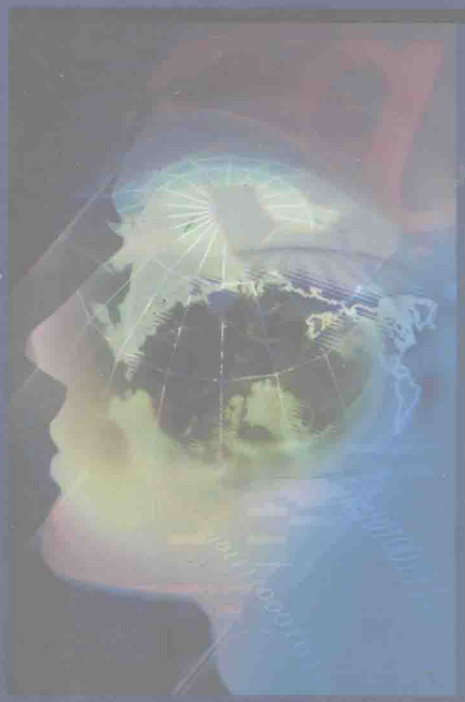
Temporal Logic and Temporal Logic Programming



Duan Zhenhua

School of Computer Science and Technology
Xidian University

科学出版社
www.sciencep.com

Temporal Logic and Temporal Logic Programming

Duan Zhenhua

School of Computer Science and Technology
Xidian University

Science Press

Responsible Editors: Li Na and Li Wei

Temporal Logic and Temporal Logic Programming

Published by Science Press
16 Donghuangchenggen North Street
Beijing 100717,China

Printed in Beijing

ISBN 7-03-016651-5/TP. 3158

Preface

I was involved in a research project concerning interval temporal logic and temporal logic programming funded by SERC in UK and did my Ph.D in the Department of Computing Science at the University of Newcastle upon Tyne from 1991 to 1993. I submitted my Ph.D thesis in the end of 1995 and obtained my Ph.D in July 1996.

This book is basically based on my Ph.D thesis [1] though some corrections and changes have been made in order to keep up with the development of this research area.

Temporal logic programming is a paradigm for specification and verification of concurrent programs in which a program can be written, and the properties of the program can be described and verified in a same notation. However, there are many aspects of programming in temporal logics that are not well-understood. One such an aspect is concurrent programming, another is framing and the third is synchronous communication for parallel processes.

This book extends the original Interval Temporal Logic (ITL) to include infinite models, past operators, and a new projection operator for dealing with concurrent computation, synchronous communication, and framing in the context of temporal logic programming.

The book generalizes the original ITL to include past operators such as previous and past chop, and extends the model to include infinite intervals. A considerable collection of logic laws regarding both propositional and first order logics is formalized and proved within model theory.

After that, a subset of the extended ITL is formalized as a programming language, called extended Tempura. These extensions, as in their logic basis, include infinite models, the previous operator, projection and framing constructs. A normal form for programs within the extended Tempura is demonstrated.

Next, a new projection operator is introduced. In the new construct, the sub-processes are autonomous; each process has the right to specify its own interval over which it is executed.

The book presents a framing technique for temporal logic programming, which includes the definitions of new assignments, the assignment flag and the framing operator, the formalization of algebraic properties of the framing operator, the minimal model semantics of framed programs, as well as an executable framed interpreter.

The synchronous communication operator *await* is based directly on the proposed framing technique. It enables us to deal with concurrent computation. Based on EITL and *await* operator, a framed concurrent temporal logic programming language, FTLL, is formally defined within EITL.

Finally, the book describes a framed interpreter for the extended Tempura which has been developed in SICSTUS prolog. In the new interpreter, the implementation of new assignments, the frame operator, the await operator, and the new projection operator are all included.

During the year when I worked in the Department of Computing Science at the University of Newcastle upon Tyne, many colleagues and friends supported and helped me for my research. By this opportunity, I would like to thank all of them. Especially,

I am grateful to my supervisor, Maciej Koutny, for his excellent guidance throughout many years and invaluable help with my thesis. Maciej's insistence on clear writing and notation has set a standard I always aim for. His detailed readings and numerous comments on the early different versions of the draft thesis have challenged me to form many ideas and to improve the contents of the final version.

