AN EXTENDED INTERVAL TEMPORAL LOGIC AND A FRAMING TECHNIQUE FOR TEMPORAL LOGIC PROGRAMMING

Submitted by Zhenhua Duan
to the University of Newcastle upon Tyne
as a thesis for the degree of
Doctor of Philosophy
in the Faculty of Science

The University of Newcastle upon Tyne
Newcastel upon Tyne, United Kingdom
May, 1996.

I certify that all the material in this thesis which is not my own work has been identified and that no material is included for which a degree has previously been conferred upon me.

Zhenhua Duan

To My Motherland

To My Parents

To Junru and Feng

Abstract

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 thesis 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 thesis 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 thesis 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 thesis 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.

Acknowledgements

I am grateful to my supervisor, Maciej Koutny, for his excellent guidance throughout the past few years and invaluable help with this 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.

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

Contents

Chapter 1

Introduction

This thesis extends the Interval Temporal Logic (ITL)[61] 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 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 [53]. 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 e.g. [79, 81, 61]. This has led to the definition of a number of programming languages based on temporal logics [82, 61, 32, 3, 35, 87].

One of the earliest temporal logic programming languages, XYZ/E [82], is based on linear time temporal logic proposed by Manna and Pnueli [63]. Moreover, 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 [83].

Another temporal logic programming language, Tempura [61], 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 [35], which follows an imperative future approach. The METATEM language [9, 29] is a development of USF consisting of a larger range of operators, a better defined execution mechanism [31] and a more practical normal form [30]. 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 [32] 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 [3, 87] 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 [36, 14, 80, 48, 55, 70, 71].

An interpreter written in C for Tempura was developed by Hale [39]. 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 [61] and [39].

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 [58, 59], CSP [44, 45]. In temporal logic programming languages such as XYZ/E [82, 83] and Tempura [61], 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 [66]. 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.

1.2 Description of Thesis

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, $(p_1, ..., s_m)$ *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 [76, 56, 57]. 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 [13]. 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 thesis, we present a general framed concurrent temporal logic programming language FTLL which is similar to the language presented in [66] except that the concurrent computation model is true concurrency [49, 12] 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 thesis is in the following six respects:

1. The extended interval temporal logics, both propositional and first order, are new. The thesis generalizes the original ITL [61] by adding past operators such as previous ($\ominus$), and past chop ($\bar{;}$), 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 thesis is entirely our own work. It is a new methodology for temporal logic programming, which includes the definitions of new assignments ($<=, :=^+, \circ=^+, \leftarrow^+$), 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 $\neg, \wedge, \bigcirc, \Diamond, ;, \ominus, \diamondminus, \bar{;}, +$.

2. Inductive Definition of Formulas

   1) Every proposition $p \in Prop$ is a formula.

   2) If p, q are formulas, so are the constructs

$$\neg p, p \wedge q, \bigcirc p, \Diamond p, p ; q, \ominus p, \diamondminus p, p \,\bar{;}\, q, p^{+}.$$

$\bigcirc$ (read next), $\Diamond$ (read sometimes), and ; (read chop) are the basic future (temporal) operators whereas $\ominus$ (read previous), $\Diamond$ (read sometimes in the past), and $\bar{;}$ (read chop in the past) are the basic past (temporal) operators. The chop plus (+) operator is introduced for the purpose of easily constructing *while*-loop (Chapter 4).

Here are some sample formulas:

$$p, \neg p, p \wedge q, \bigcirc(p;q), p\bar{;}\ominus q, \bigcirc\ominus p.$$

2.2 Semantics

1. States

   Following the definition of Kripke's structure [50], we define a *state* s over *Prop* to be a mapping from *Prop* to $B = \{\text{true, false}\}$:

   $s : Prop \longrightarrow B$

   We will use $s[p]$ to denote the valuation of p at the state s.

2. Intervals

   An *interval* σ is a non-empty sequence of states, which can be finite or infinite. The length, $|\sigma|$, of σ is ω if σ is infinite, and the number of states minus 1 if σ is finite. To have a uniform notation for both finite and infinite intervals, we will use *extended integers* as indices. That is, we consider the set N_0 of non-negative integers and ω,

   $$N_\omega = N_0 \cup \{\omega\}$$

   and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega,\omega)\}$. To simplify definitions, we will denote σ as $< s_0, ..., s_{|\sigma|} >$, where $s_{|\sigma|}$ is undefined if σ is infinite. With such a notation, $\sigma_{(i..j)}$ $(0 \leq i \preceq j \leq |\sigma|)$ denotes the sub-interval $< s_i, ..., s_j >$ and $\sigma^{(k)}$ $(0 \leq k \preceq |\sigma|)$ denotes $< s_k, ..., s_{|\sigma|} >$.

