$$(P : in?x; w_1!x; w_2?y) \textbf{ sat } spec_{p1}\mathbf{C}spec_{p2}\mathbf{C}spec_{p3}$$

and using the rule for guarded command without delay

$$(P : [true \rightarrow in?x; w_1!x; w_2?y]) \textbf{ sat } pspec$$

where $pspec \triangleq P.rdy \mathsf{U} \Diamond_{=k_s}(spec_{p1} \mathbf{C} spec_{p2} \mathbf{C} spec_{p3})$.

The rule for an iteration command leads to

$$(P : \star[true \rightarrow in?x; w_1!x; w_2?y]) \ \textbf{sat} \ ((P.rdy \mathsf{U} P.run) \wedge pspec) \ \mathbf{C}^*(pspec \wedge \Box P.rdy)$$

For the maximal parallelism model, this leads to

$$(P : \star[true \rightarrow in?x; w_1!x; w_2?y]) \ \textbf{sat} \ (\Diamond_{=k_g}(spec_{p1} \mathbf{C} spec_{p2} \mathbf{C} spec_{p3})) \ \mathbf{C}^* false$$

To prove that the implementation of P satisfies its specification, we have to reason about *chop* and *iterated chop* operators for which we have not given an axiomatisation. We therefore give an informal argument. P executes $in?x$ in k_s time units and since there is no waiting time it executes $w_1!x$ within one time unit iff $k_s \leq 1$. It also executes $w_2?y$ in k_s time units and takes k_g time units to evaluate the guard before executing $in?x$ again. Therefore, P executes $in?x$ after executing $w_2?y$ within one time unit iff $k_g + k_s \leq 1$. This proves that the implementation of P satisfies the original specification iff $k_g + k_s \leq 1$. The proof that the implementation of Q meets its specification is similar.

6.7 Discussion

In this chapter, we developed a proof system for a CSP-like language using the specification language developed in Chapter 3. The proof system is quite general and is independent of the implementation issues such as maximal parallelism, shared processors, etc. The analysis given here can easily be extended to include other types of communication such as broadcast communication.

We have already mentioned some closely related work in the brief survey given in Chapter 1. In [Hoo91], proof systems based on MTL are developed. The computation model described is largely influenced by the work in [KSR+85] where a denotational semantics based on the maximal parallelism model of real-time systems is given. A model for a real-time system in [Hoo91] is given by a triple $(init, comm, final)$ where $init, final \in STATE$ and $comm$ is a function from time to a set of directed channels and pairs of a channel name and a value. For example, the proof rule for the output statement is written in it as:

$$c!e \textbf{ sat } wait(c!)\mathcal{U}(comm(c,e)\mathsf{U}_{=k_c} done)$$

where $wait(c!)$ asserts that the process is waiting to output on the channel c, $comm(c,e)$ asserts that it is sending the value of e on the channel c and $done$ asserts that the command has terminated.

A separate proof system for a shared processor implementation is also given in [Hoo91]. The computation model in it is a mapping from time to a set of triples $(comm, req, exec)$ where $comm$ is a subset of the set of directed channels, req is a set of priorities of the statements which are requesting execution and $exec$ is either an empty set or a singleton containing the priority of the statement which is currently being executed. The output statement using this model is written as:

$$c! \textbf{ sat } Req(\{c!,c\})\mathcal{U}(Execute(\{c!,c\},k_c)\mathbf{C}Send(c))$$

where $Req(\{c!,c\})$ asserts that the process $c!$ is requesting processor time, $Execute(\{c!,c\},k_c)$ asserts that the process is executed during k_c time units and $Send(c)$ asserts that the process is waiting for the corresponding receive operation until the communication takes place. Our model of computation is not specific to any implementation such as maximal parallelism or shared processors since we consider these to be implementation issues.

A proof system for shared processors which separates implementation issues and scheduling from program properties using Duration Calculus is given in [CHRR92]. The model of a real-time computation used is a variation of the one adopted in this chapter; however, no reference to processors is made in it since processes are considered to be distributed statically. The proof system only considers timing properties though it can be extended to reason about functional properties. The output statement in it is stated as :

$$\lceil\ \rceil \vee \lceil p.wait.c!\rceil \vee (\lceil p.wait.c!\rceil^\frown\lceil p.pass.c\rceil^\frown C)$$

The first disjunct describes the condition for a point interval and the second disjunct describes the condition when the communication is on offer. The last disjunct describes the condition when the communication is on offer followed by an interval in which the communication takes place. The duration formula C describes the behaviour after the output statement has been executed.

A compositional proof method using an explicit clock temporal logic based on the maximal parallelism model is given in [ZHK91]. The proof system is

similar to the one given for MTL in [Hoo91]. In [HMP91a] two proof systems for a transition language are given - one using a bounded-operator temporal logic and the other in an explicit clock temporal logic. Both proof systems are based on an interleaving semantics. An alternative proof system which uses TLA and not based on any implementation language is given in [AL91]. It is also based on an interleaving semantics.

Chapter 7

Conclusions and Discussion

7.1 Conclusions

In this thesis, we developed a formal notation applicable to the specification of the requirements, and the design and verification of real-time systems. We began in Chapter 1 with a brief survey of some of the formalisms used in the specification and verification of a real-time system. This was followed by a description of the requirements of a logic for real-time systems in Chapter 2. In Chapter 3, we developed a temporal logic ωTL to meet these requirements. We then defined a calculus of durations and the number of occurrences of predicates in the logic which can be used for describing high-level requirements of a real-time system. We also developed proof techniques for proving timing and functional properties of a system. In Chapter 6, we developed a proof system for a real-time extension of a CSP-like language which is independent of the implementation issues. Several varied examples including communication protocols and digital circuits were done in Chapters 3, 4, 5 and 6 to demonstrate the applicability of the logic and the proof techniques. However, to assess and compare the notation with others, we must consider its role in the development of a system. A system is initially described informally using a graphical or a semi-formal notation. These notations do not have a formal semantics and therefore cannot be used for proving the properties which the system must exhibit. Therefore, the informal requirements have to be translated to a formal specification. The formal notation used must exhibit two important properties:

- it must be expressive enough to specify a large class of real-time properties,

- and it should be possible to develop proof techniques for the verification of systems in it.

The logic formulated in Chapter 3 is an attempt to address these issues. If the intended behaviour of the system can be described in terms of the number of occurrences of predicates, then we may use the calculus to capture high-level system requirements and prove the desired properties in an abstract and simple fashion. The resulting specification can then be translated to the 'raw' logic used for the proof system. Alternatively, we may directly specify the requirements of the system in the 'raw' logic.

Once the system requirements have been formally specified and the desired properties proved, we can proceed with the design. An implementation language can be chosen to describe the system. The proof system introduced in Chapter 6 provides a link between the logic which is used for describing the behaviour of the system and the implementation language. We may propose an implementation of the system and use the axioms and the inference rules to prove that the system meets the requirements placed on it. The proof system is compositional and supports hierarchical development of programs.

7.2 Related Work

A wide variety of formal notations have been developed for the specification and verification of real-time systems, a brief survey of which is given in Chapter 1. However, there is comparatively little published work on formal requirements specification and proof techniques for real-time systems under a continuous model of time.

Process algebras (e.g. Timed CCS [MT90, Yi90], Timed ACP [BB90], ATP [NRSV89]) use a system description language to describe the implementation and its specification. Therefore, to show that an implementation meets its specification we describe both as processes and show that they exhibit the same behaviour according to their operational semantics. A notation that is used for the description of systems is generally not well-suited for specifying its requirements. For example, how would one describe that "the duration of gas leaks in any period must be less than or equal to one-twentieth of its duration provided the observation period is greater than or equal to sixty seconds". Process algebras in general have proved successful when dealing with systems which do not have any timing constraints. However, because of the complexity of real-time systems process languages do not seem to be appropriate for describing their requirements.

An alternative approach taken in Timed CSP [Sch90, Dav91] is to employ separate languages for system implementation and its specification. Specifications are expressed as predicates on the semantic domain of the implementation language. While, in the most general case, a specification may be any predicate over the semantic domain, it is usually restricted so that reasoning is easier.

Temporal logics [Ost89, Lam91, Koy89, CHR91] have widely been used as a specification language for real-time systems. An advantage of using a temporal logic over process algebras and Timed CSP is that it is independent of the system description language and therefore can be applied to a wide range of implementation languages. The main differences between the approach taken in this thesis and other temporal logics are as follows :

- The *next* operator in the logic is defined as the next time a change in state occurs or if no state change occurs then it is the time obtained by incrementing the current time by one. This allowed us to develop a logic with a calculus and proof techniques so that we can reason about real-time systems at various levels of abstraction and independently of the time domain that is used.
- The proof system developed in Chapter 6 separates implementation issues from program properties.

A detailed comparison between ωTL and other temporal logics is given in Chapter 3.

As we have seen from the examples in this thesis, the specification of the requirements and proving the desired properties of a system is far from a trivial task. The choice between different formal notations therefore rests on how easy it is to express the requirements of a system and prove properties in it. We have attempted to approach this by developing a logic with a calculus and proof techniques.

7.3 Future Work

There are two main areas of further work. The first is the logic itself and the other is the proof system. The logic presented in this thesis is very powerful. We developed a calculus in it which was shown through examples to be a useful tool for specifying high-level requirements and for proving properties of a real-time system. Therefore, an area for possible research would be to investigate ways of simplifying the logic while still retaining the calculus.