I am greatly indebted to Ben Moszkowski for leading me to this research field, and for all of his support and advice as I was working with him for development of the framed interpreter.

I would also like to thank Roger Hale and Shinji Kono for some useful discussions. In particular, Kono's assistance in using Sicstus prolog during the period when I was developing the framed interpreter for Tempura is appreciated.

I am thankful to Shirley Craig for her patience and efficiency in searching out many relevant references for my research.

Thanks go to the two examiners, Michael Fisher and John Fitzgerald, who discovered many typos in an earlier version of my thesis and suggested a number of changes to improve the thesis.

Many thanks also go to Tom Anderson, Brian Randell, Santosh Shrivastava, and John Lloyd for their support and help.

I wish to acknowledge the help of my students at Xidian University for their detailed comments and helpful criticisms. In particular, Tian Cong helped with the technical preparation of the manuscript. She spent long hours patiently preparing the format of different versions of this book.

The research behind this book is now supported by NSFC under Grant 60373103 and Grant 60433010, and SRFDP under Grant 2003701015. I am particularly thankful to National Natural Science Foundation of China, Science and Technology Development Center of the Education Department of China.

Contents

Chapter 1

Introduction

This book extends the Interval Temporal Logic (ITL)[2] to include infinite models, past operators, a new projection operator for dealing with concurrent computation, synchronous communication, and framing in the context of temporal logic programming.

1.1 Temporal Logic

In the past three decades temporal logic has emerged as a formalism for specification and verification of reactive systems. Temporal logic is a branch of modal logic which has been studied for a long time[3, 4, 5, 6, 7, 8]. Modal logic deals with two propositional operators $\Box$ (always) and $\Diamond$ (sometimes) besides the usual logic connectives such as $\wedge, \vee, \rightarrow$ and $\leftrightarrow$ etc. interpreted as "necessarily" and "possibly". This is based on the idea that the truth of an assertion is a relative notion depending on possible worlds. A formal semantics of modal logic is given by Kripke[6]. To model a system, *kripke structures* and *labeled transition systems* (LTS) are normally employed [6, 9].

There are a number of temporal logics in literature[2, 10, 11, 12, 13, 14]. They can be divided into two categories, linear-time and branching-time logics. Linear-time logics are concerned with properties of paths while branching-time logics describe properties depending on the branching of computational structures.

Within the family of branching-time logics, there are three formalisms which received particular attention: Hennessy-Milner Logic (HML), Modal $\mu-$Calculus and Computational Tree Logic (CTL). HML is a simple modal logic introduced by Hennessy and Milner[15, 16]. It is defined over a given set, *Act*, of actions, ranged over by a. Formulas are constructed according to the following grammar:

$$\phi ::= true \mid false \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \langle a \rangle \phi$$

The most interesting operators of HML are the branch time modalities $[a]$ and $\langle a \rangle$. They relate a state to its $a-$successors. $[a]\phi$ holds for a state if all $a-$successors satisfy formula ϕ while $\langle a \rangle \phi$ holds if there exists an $a-$successor satisfying formula ϕ.

The modal $\mu-$calculus[17] is a branching temporal logic which extends Hennessy-Milner Logic by fixpoint operators. Let *Act* be a set of actions and Var a set of variables. Modal

$\mu-$calculus formulas are constructed according to the following grammar:

$$\phi ::= true \mid false \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

Here, X ranges over Var and a over Act. The two fixpoint operators μX and νX, bind free occurrences of variable X. Modal $\mu-$calculus formulas are interpreted over labeled transition systems. Given an LTS $T = (S, Act, \rightarrow)$, we interpret a closed formula, ϕ, as that subset of S whose states make ϕ true. The formulas, $[a]\phi$ and $\langle a \rangle \phi$, can be interpreted as in HML. The least fixpoint formula, $\mu X.\phi$, is interpreted by the smallest subset x of S that recurs when ϕ is interpreted with the substitution of x for X while the greatest fixpoint formula, $\nu X.\phi$, is interpreted by the largest such set.