3. Interpretations

   An interpretation is a quadruple $\mathcal{I} = (\sigma, i, k, j)$, where σ is an interval, i, k integers, and j an integer or ω such that $i \leq k \preceq j \leq |\sigma|$. We use the notation $(\sigma, i, k, j) \models p$ to mean that some formula p is interpreted and satisfied over the subinterval $< s_i, ..., s_j >$ of σ with the current state being s_k.

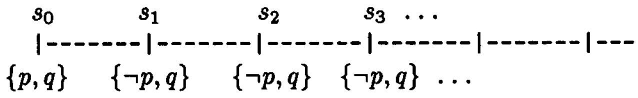
   The satisfaction relation ($\models$) is inductively defined as follows:

I − prop	$\mathcal{I}$	$\models$	p iff $s_k[p]$ =true, for any given proposition p.
I − not	$\mathcal{I}$	$\models$	$\neg p$ iff $\mathcal{I} \not\models p$.
I − and	$\mathcal{I}$	$\models$	$p \wedge q$ iff $\mathcal{I} \models p$ and $\mathcal{I} \models q$.
I − next	$\mathcal{I}$	$\models$	$\bigcirc p$ iff $k < j$ and $(\sigma, i, k+1, j) \models p$.

$I - som$	$\mathcal{I} \models$	$\Diamond p$ **iff there exists r such that $k \leq r \preceq j$ and $(\sigma, i, r, j) \models p$.**
$I - chop$	$\mathcal{I} \models$	$(p; q)$ **iff there exists r such that $k \leq r \preceq j$ and $(\sigma, i, k, r) \models p$ and $(\sigma, r, r, j) \models q$.**
$I - prev$	$\mathcal{I} \models$	$\ominus p$ iff $i < k$ *and* $(\sigma, i, k-1, j) \models p$.
$I - somp$	$\mathcal{I} \models$	$\Diamond p$ iff there exists l such that $i \leq l \leq k$ and $(\sigma, i, l, j) \models p$.
$I - chopp$	$\mathcal{I} \models$	$(p \bar{;} q)$ iff there exists l such that $i \leq l \leq k$ and $(\sigma, i, l, l) \models p$ and $(\sigma, l, k, j) \models q$.
$I - plus$	$\mathcal{I} \models$	p^+ iff there are finitely many $r_0, ..., r_n \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \leq r_{n-1} \preceq r_n = j$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $1 < l \leq n, (\sigma, r_{l-1}, r_{l-1}, r_l) \models p$; or $j = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq r_2 \leq ...$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

Note that, in the interpretation of p^+, the case with infinitely many partition points over a finite interval is not permitted because this would imply infinitely many partitions at one state. Therefore, only two cases in the interpretation of p^+ are permitted: one in which there are finitely many partition points over a finite or infinite subinterval $\sigma_{(i..j)}$; and the other case in which there are infinitely many partition points over an infinite subinterval $\sigma_{(i..j)}$. With the latter, we require that the integer sequence in the correspondence with partition points be unbounded.

Example 2.1 Let A denote $(\bigcirc(\ominus(p \wedge q) \bar{;} (\neg p))); (\Diamond q)$. We claim that $(\sigma, 0, 0, |\sigma|) \models A$, where σ is given in Fig. 2.1. Note that each state is associated with a set of propositions and/or negations of propositions to mean that a proposition p is true at the state if p is in the set, and is false if $\neg p$ is in the set.



s_0 s_1 s_2 s_3 ...
$\{p, q\}$ $\{\neg p, q\}$ $\{\neg p, q\}$ $\{\neg p, q\}$ ...