Another area of research is to extend the calculus to reason about probabilities. At present it cannot be used to express some properties such as *the probability of a gas leak occurring within a certain time is .7.* Therefore, it would be natural to extend the logic with a probabilistic calculus along the lines of [LRSZ92].

On the proof system side, we have used a very general computation model. This has resulted in a proof system which can be difficult to apply since it involves the use of complex operators such as *chop* and *iterated chop* for which we have not given an axiomatisation. In an interleaving model, only one action is analysed at any time and therefore it yields simpler proof rules to reason about programs. Therefore, various different proof systems and trade-offs in the context of the specification and verification of a real-time system in ωTL need further investigation.

Refinement from specification provides a useful way of constructing programs. Using this method, a high-level specification is transformed by a sequence of correctness preserving rules into an executable code [AL88, Bac89, BvW89]. While the method has successfully been applied to sequential and to some extent parallel programs, its application to real-time programs using ωTL is an area of further research.

Appendix A

Logic : Soundness and Theorems

A.1 Proofs of Soundness of the Logic

We prove soundness of the axioms of ωTL given in Section 3.3 and in Section 3.4.

F1. $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$

Proof :

By the semantic definition of *implication*, we have

1. $\Box(\varphi \Rightarrow \psi)|_i^\sigma = \mathbf{ff}$ or $\Box\varphi|_i^\sigma = \mathbf{ff}$ or $\Box\psi|_i^\sigma = \mathbf{tt}$

and by the definition of *henceforth* and the semantics of *weak until*

2. for some j, $i \leq j < |ts(\sigma)|$, $(\varphi \Rightarrow \psi)|_j^\sigma = \mathbf{ff}$
   or for some j, $i \leq j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{ff}$
   or for every j, $i \leq j < |ts(\sigma)|$, $\psi|_j^\sigma = \mathbf{tt}$

and by the semantic definition of *implication*

3. for some j, $i \leq j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{tt}$ and $\psi|_j^\sigma = \mathbf{ff}$
   or for some j, $i \leq j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{ff}$
   or for every j, $i \leq j < |ts(\sigma)|$, $\psi|_j^\sigma = \mathbf{tt}$

(3) holds for every σ and i. This proves soundness of the axiom. ■

F2. $\vdash \odot(\varphi \Rightarrow \psi) \Rightarrow (\odot\varphi \Rightarrow \odot\psi)$

Proof :

By the semantic definition of *implication*, we have

1. $\odot(\varphi \Rightarrow \psi)|_i^\sigma = \mathbf{ff}$ or $\odot\varphi|_i^\sigma = \mathbf{ff}$ or $\odot\psi|_i^\sigma = \mathbf{tt}$

and by the semantic definition of *weak next*

2. $(|ts(\sigma)| \neq i+1$ and $(\varphi \Rightarrow \psi)|_{i+1}^\sigma = \mathbf{ff})$
   or $(|ts(\sigma)| \neq i+1$ and $\varphi|_{i+1}^\sigma = \mathbf{ff})$ or $(|ts(\sigma)| = i+1$ or $\psi|_{i+1}^\sigma = \mathbf{tt})$

and by the semantic definition of *implication*

3. $(|ts(\sigma)| \neq i+1$ and $\varphi|_{i+1}^\sigma = \mathbf{tt}$ and $\psi|_{i+1}^\sigma = \mathbf{ff})$
   or $(|ts(\sigma)| \neq i+1$ and $\varphi|_{i+1}^\sigma = \mathbf{ff})$ or $(|ts(\sigma)| = i+1$ or $\psi|_{i+1}^\sigma = \mathbf{tt})$

(3) holds for every σ and i. This proves soundness of the axiom. ■

F3. $\vdash \oplus\varphi \Rightarrow \odot\varphi$

Proof :

By the definition of *strong next*, we have

1. $\neg \odot \neg\varphi \Rightarrow \odot\varphi$

and by the semantic definition of *implication* and *negation*

2. $(\odot\neg\varphi)|_i^\sigma = \mathbf{tt}$ or $(\odot\varphi)|_i^\sigma = \mathbf{tt}$

and by the semantic definition of *weak next* and *negation*

3. $(|ts(\sigma)| = i+1$ or $\varphi|_{i+1}^\sigma = \mathbf{ff})$ or $(|ts(\sigma)| = i+1$ or $\varphi|_{i+1}^\sigma = \mathbf{tt})$

(3) holds for every σ and i. This proves soundness of the axiom. ■

F4. $\vdash \varphi \Rightarrow \odot \ominus \varphi$

Proof :

By the semantic definition of *implication*, we have

1. $\varphi|_i^\sigma = \mathbf{ff}$ or $(\odot \ominus \varphi)|_i^\sigma = \mathbf{tt}$

and by the semantic definition of *weak next* and the definition of *strong previous*

2. $\varphi|_i^\sigma = \mathbf{ff}$ or $(|ts(\sigma)| = i+1$ or $(\neg \oslash \neg\varphi)|_{i+1}^\sigma = \mathbf{tt})$

and by the semantic definitions of *weak previous* and *negation*

3. $\varphi|_i^\sigma = \mathbf{ff}$ or $(|ts(\sigma)| = i+1$ or $((i+1 \neq 0)$ and $\varphi|_i^\sigma = \mathbf{tt}))$

(3) holds for every σ and i. This proves soundness of the axiom.

■

F5. $\vdash \Box\varphi \Rightarrow \Box \odot \varphi$

Proof:

By the semantic definition of *implication*, we have

1. $\Box\varphi|_i^\sigma = \mathbf{ff}$ or $(\Box \odot \varphi)|_i^\sigma = \mathbf{tt}$

and by the definition of *henceforth* and the semantics of *weak until*

2. for some j, $i \leq j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{ff}$
   or for every j, $i \leq j < |ts(\sigma)|$, $(\odot\varphi)|_j^\sigma = \mathbf{tt}$

and by the semantic definition of *weak next*

3. for some j, $i \leq j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{ff}$
   or for every j, $i \leq j < |ts(\sigma)|$, $|ts(\sigma)| = j+1$ or $\varphi|_{j+1}^\sigma = \mathbf{tt}$

(3) holds for every σ and i. This proves soundness of the axiom.

■

F6. $\vdash \Box(\varphi \Rightarrow \odot\varphi) \Rightarrow (\varphi \Rightarrow \Box\varphi)$

Proof :

By the semantic definition of *implication*, we have

1. $\Box(\varphi \Rightarrow \odot\varphi)|_i^\sigma = \mathbf{ff}$ or $\varphi|_i^\sigma = \mathbf{ff}$ or $\Box\varphi|_i^\sigma = \mathbf{tt}$

and by the definition of *henceforth* and the semantic definition of *weak until*

2. for some j, $i \leq j < |ts(\sigma)|$, $((\varphi \Rightarrow \odot\varphi)|^{\sigma}_{j} = \mathbf{ff})$
or $\varphi|^{\sigma}_{i} = \mathbf{ff}$ or for every j, $i \leq j < |ts(\sigma)|$, $\varphi|^{\sigma}_{j} = \mathbf{tt}$

and by the semantic definition of *implication* and *weak next*

3. for some j, $i \leq j < |ts(\sigma)|$, $(\varphi|^{\sigma}_{j} = \mathbf{tt}$ and $(|ts(\sigma)| \neq j+1$
and $\varphi|^{\sigma}_{j+1} = \mathbf{ff}))$ or $\varphi|^{\sigma}_{i} = \mathbf{ff}$ or for every j, $i \leq j < |ts(\sigma)|$, $\varphi|^{\sigma}_{j} = \mathbf{tt}$

(3) holds for every σ and i. This proves soundness of the axiom.

■

F7. $\vdash \varphi \mathbf{U} \psi \Leftrightarrow (\psi \vee (\varphi \wedge \odot(\varphi \mathbf{U} \psi)))$

Proof :

By the semantic definition of *equivalence*, we have

1. $(\varphi \mathbf{U} \psi \Rightarrow (\psi \vee (\varphi \wedge \odot(\varphi \mathbf{U} \psi)))) \wedge ((\psi \vee (\varphi \wedge \odot(\varphi \mathbf{U} \psi))) \Rightarrow \varphi \mathbf{U} \psi)$

We prove soundness of $\varphi \mathbf{U} \psi \Rightarrow (\psi \vee (\varphi \wedge \odot(\varphi \mathbf{U} \psi)))$. The proof of soundness for $(\psi \vee (\varphi \wedge \odot(\varphi \mathbf{U} \psi))) \Rightarrow \varphi \mathbf{U} \psi$ is similar. By the semantic definition of *implication*

2. $\varphi \mathbf{U} \psi|^{\sigma}_{i} = \mathbf{ff}$ or $(\psi|^{\sigma}_{i} = \mathbf{tt}$ or $(\varphi|^{\sigma}_{i} = \mathbf{tt}$ and $\odot(\varphi \mathbf{U} \psi)|^{\sigma}_{i} = \mathbf{tt}))$

By the semantic definition of *weak until*

3. for some j, $i \leq j < |ts(\sigma)|$, $\varphi|^{\sigma}_{j} = \mathbf{ff}$
and for every k, $i \leq k < |ts(\sigma)|$, $\psi|^{\sigma}_{k} = \mathbf{ff}$
or for some j, $i \leq j < k$, $\varphi|^{\sigma}_{j} = \mathbf{ff}$
or $(\psi|^{\sigma}_{i} = \mathbf{tt}$ or $(\varphi|^{\sigma}_{i} = \mathbf{tt}$ and $(|ts(\sigma)| = i+1$
or for every j, $i+1 \leq j < |ts(\sigma)|$, $\varphi|^{\sigma}_{j} = \mathbf{tt}$
or for some k, $i+1 \leq k < |ts(\sigma)|$, $\psi|^{\sigma}_{k} = \mathbf{tt}$
and for every j, $i+1 \leq j < k$, $\varphi|^{\sigma}_{j} = \mathbf{tt})))$

(3) holds for every σ and i. This proves soundness of the axiom.