The syntax of Computation Tree Logic[12] looks as follows:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid AU(\phi_1, \phi_2) \mid EU(\phi_1, \phi_2) \mid AF(\phi) \mid EF(\phi) \mid AG(\phi) \mid EG(\phi)$$

CTL has the six modalities AU, EU, AF, EF, AG and EG. Each takes the form QL, where Q is one of the paths quantifiers A (all) and E (existence) while L is one of the linear-time modalities U (until), F (sometimes) and G (always). The path quantifier provides a universal (A) or existential (E) quantification over the paths emanating from a state, and on these paths the corresponding linear-time property must hold.

There are many types of linear-time logics in literature [10, 11, 12]. Lamport defines the temporal logic of actions (TLA) which includes temporal operators $\Box$ and $\Diamond$ without the next operator [10]. Instead, he employs x' to denote the new value of x. TLA is a linear logic for specifying and reasoning about concurrent systems in which systems and their properties can be represented in the same logic. A typical linear-time logic is given by Manna and Pnueli [11]. The following is its syntax:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \bigcirc \phi \mid U(\phi_1, \phi_2) \mid F(\phi) \mid G(\phi)$$

With this logic, the temporal operators are $\bigcirc$ (next), U (until), F (sometimes) and G (always). The formulas are interpreted over *Kripke structures.*

Among linear-time temporal logics, there exist a number of logics based on the chop (;) operator [18, 19]. We call these logics choppy logics. The chop operator is different from the conventional temporal operators $\Box$ and $\bigcirc$. A formula, $p; q$, holds over a path (or interval) if and only if the path can be split into two parts such that p holds on the first part and q holds on the second part. Chop was first introduced as a temporal construct by Harel, Kozen and Parikh[19]. It was considered in more detail by Chandra, Halpern, Meyer and Parikh[20]. Halpern, Manna and Moszkowski used chop to facilitate reasoning about time-dependent digital hardware[25]. Subsequently, Moszkowski formalized a temporal logic based on the chop, chop star, next, and projection operators[2]. This linear temporal logic is now called the Interval Temporal Logic (ITL). It is a choppy logic based on finite intervals of time. Within this logic, in the first order case, the chop star operator can be defined by means of the chop operator. Several researchers have developed ITL extensions for hybrid systems. Zhou, Hoare and Ravn[22] formalized a real-time logic called Duration Calculus (DC) for hybrid systems. Duan, Holcombe and Bell generalized ITL to a Hybrid Projection Temporal Logic (HPTL) for hybrid systems [23, 24].

Several researchers have investigated choppy logic for infinite time. Rosner and Pnueli formalized a propositional choppy logic which includes chop [18], next and until operators in 1986. The formulas are interpreted over finite and infinite time intervals. Paech [26] defined a choppy logic with chop, chop star (restricted), next and unless operators in 1988. The formulas are interpreted over infinite time intervals. Dutertre [27] gave first order chop logic with chop, next operators, but the formulas are interpreted over only finite time. Wang Hanpin and Xu Qiwen generalized this logic to infinite time intervals in 1999 [28].

Within the ITL developments, Duan, Koutny and Holt introduced a new projection construct, $(p_1, ..., p_m)$ *prj* q, and generalized ITL to infinite time intervals in 1994[29]. The new projection operator, *prj*, can subsume the chop and original projection operator *proj*[30]. The subsequent work[1, 32, 33] generalized ITL to Projection Temporal Logic (PTL) with infinite time intervals. Moszkowski extended the axioms systems for the finite intervals PITL and ITL[2] to projection and infinite time intervals in 1995[30].