Fig. 2.1 An interval

Proof

$(\sigma, 0, 0, |\sigma|) \models \bigcirc(\ominus(p \wedge q) \bar{;} (\neg p)); (\Diamond q)$ if and only if there exists $r, 0 \leq r \preceq |\sigma|$ such that $(\sigma, 0, 0, r) \models \bigcirc(\ominus(p \wedge q) \bar{;} (\neg p))$ and $(\sigma, r, r, |\sigma|) \models \Diamond q$. Let $r = 2$. Then $(\sigma, 2, 2, |\sigma|) \models \Diamond q$ is obvious because $s_k[q]$=true for all $k \geq 2$. So we need only to prove $(\sigma, 0, 0, 2) \models \bigcirc(\ominus(p \wedge q) \bar{;} (\neg p))$.

$$(\sigma, 0, 0, 2) \models \bigcirc(\ominus(p \wedge q) \bar{;} (\neg p))$$
$$\Longleftrightarrow (\sigma, 0, 1, 2) \models \ominus(p \wedge q) \bar{;} (\neg p)$$
$$\Longleftrightarrow \text{there exists } l, 0 \leq l \leq 1 \text{ such that } (\sigma, 0, l, l) \models \ominus(p \wedge q) \text{ and } (\sigma, l, 1, 2) \models (\neg p).$$

Let $l = 1$. We have

$$\begin{aligned}
& (\sigma,0,1,1)\models\circleddash(p\wedge q) \\
\Longleftrightarrow\ & (\sigma,0,0,1)\models p \text{ and } (\sigma,0,0,1)\models q \\
\Longleftrightarrow\ & s_0[p] = \text{true and } s_0[q] = \text{true} \\
\Longleftrightarrow\ & \mathit{true}
\end{aligned}$$

and

$$\begin{aligned}
(\sigma,1,1,2)\models(\neg p) \Longleftrightarrow\ & s_1[\neg p] = \text{true} \\
\Longleftrightarrow\ & s_1[p] = \text{false} \\
\Longleftrightarrow\ & \mathit{true}.
\end{aligned}$$

Therefore

$$(\sigma,0,0,|\sigma|)\models\bigcirc(\circleddash(p\wedge p)\,\overline{;}\,p);\Diamond q.$$

$\square$

2.3 Satisfaction and Validity

A model is an interval which can be finite or infinite.

1. Satisfaction and Validity

   A formula p is satisfied by an interval σ, denoted by $\sigma\models p$, if $(\sigma,0,0,|\sigma|)\models p$. A formula p is called satisfiable if $\sigma \models p$ for some σ. A formula p is valid, denoted by $\models p$, if $\sigma\models p$ for all σ.

2. Special Model

   In some cases, we need to define a set R of models satisfying some properties. A formula p is called R-satisfiable, if there exists a model $\sigma\in R$ such that $\sigma\models p$. A formula p is said to be R-valid if for all intervals σ in R, $\sigma \models p$. In this case, the satisfaction relation $\models_R$ will be used instead of $\models$.

Example 2.2 Let A denote $(\bigcirc\circleddash p)\leftrightarrow p$ (see Section 2.4 for the definition of $\leftrightarrow$). A is satisfiable but not valid because $< s_0 > \not\models A$. However, A is R-valid with $R = \{\sigma \mid |\sigma| > 0\}$.

$\neg(\circleddash\bigcirc p \rightarrow \bigcirc\circleddash p)$ is not satisfiable because $\circleddash\bigcirc p \rightarrow \bigcirc\circleddash p$ is valid. $\square$

2.4 Abbreviations

The abbreviations *true*, *false*, $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $\mathit{true} \overset{\text{def}}{=} p \vee \neg p$ and $\mathit{false} \overset{\text{def}}{=} p \wedge \neg p$ for any formula p. Hence, for any interpretation $\mathcal{I}$, $\mathcal{I} \models \mathit{true}$ and $\mathcal{I} \not\models \mathit{false}$. Furthermore, we use the following abbreviations:

1. Derived future operators

   1) *empty*

   Informally, *empty*, read as empty in the future, means that the current state is the final state of an interval. Its formal semantics is defined by

   $$(\sigma,i,k,j)\models \mathit{empty} \text{ iff } k = j$$

With this semantics, *empty* can be expressed in terms of the *next* and *true*, as follows:

$$(Abb - e) \qquad empty \overset{\text{def}}{=} \neg \bigcirc true$$

Note that the consistency (this means all derived formulas are valid equivalences) between the semantics and syntax definitions can be proved. This claim is true for all abbreviations in this section.