■

F8. $\vdash \Box\varphi \Rightarrow \varphi \mathbf{U} \psi$

Proof :

By the semantic definition of *implication*, we have

1. $\Box\varphi|_i^\sigma = \mathbf{ff}$ or $\varphi \mathsf{U} \psi|_i^\sigma = \mathbf{tt}$

and by the definition of *henceforth* and the semantic definition of *weak until*

2. for some j, $i \le j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{ff}$
   or for every j, $i \le j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{tt}$
   or for some k, $i \le k < |ts(\sigma)|$, $\psi|_k^\sigma = \mathbf{tt}$
   and for every j, $i \le j < k$, $\varphi|_j^\sigma = \mathbf{tt}$

(2) holds for every σ and i. This proves soundness of the axiom. ■

F9. $\vdash (\forall x. \odot \varphi) \Rightarrow (\odot \forall x. \varphi)$ where x is a global variable

Proof :

By the semantic definition of *implication*, we have

1. $(\forall x. \odot \varphi)|_i^\sigma = \mathbf{ff}$ or $(\odot \forall x. \varphi)|_i^\sigma = \mathbf{tt}$

and by the semantic definition of the universal quantifier over global variables and *weak next*

2. (for some $d \in D$, $|ts(\sigma)| \neq i+1$ and $\varphi(d/x)|_{i+1}^\sigma = \mathbf{ff}$)
   or ($|ts(\sigma)| = i+1$ or (for every $d \in D$, $\varphi(d/x))|_{i+1}^\sigma = \mathbf{tt}$)

(2) holds for every σ and i. This proves soundness of the axiom. ■

F10. $\vdash \odot F(t_1, \ldots, t_n) = F(\odot t_1, \ldots, \odot t_n)$

Proof :

By the semantic definition of *weak next term* and an n-ary function, we have

$$1.\ \odot F(t_1, \ldots, t_n)|_i^\sigma = \begin{cases} F(t_1, \ldots, t_n)|_i^\sigma & \text{if } |ts(\sigma)| = i+1 \\ F(t_1, \ldots, t_n)|_{i+1}^\sigma & \text{if } |ts(\sigma)| \neq i+1 \end{cases}$$

$$= \begin{cases} I(F)(t_1|_i^\sigma, \ldots, t_n|_i^\sigma) & \text{if } |ts(\sigma)| = i+1 \\ I(F)(t_1|_{i+1}^\sigma, \ldots, t_n|_{i+1}^\sigma) & \text{if } |ts(\sigma)| \neq i+1 \end{cases}$$

$$= F(\odot t_1, \ldots, \odot t_n)$$

This proves soundness of the axiom. ■

F11. $\vdash \odot P(t_1,\ldots,t_n) = P(\odot t_1,\ldots,\odot t_n)$

Proof :

The proof of soundness is a slight variation of the proof for F10.

■

F12. $\vdash \odot false \Leftrightarrow (\odot T) = T$

Proof :

By the definition of *equivalence* and the semantic definition of *implication*, we have

1. $(\odot false|_i^\sigma = \mathbf{ff}$ or $((\odot T) = T)|_i^\sigma = \mathbf{tt})$ and $(((\odot T) = T)|_i^\sigma = \mathbf{ff}$ or $\odot\ false|_i^\sigma = \mathbf{tt})$

and by the semantic definitions of *weak next* and *weak next term*

2.$(|ts(\sigma)| \neq i+1$ or $|ts(\sigma)| = i+1)$ and $(|ts(\sigma)| \neq i+1$ or $|ts(\sigma)| = i+1)$

(2) holds for every σ and i. This proves soundness of the axiom.

■

F13. $\vdash \neg \odot\ false \Leftrightarrow (\odot T) > T$

Proof :

The proof of soundness is a slight variation of the proof for F12.

■

F14. $\vdash (\forall x. \odot \varphi) \Rightarrow (\odot \forall x.\varphi)$ where x is a local variable

Proof :

By the semantic definition of *implication*, we have

1. $(\forall x. \odot \varphi)|_i^\sigma = \mathbf{ff}$ or $(\odot \forall x.\varphi)|_i^\sigma = \mathbf{tt}$

and by the semantic definitions of the universal quantifier over local variables and *weak next*

2. (for some $\sigma' \in \Sigma$, $ts(\sigma') = ts(\sigma)$ and $\varphi|_i^{\sigma'} = \mathbf{ff}$, where σ' is obtained from σ by assigning at all $\tau \in \Gamma$ the same values to all local variables except x)
   or ($|ts(\sigma)| = i + 1$ or for every $\sigma' \in \Sigma$ such that $ts(\sigma') = ts(\sigma)$, $\varphi|_i^{\sigma'} = \mathbf{tt}$, where σ' is obtained from σ by assigning at all $\tau \in \Gamma$ the same values to all local variables except x)

(2) holds for every σ and i. This proves soundness of the axiom.

■

Axioms P1-P7 are symmetrical to F1-F7 and their proofs of soundness are similar. The proof of soundness of P8 is as follows :

P8. $\vdash \Diamond \oslash \mathit{false}$

Proof :

By the definition of *sometime in the past*, and the semantic definition of *weak since*

1. for some k, $0 \leq k \leq i$, $\oslash \mathit{false}|_k^\sigma = \mathbf{tt}$

and by the semantic definition of *weak previous*

2. for some k, $0 \leq k \leq i$, $k = 0$

(2) holds for every σ and i. This proves soundness of the axiom.

■

The proofs of soundness of Axioms P9-P14 are similar to those for F9-F14. This completes the proofs of soundness of all the axioms of ωTL. We now prove soundness of the inference rules.

R1. If φ is a propositional tautology then $\vdash \varphi$

Proof :

Since φ is a tautology in propositional calculus, we have

1. for every σ and i, $\varphi|_i^\sigma = \mathbf{tt}$

and by the definition of $\vdash$

2. $\vdash \varphi$

This proves soundness of the inference rule. ∎

R2. $\dfrac{\vdash \varphi, \vdash \varphi \Rightarrow \psi}{\vdash \psi}$

Proof :

By the definition of $\vdash$, we have

1. for every σ and i, $\varphi|_i^\sigma = \mathbf{tt}$ and
   for every σ and i, $(\varphi \Rightarrow \psi)|_i^\sigma = \mathbf{tt}$

and by the semantic definition of *implication*

2. for every σ and i, $\varphi|_i^\sigma = \mathbf{tt}$ and
   for every σ and i, $\varphi|_i^\sigma = \mathbf{ff}$ or $\psi|_i^\sigma = \mathbf{tt}$

(3) follows from (2)

3. for every σ and i, $\psi|_i^\sigma = \mathbf{tt}$

and by the definition of $\vdash$

4. $\vdash \psi$

This proves soundness of the inference rule. ∎

R3. $\dfrac{\vdash \varphi}{\vdash \Box\varphi \wedge \boxdot\varphi}$

Proof :

By the definition of $\vdash$, we have

1. for every σ and i, $\varphi|_i^\sigma = \mathbf{tt}$

(2) follows from (1).

2. for every σ and i, for every j, $i \leq j < |ts(\sigma)|$, $\varphi|_j^\sigma = \mathbf{tt}$
   and for every σ and i, for every j, $0 \leq j \leq i$, $\varphi|_j^\sigma = \mathbf{tt}$

and by the definitions of $\vdash$, *henceforth* and *and*

3. $\vdash \Box\varphi \wedge \boxdot\varphi$

This proves soundness of the inference rule. ■

R4. If φ is a tautology in predicate calculus then $\vdash \varphi$

Proof :

The proof of soundness is a slight variation of the proof for R1. ■

R5. $\dfrac{\vdash \varphi \Rightarrow \psi}{\vdash \varphi \Rightarrow \forall x.\psi}$

where x is a global variable not free in φ

Proof :

By the definition of $\vdash$ and the semantic definition of *implication*, we have

1. for every σ and i, $\varphi|_i^\sigma = \mathbf{ff}$ or $\psi|_i^\sigma = \mathbf{tt}$

and since x is not free in φ

2. for every σ and i, $\varphi|_i^\sigma = \mathbf{ff}$ or for every $d \in D$, $\psi(d/x)|_i^\sigma = \mathbf{tt}$

and by the definition of $\vdash$, and the semantic definitions of *implication* and the universal quantifier over global variables

3. $\vdash \varphi \Rightarrow \forall x.\psi$

This proves soundness of the inference rule. ■

R6. $\dfrac{\vdash \varphi \Rightarrow \psi}{\vdash \varphi \Rightarrow \forall x.\psi}$

where x is a local variable not free in φ

Proof :

The proof of soundness is a slight variation of the proof for R5. ■

This completes the proofs of soundness of all the inference rules of ωTL.

A.2 Theorems of the Logic

In this section, we give some theorems which hold in ωTL. The proofs of these theorems are similar to the proofs given for theorems in TL [MP82] and in the 'anchored' version [MP89a] and therefore have been omitted.