Within the choppy logic community, several researchers have looked at decision procedures and axioms systems for the variations of choppy logics. Rosner and Pnueli[18] presented an axiom system for a propositional choppy logic with chop, next and until, and based the completeness proof on tableaux-based decision procedure. Paech[26] formalized a complete Gentzen-style proof system over finite intervals with the temporal operators chop, chop star and unless. Dutertre[27] presented two complete proof systems for the first order choppy logic over finite time with the temporal operators chop and next. Wang Hanpin and Xu Qiwen generalized this to infinite time[28].

Halpern and Moszkowski[25, 34] proved the decidability of quantifier-free propositional ITL (QPITL) over finite time. Kono presented a tableaux-based decision procedure for QPITL with projection[35]. However, no formal proof was given that all models were considered. It appears that Bowman and Thompson[36] presented a first tableaux-based decision procedure for quantifier-free propositional ITL over finite intervals with projection. Subsequently, they presented a completeness proof for an axiomatization of this logic[37].

Moszkowski[38] presented axioms systems over finite intervals for the propositional ITL and first order ITL. The propositional part is claimed to be complete but only an outline of a proof was given. Later work extended this for projection with infinite time[30]. In 2000, Moszkowski formalized a complete axiomatization of ITL with infinite time[39].

1.2 Temporal Logic Programming

Temporal logic has been proposed for the purpose of verifying properties of programs. However, the verification of programs has suffered from the convention that different languages (and thus different semantic domains) have been used for writing programs, writing about their properties, and writing about whether and how a program satisfies a given property [10]. One way to simplify this is to use the same language in each case, as far as possible.

It has therefore been suggested that a subset of a temporal logic be used as the foundational basis for a programming language [2, 41, 42]. This has led to the definition of a number of programming languages based on temporal logics [2, 3, 43, 44, 45, 46].

One of the earliest temporal logic programming languages, XYZ/E [43], is based on linear time temporal logic proposed by Manna and Pnueli [48]. Furthermore, XYZ system consists of a temporal logic programming language XYZ/E as its basis, and a group of CASE tools to support various kinds of methodologies [49].

Another temporal logic programming language, Tempura [2], which we are particularly interested in, is based on a subset of interval temporal logic whose formulas can be interpreted as a traditional imperative program. In logic terms, executing a Tempura formula (program) amounts to building a model for the formula.

Gabbay developed the language USF [3], which follows an imperative future approach. The METATEM language [50, 51] is a development of USF consisting of a larger range of operators, a better defined execution mechanism [52] and a more practical normal form [53]. A METATEM program for controlling a process is presented as a collection of temporal rules. The rules apply universally in time and determine how the process progresses.

TOKIO [44] is a logic programming language based on the extension of Prolog with ITL formulas. It provides a useful system in which a range of applications can be implemented and verified. TOKIO supports an extended subset of ITL incorporating the non-deterministic operators "$\Diamond$" and "$\vee$".

The temporal logic programming languages [45, 46] are based on the logic programming paradigm and view an execution of a program as a refutation proof. Many other temporal logic programming languages can be found in literature[4, 5, 47, 54, 55, 56, 57].

An interpreter written in C for Tempura was developed by Hale [58]. He also investigated how to use Tempura in programming. Many samples illustrating how to model the structure and behaviour of hardware and software systems in a unified way can be found in literature[2, 58].

However, there are many aspects of programming in temporal logics that are not well understood (at least in Tempura). One such an aspect is concurrent programming, another is the problem of framing, and the third is synchronous communication for parallel processes.

In a temporal logic programming language such as Tempura, the conjunction and parallel composition (see Chapter 4) are basic operators for concurrent programming. However, the conjunction seems appropriate for dealing with fine-grained parallel operations that proceed in lock step; while the parallel composition, on the other hand, permits the combined processes to specify their own intervals. Thus it is better suited to the coarse-grained concurrency of a typical multiprocessor, where each process proceeds at its own speed. However, processes combined through the parallel composition operator share all the states and may interfere with one another. It is therefore necessary for us to investigate other possible ways to handle concurrent programming.