2) *more*

Informally, *more*, read as more in the future, means that the current state is a non-final state of an interval. Its semantics is defined by

$$(\sigma, i, k, j) \models more \text{ iff } k < j$$

more can be expressed as the negation of *empty*:

$$(Abb - m) \qquad more \overset{\text{def}}{=} \neg empty \text{ (or } \bigcirc true)$$

3) *weak next*

If p is a formula, so is $\odot p$ ($\odot$ read as weak next). Its semantics is defined by

$$(\sigma, i, k, j) \models \odot p \text{ iff } (\sigma, i, k, j) \models \bigcirc p \text{ or } k = j.$$

It can be expressed in terms of the *next* and the *negation* (see the definition of empty) operators to satisfy the semantics:

$$(Abb - wnext) \qquad \odot p \overset{\text{def}}{=} empty \vee \bigcirc p$$

4) n^{th} *next* $(n \geq 0)$

If p is a formula, so is $\bigcirc^n p$ ($\bigcirc^n$ read as the n^{th} next). The operator $\bigcirc^n$ is an extension of the *next* operator. Its semantics is defined, as follows:

$$(\sigma, i, k, j) \models \bigcirc^n p \text{ iff } (k + n) \preceq j \text{ and } (\sigma, i, (k + n), j) \models p$$

The n^{th} *next* operator can be inductively defined in terms of the *next* operator.

$$(Abb - nnext) \qquad \begin{array}{l} \bigcirc^0 p \overset{\text{def}}{=} p \\ \bigcirc^n p \overset{\text{def}}{=} \bigcirc(\bigcirc^{n-1} p) \quad (n \in N) \end{array}$$

5) *length of an interval*

The future length of a finite interval (from the current point to the end) can be specified by the operator *len*. Its semantics is given by

$$(\sigma, i, k, j) \models len(n) \text{ iff } j \neq \omega \text{ and } (j - k) = n \ (n \geq 0).$$

The *len* construct can be defined in terms of the n^{th} *next* operator.

$$\begin{array}{ll} (Abb - len) & len(n) \overset{\text{def}}{=} \bigcirc^n empty \quad (n \in N_0) \\ (Abb - skip) & skip \overset{\text{def}}{=} len(1) \end{array}$$

skip specifies one unit of time over an interval.

6) *always in the future*

If p is a temporal formula, so is $\Box p$ (read always in the future). The semantics of the *always* operator can be defined, as follows:

$$(\sigma, i, k, j)\models\Box p \text{ iff for all } k\leq r\preceq j,\ (\sigma, i, r, j)\models p$$

The *always* operator can be defined as a dual of the *sometimes* operator.

$$(Abb - alw) \qquad \Box p \overset{\text{def}}{=} \neg\Diamond\neg p$$

7) *chop star*

If p is a temporal formula, so is p^* (read chop star). The semantics of the *chop-star* construct can be defined, as follows:

$$(\sigma, i, k, j)\models p^* \text{ iff } (\sigma, i, k, j)\models p^+ \text{ or } k = j$$

The *chop-star* operator can be defined by the *chop-plus* operator, as follows:

$$(Chop - star) \qquad p^* \overset{\text{def}}{=} p^+ \vee empty$$

2. Derived past operators

1) *first*

The $first$, read as the first state, means that the current state is the left end state of an interval. Its semantics is defined by

$$(\sigma, i, k, j)\models first \text{ iff } k = i$$

The $first$ construct can be defined in terms of the *previous* operator.

$$(Abb - f) \qquad first \overset{\text{def}}{=} \neg\ominus true$$

2) *elaps*

The *elaps*, read as elaps, means that the current state is a non-left end state of an interval. Its semantics is defined by

$$(\sigma, i, k, j)\models elaps \text{ iff } k > i$$

The *elaps* construct can be defined by the negation of the $first$ construct.

$$(Abb - el) \qquad elaps \overset{\text{def}}{=} \neg first \text{ (or } \ominus true)$$

3) *weak previous*

If p is a temporal formula, so is $\odot p$ (read weak previous). Its semantics is defined by

$$(\sigma, i, k, j)\models\odot p \text{ iff } (\sigma, i, k, j)\models\ominus p \text{ or } i = k$$

The weak previous operator can be defined using the previous operator.

$$(Abb - wprev) \qquad \odot p \overset{\text{def}}{=} first \vee \ominus p$$

4) n^{th} *previous*

If p is a temporal formula, so is $\ominus^n p$ (read n^{th} previous). The operator $\ominus^n$ is an extension of the previous operator. Its semantics is defined by

$$(\sigma, i, k, j) \models \ominus^n p \text{ iff } i \leq (k-n) \text{ and } (\sigma, i, (k-n), j) \models p.$$