Theorems and Rules about $\odot$

$\odot$-Insertion

$$\frac{\vdash \varphi}{\vdash \odot\varphi}$$

$\odot\odot$-Rules

$$\frac{\vdash \varphi \Rightarrow \psi}{\vdash \odot\varphi \Rightarrow \odot\psi} \qquad \frac{\vdash \varphi \Leftrightarrow \psi}{\vdash \odot\varphi \Leftrightarrow \odot\psi}$$

C-IND

$$\frac{\vdash \varphi \Rightarrow \odot\varphi}{\vdash \varphi \Rightarrow \Box\varphi}$$

DC-IND

$$\frac{\vdash \varphi \Rightarrow (\psi \wedge \odot\varphi)}{\vdash \varphi \Rightarrow \Box\psi}$$

T1. $\vdash \odot(\varphi \wedge \psi) \Leftrightarrow (\odot\varphi \wedge \odot\psi)$
T2. $\vdash \odot(\varphi \vee \psi) \Leftrightarrow (\odot\varphi \vee \odot\psi)$
T3. $\vdash \neg \odot \varphi \Leftrightarrow \oplus\neg\varphi$
T4. $\vdash \odot(\varphi \Rightarrow \psi) \Leftrightarrow (\odot\varphi \Rightarrow \odot\psi)$
T5. $\vdash \odot(\varphi \Leftrightarrow \psi) \Leftrightarrow (\odot\varphi \Leftrightarrow \odot\psi)$

Theorems and Rules about $\oplus$

$\oplus\oplus$-Rules

$$\frac{\vdash \varphi \Rightarrow \psi}{\vdash \oplus\varphi \Rightarrow \oplus\psi} \qquad \frac{\vdash \varphi \Leftrightarrow \psi}{\vdash \oplus\varphi \Leftrightarrow \oplus\psi}$$

C-IND

$$\frac{\vdash \varphi \Rightarrow \oplus\varphi}{\vdash \varphi \Rightarrow \Box\varphi}$$

DC-IND

$$\frac{\vdash \varphi \Rightarrow (\psi \wedge \oplus\varphi)}{\vdash \varphi \Rightarrow \Box\psi}$$

T6. $\vdash \oplus(\varphi \wedge \psi) \Leftrightarrow (\oplus\varphi \wedge \oplus\psi)$
T7. $\vdash \oplus(\varphi \vee \psi) \Leftrightarrow (\oplus\varphi \vee \oplus\psi)$
T8. $\vdash \oplus(\varphi \Rightarrow \psi) \Leftrightarrow (\oplus\varphi \Rightarrow \oplus\psi)$
T9. $\vdash \oplus(\varphi \Leftrightarrow \psi) \Leftrightarrow (\oplus\varphi \Leftrightarrow \oplus\psi)$

Theorems and Rules about $\Box$

$\Box\Box$-Rules

$$\frac{\vdash \varphi \Rightarrow \psi}{\vdash \Box\varphi \Rightarrow \Box\psi} \qquad \frac{\vdash \varphi \Leftrightarrow \psi}{\vdash \Box\varphi \Leftrightarrow \Box\psi}$$

T10. $\vdash \Box\varphi \Leftrightarrow (\varphi \wedge \odot\Box\varphi)$
T11. $\vdash \Box\varphi \Rightarrow \varphi$
T12. $\vdash \Box\varphi \Rightarrow \odot\Box\varphi$
T13. $\vdash \Box\varphi \Rightarrow \odot\varphi$
T14. $\vdash \Box\varphi \Leftrightarrow \Box\Box\varphi$
T15. $\vdash \Box(\varphi \wedge \psi) \Leftrightarrow (\Box\varphi \wedge \Box\psi)$
T16. $\vdash (\Box\varphi \vee \Box\psi) \Rightarrow \Box(\varphi \vee \psi)$
T17. $\vdash \odot\Box\varphi \Leftrightarrow \Box \odot \varphi$

Theorems and Rules about $\Diamond$

$\Diamond\Diamond$-Rules

$$\frac{\vdash \varphi \Rightarrow \psi}{\vdash \Diamond\varphi \Rightarrow \Diamond\psi} \qquad \frac{\vdash \varphi \Leftrightarrow \psi}{\vdash \Diamond\varphi \Leftrightarrow \Diamond\psi}$$

T18. $\vdash \varphi \Rightarrow \Diamond\varphi$
T19. $\vdash \Diamond\varphi \Leftrightarrow \Diamond\Diamond\varphi$
T20. $\vdash \Diamond(\varphi \wedge \psi) \Rightarrow (\Diamond\varphi \wedge \Diamond\psi)$
T21. $\vdash \Diamond(\varphi \vee \psi) \Leftrightarrow (\Diamond\varphi \vee \Diamond\psi)$
T22. $\vdash (\Box\varphi \wedge \Diamond\psi) \Rightarrow \Diamond(\varphi \wedge \psi)$
T23. $\vdash \oplus\Diamond\varphi \Leftrightarrow \Diamond \oplus \varphi$
T24. $\vdash \Diamond\varphi \Leftrightarrow \varphi \vee \oplus\Diamond\varphi$
T25. $\vdash \Box\Diamond\varphi \Leftrightarrow \Diamond\Box\varphi$
T26. $\vdash \Diamond\Box\Diamond\varphi \Leftrightarrow \Box\Diamond\varphi$
T27. $\vdash (\varphi \wedge \Diamond\neg\varphi) \Rightarrow \Diamond(\varphi \wedge \oplus\neg\varphi)$

Consequence Rules

$$\frac{\vdash \varphi_1 \Rightarrow \varphi_2, \vdash \varphi_2 \Rightarrow \Box\psi_1, \vdash \psi_1 \Rightarrow \psi_2}{\vdash \varphi_1 \Rightarrow \Box\psi_2}$$

$$\frac{\vdash \varphi_1 \Rightarrow \varphi_2, \vdash \varphi_2 \Rightarrow \Diamond\psi_1, \vdash \psi_1 \Rightarrow \psi_2}{\vdash \varphi_1 \Rightarrow \Diamond\psi_2}$$

$$\frac{\vdash \varphi_1 \Rightarrow \varphi_2, \vdash \varphi_2 \Rightarrow \odot\psi_1, \vdash \psi_1 \Rightarrow \psi_2}{\vdash \varphi_1 \Rightarrow \odot\psi_2}$$

$$\frac{\vdash \varphi_1 \Rightarrow \varphi_2, \vdash \varphi_2 \Rightarrow \oplus\psi_1, \vdash \psi_1 \Rightarrow \psi_2}{\vdash \varphi_1 \Rightarrow \oplus\psi_2}$$

Theorems and Rules about U

UU-Rules

$$\frac{\vdash \varphi_1 \Rightarrow \varphi_2, \vdash \psi_1 \Rightarrow \psi_2}{\vdash \varphi_1 \mathsf{U} \psi_1 \Rightarrow \varphi_2 \mathsf{U} \psi_2} \qquad \frac{\vdash \varphi_1 \Leftrightarrow \varphi_2, \vdash \psi_1 \Leftrightarrow \psi_2}{\vdash \varphi_1 \mathsf{U} \psi_1 \Leftrightarrow \varphi_2 \mathsf{U} \psi_2}$$

T28. $\vdash (\neg\varphi)\mathsf{U}\varphi$

T29. $\vdash (\varphi \wedge \psi)\mathsf{U}\omega \Leftrightarrow (\varphi\mathsf{U}\omega) \wedge (\psi\mathsf{U}\omega)$
T30. $\vdash \varphi\mathsf{U}(\psi \vee \omega) \Leftrightarrow (\varphi\mathsf{U}\psi) \vee (\varphi\mathsf{U}\omega)$
T31. $\vdash (\varphi\mathsf{U}\psi)\mathsf{U}\psi \Leftrightarrow \varphi\mathsf{U}\psi$
T32. $\vdash \varphi\mathsf{U}(\varphi\mathsf{U}\psi) \Leftrightarrow \varphi\mathsf{U}\psi$

U Introduction

$$\frac{\vdash \varphi}{\vdash \varphi\mathsf{U}\psi}$$

U Concatenation Rule

$$\frac{\vdash \varphi_1 \Rightarrow \psi\mathsf{U}\varphi_2, \vdash \varphi_2 \Rightarrow \psi\mathsf{U}\varphi_3}{\vdash \varphi_1 \Rightarrow \psi\mathsf{U}\varphi_3}$$

T33. $\vdash (\Box\varphi \wedge \psi\mathsf{U}\omega) \Rightarrow (\varphi \wedge \psi)\mathsf{U}(\varphi \wedge \omega)$
T34. $\vdash \varphi\mathsf{U}(\psi \wedge \omega) \Rightarrow (\varphi\mathsf{U}\psi \wedge \varphi\mathsf{U}\omega)$
T35. $\vdash \varphi\mathsf{U}\omega \vee \psi\mathsf{U}\omega \Rightarrow (\varphi \vee \psi)\mathsf{U}\omega$
T36. $\vdash (\varphi \Rightarrow \psi)\mathsf{U}\omega \Rightarrow (\varphi\mathsf{U}\omega \Rightarrow \psi\mathsf{U}\omega)$
T37. $\vdash \varphi\mathsf{U}\psi \wedge (\neg\psi)\mathsf{U}\omega \Rightarrow \varphi\mathsf{U}\omega$
T38. $\vdash \varphi\mathsf{U}(\psi \wedge \omega) \Rightarrow (\varphi\mathsf{U}\psi)\mathsf{U}\omega$
T39. $\vdash (\varphi\mathsf{U}\psi)\mathsf{U}\omega \Rightarrow (\varphi \vee \psi)\mathsf{U}\omega$
T40. $\vdash \varphi\mathsf{U}(\psi\mathsf{U}\omega) \Rightarrow (\varphi \vee \psi)\mathsf{U}\omega$