Framing techniques have been employed by conventional imperative languages for many years. However, framing in conventional languages has often been taken for granted. Nevertheless, we have to consider this option carefully in temporal logic programming. Framing is concerned with how the value of a variable from one state can be carried to the next. Temporal logic offers no solution in this respect; no value from a previous state is assumed to be carried. Therefore, if we want the value of a variable to be inherited, we have to repeatedly assign the value to the variable from state to state. This is not

only tedious but also may decrease the efficiency of the program. Moreover, synchronous communication can not be handled without framing in temporal logic programming (see below).

Another problem that must be dealt with in temporal logic programming is that of communication between concurrent processes. Some models of concurrency involve shared (programming) variables, some involve synchronous message passing, and some involve asynchronous channels e.g. CCS [59, 60], CSP [61, 62]. In temporal logic programming languages such as XYZ/E [43, 49] and Tempura [2], communication between parallel components is based on shared variables.

To synchronize communication between parallel processes in a concurrent program (e.g. solving the mutual exclusion problem) with the shared variables model, a synchronization construct, $await(c)$ or some equivalent is required, as in many concurrent programming languages [11]. The meaning of $await(c)$ is simple: it changes no variables, but waits until the condition c becomes true, at which point it terminates.

Modelling an $await(c)$ in a temporal logic requires a kind of indefinite stability, since it cannot be known at the point of use how long the wait will be; but it must also allow variables to change, so that an external process can modify the boolean parameter and it can eventually become true. Solving this problem also requires some kind of framing operation.

1.3 Description of Book

To deal with framing, synchronization and communication, and concurrent programming, an extend interval temporal logic (EITL) is formalised. The main extensions are made in two aspects: one is that the past operators such as previous and past chop (the counterpart of chop in the future) operators can be used; the other is that infinite models are permitted. (Of course, projection is an extension to ITL but it is treated as another topic). The reason for introducing past operators is that a framed program may involve immediate assignments which require the previous operator for reducing the program in an operational manner. The infinite intervals are needed because we are concerned with reactive systems. The extensions are not trivial. In some sense, we generalise ITL from an interval-based notation to a point-based notation since we refer to no explicit subintervals but points over a fixed interval. The extended logic systems are divided into two parts: propositional and first order logics.

Chapter 2 introduces the extended propositional interval temporal logic (EPITL). First, the syntax and semantics of the underlying logic are presented, then the fundmental logic laws concerning the temporal operators, both future and past, are formalized and proved. These logic laws also provide a basis for the first order EITL.

Chapter 3 presents the first order extended interval temporal logic (EITL). The logic laws regarding variables, functions, predicates, equality, and quantifications, in addition to its syntax and semantics are presented. These logic laws, as a foundation, allow us to prove some useful properties of programs, to capture the temporal semantics of a framed program.

Chapter 4 formalizes a programming language which is an executable subset of the extended logic system and an extension of Tempura. Within this language, a variable

can refer to its previous value, as well as its next value as in the original Tempura. The computation trace of a program can also be infinite. These extensions enable us to handle a concurrent computation for a reactive system.

A program p with the extended language has the normal form, $\bigvee_{i=1}^{l} p_{ei} \wedge empty \vee \bigvee_{j=1}^{t} p_{cj} \wedge \bigcirc p_{fj}$, where p_{ei} and p_{cj} are state formulas consisting of equalities; whereas p_{fj} is an internal program. This conclusion is proved by induction on the structure of programs. It facilitates capturing temporal semantics and further reducing programs.

Chapter 5 introduces a new projection operator, $(p_1, ..., p_m)\ prj\ q$, which can be thought of as a combination of the parallel ($\|$) and the original projection ($p\ proj\ q$) operators in Tempura. The motivation for introducing the new projection construct is that we intend to give a more flexible parallel operator in temporal logic programming.

Intuitively, $(p_1, ..., p_m)\ prj\ q$ means that q is executed in parallel with $p_1; ...; p_m$ over an interval obtained by taking the endpoints (rendezvous points) of the intervals over which $p_1, ..., p_m$ are executed. The new projection construct permits the processes, $p_1, ..., p_m, q$, to be autonomous, each process having the right to specify the interval over which it is executed. In particular, the sequence of processes $p_1, ..., p_m$ and q may terminate at different time points. Although the communication between processes is still based on shared variables, the communication and synchronization take place only at the rendezvous points (global states), otherwise they are executed independently.