The n^{th} previous operator can be inductively defined in terms of the *previous* operator.

$$(Abb - nprev) \qquad \ominus^0 p \overset{\text{def}}{=} p$$
$$\ominus^n p \overset{\text{def}}{=} \ominus(\ominus^{n-1} p) \qquad (n \in N)$$

5) *past length*

The length of an interval in the past (from the current state to the first state) can be specified by operator $\overline{len}$. Its semantics is defined, as follows:

$$(\sigma, i, k, j) \models \overline{len}(n) \text{ iff } k - i = n.$$

The construct $\overline{len}$ can be defined in terms of the n^{th} previous and the *first*.

$$(Abb - lenp) \qquad \overline{len}(n) \overset{\text{def}}{=} \ominus^n first \qquad (n \in N_0)$$
$$(Abb - skipp) \qquad \overline{skip} \overset{\text{def}}{=} \overline{len}(1)$$

$\overline{skip}$ specifies one unit of time in the past from the current state over an interval.

6) *always in the past*

If p is a temporal formula, so is $\boxminus p$ ($\boxminus$ read as always in the past). Its semantics is defined by

$$(\sigma, i, k, j) \models \boxminus p \text{ iff for all } l,\ i \leq l \leq k,\ (\sigma, i, l, j) \models p$$

The *always* in the past operator can be defined as a dual of the *sometimes* in the past.

$$(Abb - alwp) \qquad \boxminus p \overset{\text{def}}{=} \neg \overline{\Diamond} \neg p$$

2.5 Precedence Rules

In order to avoid an excessive number of parentheses, the following precedence rules are used:

1 $\neg$
2 $\bigcirc, \odot, \Diamond, \Box, \ominus, \underline{\odot}, \overline{\Diamond}, \boxminus, +, *$
3 $\wedge, \vee$
4 $\rightarrow, \leftrightarrow$
5 $;, \overline{;}$

where 1=highest and 5=lowest.

2.6 Equivalence Relations

A formula is called a state formula if it does not contain any temporal operator; otherwise it is called a temporal formula. A formula is called a non-past (non-future) formula if it does not contain any past (future) temporal operators.

Sometimes, we denote $\models p \leftrightarrow q$ by $p \approx q$ and $\models \Box(p \leftrightarrow q)$ by $p \equiv q$. The former is called 'weak equivalence' and the latter is called 'strong equivalence'. Similarly, we denote $\models p \rightarrow q$ by $p \hookrightarrow q$, calling it 'weak implication', and $\models \Box(p \rightarrow q)$ by $p \supset q$, calling it 'strong implication'. In fact, $p \approx q$ means that p and q have the same truth value in the first state of every model while $p \equiv q$ means that p and q have the same truth value in all states of every model. Similar explanations can be given for the weak and the strong implication relations. It is obvious that the strong equivalence (implication) implies the weak equivalence (implication) but the inverse does not hold. For instance, $first \approx true$ but $first \not\equiv true$. However, within the future formulas, $p \approx q$ if and only if $p \equiv q$ (it follows from Definition 2.2 (page 22) and Theorem A.1 proved in the Appendix).

Note that the relations, $\approx$ and $\equiv$, are reflexive, symmetric, and transitive. They are therefore equivalence relations over L_{epitl} (see below).

In practice, we frequently use the strong equivalence relation for reduction of programs. This will be discussed in Chapter 4.

2.7 Logic Laws

In this section, we present a collection of logic laws, i.e. some valid formulas in the logic system. First, we discuss tautological validity.

Let L_p denote the language of classical propositional logic, i.e. the set of all well-formed formulas, and L_{epitl} denote the language of the extended propositional ITL.

Definition 2.1 A formula of L_{epitl} is called tautologically valid if it results from a tautology P of L_p by consistently replacing the atomic formulas of P by formulas of L_{epitl}. $\Box$

For instance, $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$ is a tautology in L_p. By taking P to be $\bigcirc A$ and Q to be $\Box B$, we have the tautologically valid formula: $\neg(\bigcirc A \vee \Box B) \leftrightarrow \neg \bigcirc A \wedge \neg \Box B$ in L_{epitl}.

Theorem 2.1 If p is a tautologically valid formula then $\models \Box p$.