Theorems and Rules about $\mathcal{U}$

Right $\mathcal{U}$-Rule

$$\frac{\vdash \omega \Rightarrow \Diamond\psi, \vdash \omega \Rightarrow (\psi \vee (\varphi \wedge \oplus\omega))}{\vdash \omega \Rightarrow (\varphi\mathcal{U}\psi)}$$

Left $\mathcal{U}$-Rule

$$\frac{\vdash (\psi \vee (\varphi \wedge \oplus\omega)) \Rightarrow \omega}{\vdash (\varphi\mathcal{U}\psi) \Rightarrow \omega}$$

$\mathcal{U}\mathcal{U}$-Rules

$$\frac{\vdash \varphi_1 \Rightarrow \varphi_2, \vdash \psi_1 \Rightarrow \psi_2}{\vdash \varphi_1 \mathcal{U} \psi_1 \Rightarrow \varphi_2 \mathcal{U} \psi_2} \qquad \frac{\vdash \varphi_1 \Leftrightarrow \varphi_2, \vdash \psi_1 \Leftrightarrow \psi_2}{\vdash \varphi_1 \mathcal{U} \psi_1 \Leftrightarrow \varphi_2 \mathcal{U} \psi_2}$$

T41. $\vdash (\neg\varphi)\mathcal{U}\varphi$
T42. $\vdash (\varphi \wedge \psi)\mathcal{U}\omega \Leftrightarrow (\varphi\mathcal{U}\omega) \wedge (\psi\mathcal{U}\omega)$
T43. $\vdash \varphi\mathcal{U}(\psi \vee \omega) \Leftrightarrow (\varphi\mathcal{U}\psi) \vee (\varphi\mathcal{U}\omega)$
T44. $\vdash (\varphi\mathcal{U}\psi)\mathcal{U}\psi \Leftrightarrow \varphi\mathcal{U}\psi$
T45. $\vdash \varphi\mathcal{U}(\varphi\mathcal{U}\psi) \Leftrightarrow \varphi\mathcal{U}\psi$
T46. $\vdash (\Box\varphi \wedge \Diamond\psi) \Rightarrow \varphi\mathcal{U}\psi$

$\mathcal{U}$ Introduction

$$\frac{\vdash \varphi}{\vdash \varphi\mathcal{U}\psi} \qquad \frac{\vdash \varphi, \vdash \Diamond\psi}{\vdash \varphi\mathcal{U}\psi}$$

$\mathcal{U}$ Concatenation Rule

$$\frac{\vdash \varphi_1 \Rightarrow \psi\mathcal{U}\varphi_2, \vdash \varphi_2 \Rightarrow \psi\mathcal{U}\varphi_3}{\vdash \varphi_1 \Rightarrow \psi\mathcal{U}\varphi_3}$$

T47. $\vdash (\Box\varphi \wedge \psi\mathcal{U}\omega) \Rightarrow (\varphi \wedge \psi)\mathcal{U}(\varphi \wedge \omega)$
T48. $\vdash \varphi\mathcal{U}(\psi \wedge \omega) \Rightarrow (\varphi\mathcal{U}\psi \wedge \varphi\mathcal{U}\omega)$
T49. $\vdash \varphi\mathcal{U}\omega \vee \psi\mathcal{U}\omega \Rightarrow (\varphi \vee \psi)\mathcal{U}\omega$
T50. $\vdash (\varphi \Rightarrow \psi)\mathcal{U}\omega \Rightarrow (\varphi\mathcal{U}\omega \Rightarrow \psi\mathcal{U}\omega)$
T51. $\vdash (\varphi\mathcal{U}\psi) \wedge (\neg\psi)\mathcal{U}\omega \Rightarrow \varphi\mathcal{U}\omega$
T52. $\vdash \varphi\mathcal{U}(\psi \wedge \omega) \Rightarrow (\varphi\mathcal{U}\psi)\mathcal{U}\omega$
T53. $\vdash (\varphi\mathcal{U}\psi)\mathcal{U}\omega \Rightarrow (\varphi \vee \psi)\mathcal{U}\omega$
T54. $\vdash \varphi\mathcal{U}(\psi\mathcal{U}\omega) \Rightarrow (\varphi \vee \psi)\mathcal{U}\omega$
T55. $\vdash (\oplus\varphi)\mathcal{U}(\oplus\psi) \Leftrightarrow \oplus(\varphi\mathcal{U}\psi)$
T56. $\vdash (\Diamond\varphi \vee \Diamond\psi) \Rightarrow ((\neg\varphi)\mathcal{U}\psi \vee (\neg\psi)\mathcal{U}\varphi)$

Theorems and Rules about Quantifiers

Deduction Rule (DED)

$$\frac{\varphi \vdash \psi}{\vdash (\Box\varphi) \Rightarrow \psi}$$

where the $\forall$ Introduction rule is never applied to a free variable of φ in the derivation of $\varphi \vdash \psi$

Reduced Deduction Rule (RDED)

$$\frac{\varphi \vdash \psi}{\vdash \varphi \Rightarrow \psi}$$

where $\Box$ Introduction rule is never applied and the $\forall$ Introduction rule is never applied to a free variable of φ in the derivation of $\varphi \vdash \psi$

T57. $\vdash (\forall x. \odot \varphi) \Leftrightarrow (\odot \forall x.\varphi)$
T58. $\vdash (\exists x. \odot \varphi) \Leftrightarrow (\odot \exists x.\varphi)$
T59. $\vdash (\forall x.\Box\varphi) \Leftrightarrow (\Box \forall x.\varphi)$
T60. $\vdash (\exists x.\Diamond\varphi) \Leftrightarrow (\Diamond \exists x.\varphi)$
T61. $\vdash (\forall x.\varphi \mathsf{U} \psi) \Leftrightarrow (\forall x.\varphi)\mathsf{U}\psi$ where x is not free in ψ
T62. $\vdash \exists x.(\varphi \mathsf{U} \psi) \Leftrightarrow \varphi \mathsf{U} (\exists x.\psi)$ where x is not free in φ
T63. $\vdash \exists x.\Box\varphi \Rightarrow \Box \exists x.\varphi$
T64. $\vdash \Diamond \forall x.\varphi \Rightarrow \forall x.\Diamond\varphi$
T65. $\vdash \exists x.(\varphi \mathsf{U} \psi) \Rightarrow (\exists x.\varphi)\mathsf{U}\psi$ where x is not free in ψ
T66. $\vdash \varphi \mathsf{U} (\forall x.\psi) \Rightarrow \forall x.(\varphi \mathsf{U} \psi)$ where x is not free in φ
T67. $\vdash \exists x.(\varphi \mathsf{U} \psi) \Rightarrow (\exists x.\varphi)\mathsf{U}(\exists x.\psi)$
T68. $\vdash (\forall x.\varphi)\mathsf{U}(\forall x.\psi) \Rightarrow \forall x.(\varphi \mathsf{U} \psi)$

Theorems and Rules about Equality

T69. $\vdash (t_1 = t_2) \Rightarrow (t_2 = t_1)$
T70. $\vdash (t_1 = t_2) \wedge (t_2 = t_3) \Rightarrow (t_1 = t_3)$
T71. $\vdash \Box(t_1 = t_2) \Rightarrow (\tau(t_1, t_1) = \tau(t_1, t_2))$ for any term τ

T72. $\vdash (t_1 = t_2) \Rightarrow (\tau(t_1, t_1) = \tau(t_1, t_2))$ for any term τ provided it does not contain $\odot$ operator

T73. $\vdash \Box(t_1 = t_2) \Rightarrow (\varphi(t_1, t_1) \Leftrightarrow \varphi(t_1, t_2))$ where t_2 is free for t_1 in φ

The theorems for the past fragment of the logic are similar to the theorems for the future fragment.

Appendix B

Calculus : Proofs of the Derived Rules

We prove the derived rules of the calculus of durations and occurrences of predicates given in Section 3.4. Recall that $\mathcal{D}(\varphi, d)$ was defined as follows :

$$\begin{aligned}\mathcal{D}(\varphi, d) \triangleq \Box\boxminus(((T = 0) \Rightarrow (d = 0)) \\ \wedge((T \neq 0) \Rightarrow ((\oslash\varphi \Leftrightarrow (d = (\oslash d) + T - \oslash T)) \\ \wedge(\oslash\neg\varphi \Leftrightarrow (d = \oslash d)))))\end{aligned}$$

S1. $\vdash \mathcal{D}(\mathit{false}, d) \Rightarrow \Box(d = 0) \wedge \boxminus(d = 0)$

Proof :

By substituting false for φ in the definition of $\mathcal{D}(\varphi, d)$.

1. $\begin{aligned}\Box\boxminus(((T = 0) \Rightarrow (d = 0)) \\ \wedge((T \neq 0) \Rightarrow ((\oslash \mathit{false} \Leftrightarrow (d = (\oslash d) + T - \oslash T)) \\ \wedge(\oslash \mathit{true} \Leftrightarrow (d = \oslash d)))))\end{aligned}$

Since $\vdash T \neq 0 \Leftrightarrow \neg \oslash \mathit{false}$, we have

2. $\Box\boxminus(((T = 0) \Rightarrow (d = 0)) \wedge ((T \neq 0) \Rightarrow (d = \oslash d)))$

(3) follows from (2) and the induction hypothesis.