In the chapter, a considerable set of logic laws regarding the projection construct are formalized and proved. The normal form of the projection construct is also proved. These logic laws and the normal form allow us to reduce the projection statement in temporal logic programming. Finally, an example is given to illustrate how to reduce a projection statement.

Chapter 6 discusses the framing issue. Framing is difficult to handle within a logic system. It is well known that the first order logic is monotonic. That is, adding a formula to a theory has the effect of strictly increasing the set of formulas that can be inferred. However, the framing issue is intrinsically non-monotonic. Indeed, adding a new positive fact, i.e. an explicit assignment, to a set of positive facts with the framing operator has a 'side effect': the negation of the fact cannot be inferred from the previous set.

To work out an executable framed temporal logic programming language such as framed Tempura with mixed framed and non-framed variables is not straightforward. First, we find assignment operators within Tempura are inadequate to deal with framing and thus new assignment operators must be defined. Second, the non-monotonicity makes a framed program radically shift in its semantics with respect to the one without framing. Framed programs are no longer well interpreted within the normal logic model we use. Therefore, some new models are required.

In the chapter therefore a new assignment operator ($\Leftarrow$) and an assignment flag (af) are defined within the extended logic framework. Armed with the assignment flag, a framing operator $frame(x)$ is formalized. These new constructs are interpreted within a minimal model semantics.

This allows us to specify framing status of variables throughout an interval in a flexible manner. However, introducing the framing operator destroys the monotonicity, and leads

to a default logic [63, 64, 65]. Therefore, negation by default has to be used to manipulate the framing operator.

To illustrate framing techniques, a number of examples are given within different program constructs including the sequential, conjunction, parallel, projection and the mixed cases. These examples show us the framing operator can be used in a flexible manner to facilitate framing in different program constructs.

Chapter 7 introduces minimal model semantics to interpret framed programs. As mentioned earlier, when a framing technique is introduced to temporal logic programming, the semantics of a program may be changed. So, one issue we have to face is how to interpret a framed program. That is, how to capture the intended meaning of a program. In logic programming languages such as Prolog, negation by failure has been used in programs, and a program is interpreted by the minimal model or fixed point semantics as in literature[66]. This leads us to introduce a similar idea in temporal logic programming. To interpret framed programs, the minimal model is developed in detail. As a result, the existence of a minimal model of a framed program is proved under the assumption that the program has at least one finite model or has finitely many models. Two important logic laws concerning substitution are formalized and proved. A normal form for framed programs is also presented.

The framing operator enjoys some nice algebraic properties such as equivalency, distributivity, absorptivity, and idempotency etc. These algebraic laws are very useful for the reduction of a framed program. Many reduction rules of the interpreter developed by me recently employ these laws. In this chapter, some of algebraic properties of framing operators are characterized and proved.

Finally, an example is given to show how to use the logic laws and the minimal model to reduce a framed program.

Chapter 8 discusses synchronous communication in temporal logic programming. With the framing operator, the synchronous communication construct, $await(c)$, can easily be defined. Therefore, real concurrent programs can be managed within our system. In the book, we present a general framed concurrent temporal logic programming language FTLL which is similar to the language presented in literature[11] except that the concurrent computation model is true concurrency [67, 68] for ours but interleaving for theirs. The important difference is that our language is formally defined within the logic framework whereas their language is semi-formal. Of course, the language FTLL is a non-deterministic programming language. To deal with non-determinacy, we adopt Dijkstra's guarded language. As an illustration, two examples of programs within FTLL are given: one is the program solving the well known producer-consumer problem, and the other is the program filling an even order magic square problem.