Proof

The proof is similar to the one presented in [51]. Let A^* result from a formula A of L_p by replacing the atomic formulas $p_1, ..., p_n$ of A respectively by formulas $q_1, ..., q_n$ of L_{epitl} and let σ be an interval and k $(0 \leq k \preceq |\sigma|)$ an integer. Define a classical valuation E of atomic formulas of L_p by $E(p_j)$ = true iff $(\sigma, 0, k, |\sigma|) \models q_j$ for $j = 1, ..., n$ (and with arbitrary values for other atomic propositions). We claim that:

$$E(A) = \text{true iff } (\sigma, 0, k, |\sigma|) \models A^*$$

which proves the theorem, since $E(A)$ =true if A is a tautology.

The proof of the claim runs by induction on (the syntax of) A:

(1) A is p_j: then $A^\star$ is q_j, hence $E(A) = E(p_j)$ =true iff $(\sigma, 0, k, |\sigma|) \models q_j$ iff $(\sigma, 0, k, |\sigma|) \models A^\star$.

(2) A is $\neg B$: then $A^\star$ is $\neg B^\star$ in correspondence with the meaning of the 'operator' $\star$ on B. Hence, using the hypothesis

$$\begin{array}{rcl} E(A) = \text{ true} & \Longleftrightarrow & E(B) = \text{ false} \\ & \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \not\models B^\star \\ & \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \neg B^\star \\ & \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models A^\star \end{array}$$

(3) A is $B_1 \wedge B_2$: we have again $A^\star$ being $B_1^\star \wedge B_2^\star$ and with the induction hypothesis, we get:

$$\begin{array}{rcl} E(A) = \text{ true} & \Longleftrightarrow & E(B_1) = \text{ true and } E(B_2) = \text{ true} \\ & \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models B_1^\star \text{ and } (\sigma, 0, k, |\sigma|) \models B_2^\star \\ & \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models B_1^\star \wedge B_2^\star \\ & \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models A^\star \end{array}$$

□

Theorem 2.1 allows us to use a class of valid formulas without proving them as long as they are tautologically valid. The validity and satisfaction are 'dual' notions in the following sense:

Theorem 2.2 p is valid iff $\neg p$ is not satisfiable.

Proof

$$\begin{array}{rcll} \models p & \Longleftrightarrow & \sigma \models p & \text{for all } \sigma \\ & \Longleftrightarrow & \neg\neg(\sigma \models p) & \text{for all } \sigma \\ & \Longleftrightarrow & \neg(\sigma \not\models p) & \text{for all } \sigma \\ & \Longleftrightarrow & \neg(\sigma \models \neg p) & \text{for all } \sigma \\ & \Longleftrightarrow & \neg(\sigma \models \neg p & \text{for some } \sigma) \end{array}$$

□

Theorem 2.2 provides us with a means to prove the validity of a formula. When direct proof of the validity of a formula is difficult, we frequently prove the fact that the negation of the formula is not satisfiable.

Theorem 2.3 Let $\mathcal{I}$ be an interpretation. If $\mathcal{I} \models p$ and $\mathcal{I} \models p \rightarrow q$ then $\mathcal{I} \models q$.

Proof

Suppose $\mathcal{I} \models p$ and $\mathcal{I} \models p \rightarrow q$. The latter amounts to $\mathcal{I} \models \neg p$ or $\mathcal{I} \models q$. Hence $\mathcal{I} \models p \wedge q$ leading to $\mathcal{I} \models q$.

□

Theorem 2.3 states that *modus ponens* is a 'valid rule of inference'.

Theorem 2.4 Let $\mathcal{I}$ be an interpretation. If $\mathcal{I} \models q$ implies $\mathcal{I} \models p$, then $\mathcal{I} \models q \rightarrow p$

Proof

If $\mathcal{I} \not\models q \rightarrow p$, then $\mathcal{I} \models \neg(q \rightarrow p)$ leading to $\mathcal{I} \models q \wedge \neg p$. It thus follows that $\mathcal{I} \models \neg p$ and $\mathcal{I} \models q$, a contradiction.

□

So far we have looked at the logic laws in EPITL coming from the 'classical logic'. Let us now turn our attention to proper temporal logic laws concerning the temporal operators. In what follows, we prove, first of all, the monotonicity laws stated in Theorem 2.5 which are a basis for proving the substitution law given in Theorem 2.7. Subsequently, we give an extensive list of formulas all of which we claim to be valid without proving these facts except for a few examples.