3. $\Box\boxminus(d = 0)$

(4) follows from (3), $\vdash \Box\varphi \Rightarrow \varphi$ (T11) and Modus Ponens (R2).

4. $\boxdot(d = 0)$

(5) follows from (3), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\vdash \Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

5. $\Box(d = 0)$

(6) follows from (4) and (5).

6. $\Box(d = 0) \wedge \boxdot(d = 0)$

This proves the derived rule. ∎

S2. $\vdash \mathcal{D}(true, d) \Rightarrow \Box(d = T) \wedge \boxdot(d = T)$

Proof :

By substituting *true* for φ in the definition of $\mathcal{D}(\varphi, d)$.

1. $\Box\boxdot(((T = 0) \Rightarrow (d = 0))$
$\wedge((T \neq 0) \Rightarrow ((\oslash true \Leftrightarrow (d = (\oslash d) + T - \oslash T))$
$\wedge(\oslash false \Leftrightarrow (d = \oslash d)))))$

Since $\vdash T \neq 0 \Leftrightarrow \neg \oslash false$, we have

2. $\Box\boxdot(((T = 0) \Rightarrow (d = 0)) \wedge ((T \neq 0) \Rightarrow (d = (\oslash d) + T - \oslash T)))$

(3) follows from (2) and the induction hypothesis.

3. $\Box\boxdot(d = T)$

(4) follows from (3), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

4. $\Box(d = T) \wedge \boxdot(d = T)$

This proves the derived rule. ∎

S3. $\vdash \mathcal{D}(\varphi, d) \Rightarrow \Box(d \geq 0) \wedge \boxdot(d \geq 0))$

Proof :

(1) follows from the definition of $\mathcal{D}(\varphi, d)$.

1. $\Box\boxdot(((T = 0) \Rightarrow (d = 0))$
$\wedge((T \neq 0) \Rightarrow ((\oslash\varphi \Leftrightarrow (d = (\oslash d) + T - \oslash T))$
$\wedge(\oslash\neg\varphi \Leftrightarrow (d = \oslash d)))))$

(2) follows from (1), $\vdash T \geq (\oslash T)$ and the induction hypothesis.

2. $\Box\boxdot(d \geq 0)$

(3) follows from (2), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

3. $\Box(d \geq 0) \wedge \boxdot(d \geq 0)$

This proves the derived rule. ∎

S4. $\vdash \mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(\psi, d_2) \wedge \mathcal{D}(\varphi \vee \psi, d_3) \wedge \mathcal{D}(\varphi \wedge \psi, d_4)$
$\Rightarrow \Box(d_1 + d_2 = d_3 + d_4) \wedge \boxdot(d_1 + d_2 = d_3 + d_4)$

Proof :

By substituting for φ, ψ, $\varphi \vee \psi$, and $\varphi \wedge \psi$ for φ in $\mathcal{D}(\varphi, d)$ we have

1. $\Box\boxdot((T = 0 \Rightarrow (d_1 = 0 \wedge d_2 = 0 \wedge d_3 = 0 \wedge d_4 = 0))$
$\wedge(T \neq 0 \Rightarrow ((\oslash(\varphi \wedge \psi) \Leftrightarrow ((d_1 = (\oslash d_1) + T - \oslash T)$
$\wedge(d_2 = (\oslash d_2) + T - \oslash T) \wedge (d_3 = (\oslash d_3) + T - \oslash T)$
$\wedge(d_4 = (\oslash d_4) + T - \oslash T)))$
$\wedge(\oslash(\varphi \wedge \neg\psi) \Leftrightarrow ((d_1 = (\oslash d_1) + T - \oslash T)$
$\wedge(d_2 = \oslash d_2) \wedge (d_3 = (\oslash d_3) + T - \oslash T) \wedge (d_4 = \oslash d_4)))$
$\wedge(\oslash(\neg\varphi \wedge \psi) \Leftrightarrow ((d_1 = \oslash d_1)$
$\wedge(d_2 = \oslash d_2) + T - \oslash T \wedge (d_3 = (\oslash d_3) + T - \oslash T)$
$\wedge(d_4 = \oslash d_4)))$
$\wedge(\oslash(\neg\varphi \wedge \neg\psi) \Leftrightarrow ((d_1 = \oslash d_1)$
$\wedge(d_2 = \oslash d_2) \wedge (d_3 = \oslash d_3) \wedge (d_4 = \oslash d_4)))))$

(2) follows from (1) and the induction hypothesis.

2. $\Box\boxdot((T = 0 \Rightarrow (d_1 + d_2 = d_3 + d_4)) \wedge (T \neq 0 \Rightarrow (d_1 + d_2 = d_3 + d_4)))$

(3) follows from (2).

3. $\Box\boxdot(d_1 + d_2 = d_3 + d_4)$

(4) follows from (3), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

4. $\Box(d_1 + d_2 = d_3 + d_4) \wedge \boxdot(d_1 + d_2 = d_3 + d_4)$

This proves the derived rule.

■

We now prove the derived rules which apply over the number of occurrences of predicates. Recall that $\mathcal{N}(\varphi, n)$ was defined in Section 3.4 as follows:

$$\begin{aligned}\mathcal{N}(\varphi, n) \triangleq \Box\boxdot(((T = 0) \Rightarrow (\varphi \Leftrightarrow (n = 1)) \wedge (\neg\varphi \Leftrightarrow (n = 0)))\\ \wedge((T \neq 0) \Rightarrow (((\neg\varphi \vee (\varphi \wedge \oslash\varphi)) \Leftrightarrow (n = \oslash n))\\ \wedge((\varphi \wedge \oslash\neg\varphi) \Leftrightarrow (n = (\oslash n) + 1)))))\end{aligned}$$

S5. $\vdash \mathcal{N}(false, n) \Rightarrow \Box(n = 0) \wedge \boxdot(n = 0)$

Proof :

By substituting $false$ for φ in the definition of $\mathcal{N}(\varphi, n)$.

1. $\Box\boxdot(((T = 0) \Rightarrow (false \Leftrightarrow (n = 1)) \wedge (true \Leftrightarrow (n = 0)))$
   $\wedge((T \neq 0) \Rightarrow ((true \Leftrightarrow (n = \oslash n))$
   $\wedge(false \Leftrightarrow (n = (\oslash n) + 1))))$

(2) follows from (1).

2. $\Box\boxdot(((T = 0) \Rightarrow (n = 0)) \wedge ((T \neq 0) \Rightarrow (n = \oslash n)))$

(3) follows from (2) and the induction hypothesis.

3. $\Box\boxdot(n = 0)$

(4) follows from (3), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

4. $\Box(n = 0) \wedge \boxdot(n = 0)$

This proves the derived rule.

■

S6. $\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box(n \geq 0) \wedge \boxdot(n \geq 0)$

Proof :

(1) follows from the definition of $\mathcal{N}(\varphi, n)$.

1. $\Box\boxdot(((T=0)\Rightarrow(\varphi\Leftrightarrow(n=1))\wedge(\neg\varphi\Leftrightarrow(n=0)))$
$\wedge((T\neq 0)\Rightarrow(((\neg\varphi\vee(\varphi\wedge\oslash\varphi))\Leftrightarrow(n=\oslash n))$
$\wedge((\varphi\wedge\oslash\neg\varphi)\Leftrightarrow(n=(\oslash n)+1)))))$

(2) follows from (1) and the induction hypothesis.

2. $\Box\boxdot(n\geq 0)$

(3) follows from (2), $\vdash\boxdot\varphi\Rightarrow\varphi$, $\Box$ Introduction (R3), $\vdash\Box(\varphi\Rightarrow\psi)\Rightarrow(\Box\varphi\Rightarrow\Box\psi)$ (F1) and Modus Ponens (R2).

3. $\Box(n\geq 0)\wedge\boxdot(n\geq 0)$

This proves the derived rule. ■

S7. $\vdash\mathcal{N}(\varphi,n_1)\wedge\mathcal{N}(\neg\varphi,n_2)\Rightarrow$
$\Box((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1))$
$\wedge\boxdot((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1))$

Proof :

By substituting for φ and $\neg\varphi$ for φ in the definition of $\mathcal{N}(\varphi,n)$.

1. $\Box\boxdot(((T=0)\Rightarrow((\varphi\Leftrightarrow(n_1=1)\wedge(n_2=0)))$
$\wedge(\neg\varphi\Leftrightarrow(n_1=0)\wedge(n_2=1))))$
$\wedge((T\neq 0)\Rightarrow(((\varphi\wedge\oslash\varphi)\vee(\neg\varphi\wedge\oslash\neg\varphi)\Leftrightarrow((n_1=\oslash n_1)\wedge(n_2=\oslash n_2)))$
$\wedge((\varphi\wedge\oslash\neg\varphi)\Leftrightarrow((n_1=\oslash n_1+1)\wedge(n_2=\oslash n_2)))$
$\wedge((\neg\varphi\wedge\oslash\varphi)\Leftrightarrow((n_1=\oslash n_1)\wedge(n_2=\oslash n_2+1))))))$

(2) follows from (1), and the induction hypothesis.

2. $\Box\boxdot(((T=0)\Rightarrow((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1)))$
$\wedge((T\neq 0)\Rightarrow((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1))))$

(3) follows from (2).