Chapter 9 briefly introduces the new interpreter for the extended Tempura. To implement the previous operator, projection, await and framing constructs, a new framed interpreter has been developed in SICSTUS Prolog. However, we do not intend to present the interpreter in detail; only a brief explanation about implementation is provided, and some relative reduction algorithms are described.

Chapter 10 draws some conclusions.

In short, the main contribution of this book is in the following six respects:

1) The extended interval temporal logics, both propositional and first order, are new. The book generalizes the original ITL [2] by adding past operators such as previous ($\ominus$), and past chop ($\overleftarrow{;}$), and by extending the model to an infinite case. The extension changes the logic, in some sense, from an interval-based temporal logic to a point-based one. A considerable collection of logic laws regarding both propositional and first order logics is formalized and proved in detail within model theory.

2) A subset of the extended ITL is formalized as a programming language, called extended Tempura. These extensions, as in EITL, include infinite models and the previous operator. The normal form for the programs within the extended Tempura is firstly proved in a formal way based on the logic laws we give.

3) A new projection operator, $(p_1, ..., p_m)prj\ q$, is generalised from Moszkowski's projection $p\ proj\ q$, but the semantics is different. The construct $p\ proj\ q$ requires that process p be repeatedly executed over an interval and q be executed at endpoints of each subinterval on which p is executed, but the termination is controlled by q. The new construct, $(p_1, ..., p_m)\ prj\ q$, is treated as a combination of parallel and projection computations. The processes p_1, ..., p_m, q, are autonomous, each process has the right to specify its own interval over which it is executed. Although the process q is executed in a parallel way with the process $p_1; ...; p_m$, the communication between them is only at rendezvous states, and the processes may terminate at different time points.

4) The framing technique presented in the book is entirely our own work. It is a new methodology for temporal logic programming, which includes the definitions of new assignments ($<=, :=^+, o=^+, \leftarrow^+$), the assignment flag ($af$), and the framing operator ($frame$), the formalization of algebraic properties of the framing operator, the minimal model semantics of framed programs, as well as an executable framed interpreter.

5) The synchronous communication operator *await* is a natural consequence of our framing technique. It enables us to deal with the real concurrent computation such as one solving producer-consumer. Based on EITL and *await* operator, a framed concurrent temporal logic programming language, FTLL, is formally defined within EITL. The programs in FTLL, of course, have to be interpreted with the minimal model.

6) The framed interpreter for the extended Tempura has been developed in SICSTUS prolog. In the new interpreter, the implementation of new assignments, the frame operator, the await operator, and the new projection operator are all included. It is not complete but workable. The development of the framed interpreter is also our own work.

Chapter 2

Propositional Temporal Logic

Summary: An extended propositional interval temporal logic (EPITL) is formalized. The syntax and semantics of EPITL as well as some derived formulas are presented. Furthermore, a collection of logic laws is investigated in the underlying logic.

Temporal logic, like the classical first order logic, is formalized in two parts: propositional and first order. We first present an extended propositional ITL (EPITL). This later provides a basis for the first order extended ITL (EITL). The extension is made in two respects: a model can be an infinite interval and a temporal operator can be a past operator in the underlying logic.

This chapter is organized as follows: Section 2.1 presents the syntax of EPITL. Section 2.2 presents the semantics of EPITL. To this end, first, states and intervals are defined. Then interpretations of terms and formulas are given in detail. Section 2.3 defines satisfaction and validity of formulas in EPITL. In Section 2.4, some useful derived formulas are given. Section 2.5 gives the precedence rules of operators. Section 2.6 defines strong and weak equivalence relations as well as strong and weak implication relations. In Section 2.7, a number of logic laws are formalized and proved.

2.1 Syntax

The extended propositional ITL basically consists of propositional logic with modal constructs to reason about intervals of time. The modal constructs include both future and past operators. Let Z denote all integers, N all positive integers, and N_0 all non-negative integers.

1. Alphabet

   1) A denumerable set *Prop* of atomic propositions.

   2) The symbols $\neg, \wedge, \bigcirc, ;, \ominus, \bar{;}, +$.

2. Inductive Definition of Formulas