Theorem 2.5 If p_1, p_2, q_1, q_2 are formulas, then

$FM0$	If $(p_1 \supset p_2)$ and $(q_1 \hookrightarrow q_2)$ then $((p_1; q_1) \supset (p_2; q_2))$
$PM0$	If $(p_1 \supset p_2)$ and $(q_1 \supset q_2)$ then $((p_1 \,\bar{;}\, q_1) \supset (p_2 \,\bar{;}\, q_2))$

Proof

The proof of FM0

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$. Suppose $(\sigma, 0, k, |\sigma|) \models p_1; q_1$. Thus, $(\sigma, 0, k, r) \models p_1$ and $(\sigma, r, r, |\sigma|) \models q_1$ for some $r, k \leq r \preceq |\sigma|$.

Since $p_1 \supset p_2$, we have $(\sigma_{(0..r)}, 0, k, r) \models p_1 \rightarrow p_2$. By Theorem A.1 (Appendix) $(\sigma, 0, k, r) \models p_1 \rightarrow p_2$. By Theorem 2.3, $(\sigma, 0, k, r) \models p_2$.

Moreover, by $q_1 \hookrightarrow q_2$, we obtain $\sigma_{(r..|\sigma|)} \models q_1 \rightarrow q_2$. By Theorem A.1, $(\sigma, r, r, |\sigma|) \models q_1 \rightarrow q_2$. Hence $(\sigma, r, r, |\sigma|) \models q_2$. It then follows that $(\sigma, 0, k, |\sigma|) \models p_2; q_2$.

The proof of PM0

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$. Suppose $(\sigma, 0, k, |\sigma|) \models p_1 \,\bar{;}\, q_1$. Thus, $(\sigma, 0, l, l) \models p_1$ and $(\sigma, l, k, |\sigma|) \models q_1$ for some $l, 0 \leq l \leq k$.

Since $p_1 \supset p_2$, we have $(\sigma_{(0..l)}, 0, l, l) \models p_1 \rightarrow p_2$. By Theorem A.1, $(\sigma, 0, l, l) \models p_1 \rightarrow p_2$. Hence $(\sigma, 0, l, l) \models p_2$.

Similarly, since $q_1 \supset q_2$, we have $(\sigma_{(l..|\sigma|)}, 0, k - l, |\sigma| - l) \models q_1 \rightarrow q_2$. By Theorem A.1, $(\sigma, l, k, |\sigma|) \models q_1 \rightarrow q_2$. By Theorem 2.3, it follows that $(\sigma, l, k, |\sigma|) \models q_2$.

Hence, $(\sigma, 0, k, |\sigma|) \models p_2 \,\bar{;}\, q_2$. We obtain $(p_1 \,\bar{;}\, q_1) \supset (p_2 \,\bar{;}\, q_2)$.

□

As an immediate consequence of Theorem 2.5, we obtain the following

Corollary 2.6

$FM1$	If $p_1 \supset p_2$ then $(p_1; q) \supset (p_2; q)$	$PM1$	If $p_1 \supset p_2$ then $(p_1 \,\bar{;}\, q) \supset (p_2 \,\bar{;}\, q)$
$FM2$	If $p_1 \hookrightarrow p_2$ then $(q; p_1) \supset (q; p_2)$	$PM2$	If $p_1 \supset p_2$ then $(q \,\bar{;}\, p_1) \supset (q \,\bar{;}\, p_2)$
$FM3$	If $p_1 \equiv p_2$ and $q_1 \approx q_2$ then $(p_1; q_1) \equiv (p_2; q_2)$	$PM3$	If $p_1 \equiv p_2$ and $q_1 \equiv q_2$ then $(p_1 \,\bar{;}\, q_1) \equiv (p_2 \,\bar{;}\, q_2)$

□

Theorem 2.7 If F is a formula in EPITL involving a subformula g, and f is a formula such that $f \equiv g$, then

$$F \equiv F[f/g]$$

where $F[f/g]$ denotes the formula given by replacing some occurrences of g in F by f.