3. $\Box\boxdot((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1))$

(4) follows from (3), $\vdash\boxdot\varphi\Rightarrow\varphi$, $\Box$ Introduction (R3), $\vdash\Box(\varphi\Rightarrow\psi)\Rightarrow(\Box\varphi\Rightarrow\Box\psi)$ (F1) and Modus Ponens (R2).

4. $\Box((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1))$
$\wedge\boxdot((\varphi\Rightarrow 0\leq n_1-n_2\leq 1)\wedge(\neg\varphi\Rightarrow 0\leq n_2-n_1\leq 1))$

This proves the derived rule. ■

S8. $\vdash \mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \Rightarrow \Box((n = 0) \Rightarrow (d = 0)) \wedge \boxdot((n = 0) \Rightarrow (d = 0))$

Proof :

By substituting for $\mathcal{D}(\varphi, d)$ and $\mathcal{N}(\varphi, n)$, we have

1. $\Box\boxdot(((T = 0) \wedge \varphi \Leftrightarrow ((d = 0) \wedge (n = 1)))$
   $\wedge((T = 0) \wedge \neg\varphi \Leftrightarrow ((d = 0) \wedge (n = 0)))$
   $\wedge((T \neq 0) \wedge \oslash\varphi \Leftrightarrow ((d = (\oslash d) + T - \oslash T) \wedge (n = \oslash n)))$
   $\wedge((T \neq 0) \wedge \varphi \wedge \oslash\neg\varphi \Leftrightarrow ((d = \oslash d) \wedge (n = (\oslash n) + 1)))$
   $\wedge((T \neq 0) \wedge \neg\varphi \wedge \oslash\neg\varphi \Leftrightarrow ((d = \oslash d) \wedge (n = \oslash n))))$

(2) follows from (1), the definition of $\mathcal{N}(\varphi, n)$ and the induction hypothesis.

2. $\Box\boxdot(((T = 0) \Rightarrow ((n = 0) \Rightarrow (d = 0)))$
   $\wedge((T \neq 0) \Rightarrow ((n = 0) \Rightarrow (d = 0))))$

(3) follows from (2).

3. $\Box\boxdot((n = 0) \Rightarrow (d = 0))$

(4) follows from (3), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

4. $\Box((n = 0) \Rightarrow (d = 0)) \wedge \boxdot((n = 0) \Rightarrow (d = 0))$

This proves the derived rule. ∎

S9. $\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box((n = 0) \Leftrightarrow \neg\Diamond\varphi) \wedge \boxdot((n = 0) \Leftrightarrow \neg\Diamond\varphi)$

Proof :

By substituting for $\mathcal{N}(\varphi, n)$, we have

1. $\Box\boxdot(((T = 0) \Rightarrow (\neg\varphi \Leftrightarrow (n = 0)))$
   $\wedge((T \neq 0) \Rightarrow (\neg\varphi \wedge \oslash\neg\varphi \Rightarrow (n = \oslash n))))$

(2) follows from (1), the reverse computation induction principle.

2. $\Box\boxdot((\boxdot\neg\varphi) \Leftrightarrow (n = 0))$

(3) follows from (2), $\vdash \boxdot\varphi \Rightarrow \varphi$, $\Box$ Introduction (R3), $\vdash \Box(\varphi \Rightarrow \psi) \Rightarrow (\Box\varphi \Rightarrow \Box\psi)$ (F1) and Modus Ponens (R2).

3. $\Box((\neg\Diamond\varphi) \Leftrightarrow (n = 0)) \wedge \boxdot((\neg\Diamond\varphi) \Leftrightarrow (n = 0))$

This proves the derived rule. ■

S10. $\vdash \mathcal{N}(\varphi, n) \Rightarrow \Box(\boxdot\varphi \Rightarrow (n = 1)) \wedge \boxdot(\boxdot\varphi \Rightarrow (n = 1))$

Proof :

The proof is a slight variation of the proof for S9. ■

S11. $\vdash \mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \Rightarrow$
$(\Box((\varphi \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg\varphi \wedge (n = y) \wedge d \leq x + k)) \Rightarrow$
$\Box((\neg\varphi \wedge (n = y_1) \wedge (d = x_1)) \Rightarrow \Box((n = y_1 + 1) \Rightarrow d \leq x_1 + k)))$

Proof :

To prove the derived rule, we assume (1) and prove $\Box((\neg\varphi \wedge (n = y_1) \wedge (d = x_1)) \Rightarrow \Box((n = y_1 + 1) \Rightarrow d \leq x_1 + k))$.

1. $\mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \wedge \Box((\varphi \wedge (n = y) \wedge (d = x))$
$\Rightarrow \Diamond(\neg\varphi \wedge (n = y) \wedge d \leq x + k))$

To prove the conclusion, we apply the derived computation induction principle, i.e. we prove 2i and 2ii.

2i. $\mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \wedge \Box((\varphi \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg\varphi \wedge (n = y)$
$\wedge d \leq x + k)) \Rightarrow ((\neg\varphi \wedge (n = y_1) \wedge (d = x_1))$
$\Rightarrow \Box((n = y_1 + 1) \Rightarrow d \leq x_1 + k))$

2ii. $\mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \wedge \Box((\varphi \wedge (n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg\varphi \wedge (n = y)$
$\wedge d \leq x + k)) \Rightarrow \odot(\mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \wedge \Box((\varphi \wedge$
$(n = y) \wedge (d = x)) \Rightarrow \Diamond(\neg\varphi \wedge (n = y) \wedge d \leq x + k)))$

The proof of 2ii is straightforward. To prove 2i we assume (3) and prove $\Box((n = y_1 + 1) \Rightarrow d \leq x_1 + k)$

3. $\mathcal{N}(\varphi, n) \wedge \mathcal{D}(\varphi, d) \wedge \Box((\varphi \wedge (n = y) \wedge (d = x))$
$\Rightarrow \Diamond(\neg\varphi \wedge (n = y) \wedge d \leq x + k)) \wedge \neg\varphi \wedge (n = y_1) \wedge (d = x_1)$

(4) follows from (3) and the definition of $\mathcal{D}(\varphi, d)$.

4. $\Diamond((\varphi \wedge (n = y_1 + 1)) \Rightarrow (d = x_1))$

(5) follows from (3) and (4).

5. $\Diamond(\neg\varphi \wedge (n = y_1 + 1) \wedge (d \leq x_1 + k))$

(6) follows from (5), and the definitions of $\mathcal{N}(\varphi, n)$ and $\mathcal{D}(\varphi, d)$.

6. $\Box((n = y_1 + 1) \Rightarrow (d \leq x_1 + k))$

This proves the derived rule. ■

C1. $\vdash \mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(\neg\varphi, d_2) \wedge \mathcal{D}(true, d_3)$
$\Rightarrow \Box(d_1 + d_2 = d_3) \wedge \boxdot(d_1 + d_2 = d_3)$

Proof :

(1) follows from S4.

1. $\mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(\neg\varphi, d_2) \wedge \mathcal{D}(\varphi \vee \neg\varphi, d_3) \wedge \mathcal{D}(\varphi \wedge \neg\varphi, d_4)$
$\Rightarrow \Box(d_1 + d_2 = d_3 + d_4) \wedge \boxdot(d_1 + d_2 = d_3 + d_4)$

(2) follows from (1) and S1.

2. $\mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(\neg\varphi, d_2) \wedge \mathcal{D}(true, d_3)$
$\Rightarrow \Box(d_1 + d_2 = d_3) \wedge \boxdot(d_1 + d_2 = d_3)$

This proves the derived rule. ■

C2. $\vdash \mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(true, d_2) \Rightarrow \Box(d_1 \leq d_2) \wedge \boxdot(d_1 \leq d_2)$

Proof :

(1) follows from C1.

1. $\mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(true, d_2) \wedge \mathcal{D}(\neg\varphi, d_3)$
$\Rightarrow \Box(d_1 + d_3 = d_2) \wedge \boxdot(d_1 + d_3 = d_2)$

(2) follows from (1) and S3.

2. $\mathcal{D}(\varphi, d_1) \wedge \mathcal{D}(true, d_2) \Rightarrow \Box(d_1 \leq d_2) \wedge \boxdot(d_1 \leq d_2)$

This proves the derived rule. ■

This completes the proofs of all the derived rules of the calculus given in Section 3.4.

Bibliography

[ACD90] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real-time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[AFR80] K.R. Apt, N. Francez, and W.-P. de Roever. A proof system for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–384, 1980.

[AH89] R. Alur and T. A. Henzinger. A really temporal logic. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society Press, 1989.

[AH90] R. Alur and T. A. Henzinger. Real-time logics : Complexity and expressiveness. In *Proceedings of the 5th Annual on Logic in Computer Science*, pages 390–401. IEEE Computer Society Press, 1990.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. *Proceedings of the 3rd IEEE Symposium on Logic and Computer Science*, 1988.

[AL91] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. 1991.

[Apt81] K.R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[Bac89] R. J. R. Back. Refinement calculus, part II parallel and reactive programs. Technical Report 93, Abo Akademi, 1989.

[BB86] B. Banieqbal and H. Barringer. A study of an extended temporal language and a temporal fixed point calculus. Technical Report UMCS-86-10-2, Department of Computer Science, University of Manchester, Manchester, 1986.

[BB90] J.C.M. Baeten and J.A. Bergstra. Real-time process algebra. Technical Report P8916b, University of Amsterdam, Amsterdam, 1990.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th ACM Symposium on the Theory of Computing*, pages 51–63, Washington D.C., 1984.

[BKP85] H. Barringer, R. Kuiper, and A. Pnueli. A compositional temporal approach to a CSP-like language. In *Formal Models in Programming*, pages 207–227. North-Holland, Amsterdam, 1985.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 173–183, Florida, 1986.

[BMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, 1983.

[Boc82] G. V. Bochmann. Hardware specification with temporal logic : An example. *IEEE Transactions on Computers*, C-31:223–231, 1982.

[BvW89] R. J. R. Back and J. von Wright. Refinement calculus, part I sequential nondeterministic programs. Technical Report 92, Abo Akademi, 1989.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CH91] Zhou ChaoChen and M. R. Hansen. A note on completeness of the duration calculus. ProCos Working Paper, 1991.

[CHR91] Zhou ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[CHRR92] Zhou ChaoChen, M. R. Hansen, A.P. Ravn, and H. Rischel. Duration specifications for shared processors. In *Lecture Notes in Computer Science 571*. Springer-Verlag, Heidelberg, 1992.

[CM84] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions in Computer Systems*, 2(3):251–273, 1984.

[Dav91] J. Davies. *Specification and Proof in Real-Time Systems.* PhD thesis, Programming Research Group, Oxford University, Oxford, 1991.

[DS89] J. W. Davies and S. A. Schneider. Factorising proofs in timed CSP. In *Lecture Notes in Computer Science 439*, pages 232–252. Springer-Verlag, Heidelberg, 1989.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[EH86] E.A. Emerson and J.Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[Fis87] M. D. Fisher. *Temporal Logics for Abstract Semantics.* PhD thesis, Department of Computer Science, University of Manchester, Manchester, 1987.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 163–173, Las Vegas, 1980.

[Hay87] I.J. Hayes (Editor). *Specification Case Studies.* Prentice-Hall, London, 1987.

[HCS92] M. R. Hansen, Zhou Chaochen, and J. Staunstrup. A real-time duration semantics for circuits. To appear in Workshop on Timing Issues in the Specification and Synthesis of Digital Circuits, March 1992.

[HGR87] C. Huizing, R. Gerth, and W.-P. de Roever. Full abstraction of a real-time denotational semantics for an OCCAM-like language. In *Proceedings of the 14th ACM Symposium on Principles of Programmming Languages*, pages 223–238, Munich, 1987.

[HLP90] D. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, 1990.

[HMM83] J. Halpern, B. Moszkowski, and Z. Manna. A hardware semantics based on temporal intervals. In *Lecture Notes in Computer Science 154*, pages 278–291. Springer-Verlag, Heidelberg, 1983.

[HMP91a] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 353–366, 1991.

[HMP91b] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? 1991.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International U.K., 1985.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1991.

[HW89] J. Hooman and J. Widom. A temporal-logic based compositional proof system for real-time message passing. In *Lecture Notes in Computer Science 366*, pages 424–441. Springer-Verlag, Heidelberg, 1989.

[Jac83] M. Jackson. *System Development.* Prentice-Hall International, 1983.

[JG88] M. Joseph and A. Goswami. What's 'real' about real-time systems? Technical Report 123, Department of Computer Science, University of Warwick, Coventry, 1988.

[JM86a] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12:890–904, 1986.

[JM86b] F. Jahanian and A.K. Mok. A graph-theoretic model for timing analysis in Real-Time Logic. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 98–108, 1986.

[Jon86] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, London, 1986.

[JS88] F. Jahanian and D.A. Stuart. A method for verifying properties of modechart specification. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 12–21, Huntsville, Alabama, 1988.

[Koy89] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1989.

[Kro90] F. Kroger. On the interpretability of arithmetic in temporal logic. *Theoretical Computer Science*, 73:47–60, 1990.

[KSR+85] R. Koymans, R.K. Shyamasundar, W.-P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. In *Lecture Notes in Computer Science 193*, pages 167–190. Springer-Verlag, Heidelberg, 1985.

[KVR83] R. Koymans, J. Vytopil, and W.-P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 187–197, Montreal, 1983.

[Lam80] L. Lamport. "Sometime" is sometimes "not never": On the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, 1980.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, DEC Systems Research Center, 1991.

[LG81] G.M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15:281–302, 1981.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Lecture Notes in Computer Science 193*, pages 196–218. Springer-Verlag, Heidelberg, 1985.

[LRSZ92] Z. Liu, A. P. Ravn, E. V. Sorensen, and C. Zhou. A probabilistic duration calculus. Submitted to the 2nd International Workshop on Responsive Computing Systems, 1992.

[LS87] N. G. Leveson and J. L. Stolzy. Safety analysis using petri nets. In *IEEE Transactions on Software Engineering SE-13*, 1987.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, New York, 1989.

[MM83] B. Moszkowski and Z. Manna. Reasoning in interval temporal logic. In *Lecture Notes in Computer Science 164*, pages 371–383. Springer-Verlag, Heidelberg, 1983.

[MO81] Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H. T. Kung, B. Sproul, and G. Steele, editors, *VLSI Systems and Computations*, pages 203–212. Computer Science Press Inc., Rockville,Maryland, 1981.

[MP82] Z. Manna and A. Pnueli. Verification of concurrent programs: A temporal proof system. In *Proceedings of the 4th School on Advanced Programming*, Amsterdam, 1982.

[MP89a] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Lecture Notes in Computer Science 354*, pages 201–284. Springer-Verlag, Heidelberg, 1989.

[MP89b] Z. Manna and A. Pnueli. Completing the temporal picture. In *Lecture Notes in Computer Science 372*, pages 534–558. Springer-Verlag, Heidelberg, 1989.

[MS76] P. M. Merlin and A. Segall. Recoverability of communication protocols - implications of a theoretical study. In *IEEE Transactions on Communications*, pages 1036–1043, 1976.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *CONCURRENCY 90, Lecture Notes in Computer Science 458*, pages 401–415. Springer-Verlag, Heidelberg, 1990.

[Mul76] G. Mullery. CORE- a method for controlled requirement specification. In *Proceedings of the 4th International Conference on Software Engineering*, 1976.

[Nai92] Y. Naik. A temporal approach to requirements specification of real-time systems. In *Lecture Notes in Computer Science 571*, pages 341–361. Springer-Verlag, Heidelberg, 1992.

[NRSV89] X. Nicollin, J-L. Richier, J. Sifakis, and J. Voiron. ATP: An alegbra for timed processes. *Proceedings of the IFIP Working Conference on Programming Concepts and Methods*, pages 402–429, 1989.

[OG76] S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Ost89] J.S. Ostroff. *Temporal Logic for Real-time Systems.* Research Studies Press, 1989.

[Pet77] J. L. Peterson. Petri nets. *Computing Surveys*, 1977.

[PH88] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real time systems (extended abstract). In *Lecture Notes in Computer Science 331*, pages 84–98. Springer-Verlag, Heidelberg, 1988.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Lecture Notes in Computer Science 224*, pages 510–584. Springer-Verlag, Heidelberg, 1986.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, Massachusetts Institute of Technology, Massachusetts, 1974.

[Raz83] R. R. Razouk. The derivation of performance expressions for communication protocols from timed petri net models. Technical Report 211, University of California, Irvine, 1983.

[Rei85] W. Reisig. *Petri Nets : An Introduction.* Springer-Verlag, Berlin, 1985.

[RR86] G.M. Reed and A.W. Roscoe. A timed model for Communicating Sequential Processes. In *Lecture Notes in Computer Science 226*, pages 314–323. Springer-Verlag, Heidelberg, 1986.

[RR88] G.M. Reed and A.W. Roscoe. Metric spaces as models for real-time concurrency. In *Lecture Notes in Computer Science 298*, pages 331–343. Springer-Verlag, Heidelberg, 1988.

[Sch90] S. Schneider. *Correctness and Communication in Real-Time Systems.* PhD thesis, Programming Research Group, Oxford University, Oxford, 1990.

[Sif77] J. Sifakis. Petri nets for performance evaluation. In *Measuring, Modelling and Evaluating Computer Systems, Proceedings of the 3rd Symposium IFIP Working Group 7.3*, pages 75–93, Amsterdam, The Netherlands, 1977.

[SM81] A. Salwicki and T. Müldner. On the algorithmic properties of concurrent programs. In *Lecture Notes in Computer Science 125*, pages 169–197. Springer-Verlag, Heidelberg, 1981.

[Sou83] N. Soundararajan. Correctness proofs of CSP programs. *Theoretical Computer Science*, 24(2):131–141, 1983.

[Sza86] A. Szalas. Concerning the semantic consequence relation in first-order temporal logic. *Theoretical Computer Science*, 47:329–334, 1986.

[Yi90] W. Yi. Real-time behaviour of asynchronous agents. In *CONCURRENCY 90, Lecture Notes in Computer Science 458*, pages 502–520. Springer-Verlag, Heidelberg, 1990.

[YR92] V. Yodaiken and K. Ramamritham. Verification of a reliable protocol. In *Lecture Notes in Computer Science 571*, pages 193–215. Springer-Verlag, Heidelberg, 1992.

[ZHK91] P. Zhou, J. Hooman, and R. Kuiper. A compositional proof-system for real-time systems based on explicit clock temporal logic: Soundness and completeness. Technical Report 91/, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1991.

[ZRB85] J. Zwiers, W.-P. de Roever, and P.E.M de Boas. Compositionality and concurrent networks: Soundness and completeness of a proof system. In *Lecture Notes in Computer Science 194*. Springer-Verlag, Heidelberg, 1985.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof Theories for Networks of Processes, and their Connection.* PhD thesis, Eindhoven University of Technology, Eindhoven, 1988.