Proof

We need to show that, for all f, g, h in EPITL, if $f \equiv g$ then

$$\begin{array}{llcllllcl}
1 & f \wedge h & \equiv & g \wedge h & \quad & 2 & h \wedge f & \equiv & h \wedge g \\
3 & \neg f & \equiv & \neg g & & 4 & f^{+} & \equiv & g^{+} \\
5 & \bigcirc f & \equiv & \bigcirc g & & 6 & \ominus f & \equiv & \ominus g \\
7 & \Diamond f & \equiv & \Diamond g & & 8 & \diamondminus f & \equiv & \diamondminus g \\
9 & h; f & \equiv & h; g & & 10 & h \bar{;} f & \equiv & h \bar{;} g \\
11 & f; h & \equiv & g; h & & 12 & f \bar{;} h & \equiv & g \bar{;} h
\end{array}$$

Each of 1,2,3,4,5,6,7 and 8 is an immediate consequence of the definitions; and 9, 10, 11 and 12 follow from Corollary2.6 (FM3), (PM3).

□

A simple check can be made to see that Theorem 2.5 and Corollary 2.6 still hold if we use the weak equivalence and implication to replace the strong equivalence and implication.

Theorem 2.8 If p_1, p_2, q_1, q_2 are formulas, then

1. $FM0'$ If $((p_1 \hookrightarrow p_2)$ and $(q_1 \hookrightarrow q_2))$ *then* $((p_1; q_1) \hookrightarrow (p_2; q_2))$
2. $PM0'$ If $((p_1 \hookrightarrow p_2)$ and $(q_1 \hookrightarrow q_2))$ *then* $((p_1 \bar{;} q_1) \hookrightarrow (p_2 \bar{;} q_2))$

□

Corollary 2.9

$FM1'$	If $p_1 \hookrightarrow p_2$ then $(p_1; q) \hookrightarrow (p_2; q)$	$PM1'$	If $p_1 \hookrightarrow p_2$ then $(p_1 \bar{;} q) \hookrightarrow (p_2 \bar{;} q)$
$FM2'$	If $p_1 \hookrightarrow p_2$ then $(q; p_1) \hookrightarrow (q; p_2)$	$PM2'$	If $p_1 \hookrightarrow p_2$ then $(q \bar{;} p_1) \hookrightarrow (q \bar{;} p_2)$
$FM3'$	If $p_1 \approx p_2$ and $q_1 \approx q_2$ then $(p_1; q_1) \approx (p_2; q_2)$	$PM3'$	If $p_1 \approx p_2$ and $q_1 \approx q_2$ then $(p_1 \bar{;} q_1) \approx (p_2 \bar{;} q_2)$

□

In general, Theorem 2.7 is not valid under the weak equivalence. A counterexample is given below:

$$\mathit{first} \approx \mathit{true} \text{ but } \bigcirc\, \mathit{first} \not\approx \bigcirc \mathit{true}.$$

In the following, we give some logic laws concerning the temporal operators. First, we prove an auxiliary lemma.

Lemma 2.10

1. $\mathit{more} \supset (\neg \bigcirc p \leftrightarrow \bigcirc \neg p)$
2. $\mathit{more} \wedge \neg \bigcirc p \equiv \mathit{more} \wedge \bigcirc \neg p$

Proof

(1) Let σ be an interval, and k an integer, $0 \leq k \preceq |\sigma|$. If $k = |\sigma|$, then $(\sigma, 0, k, |\sigma|) \models (\mathit{more} \rightarrow (\neg \bigcirc p \leftrightarrow \bigcirc \neg p))$ is trivially true. If $k < |\sigma|$, we have $(\sigma, 0, k, |\sigma|) \models \mathit{more}$ and

$$\begin{aligned}
(\sigma, 0, k, |\sigma|) \models \neg \bigcirc p &\iff (\sigma, 0, k, |\sigma|) \not\models \bigcirc p \\
&\iff (\sigma, 0, k+1, |\sigma|) \not\models p \\
&\iff (\sigma, 0, k+1, |\sigma|) \models \neg p \\
&\iff (\sigma, 0, k, |\sigma|) \models \bigcirc \neg p
\end{aligned}$$

Therefore, $\mathit{more} \supset (\neg \bigcirc p \leftrightarrow \bigcirc \neg p)$.

(2) A proof can be given in a similar way to the proof of 1. However, 2 can be derived from 1. Since $(A \rightarrow (B \leftrightarrow C)) \leftrightarrow (A \wedge B \leftrightarrow A \wedge C)$ is a tautology in first order logic, $(\mathit{more} \rightarrow (\neg \bigcirc p \leftrightarrow \bigcirc \neg p)) \leftrightarrow (\mathit{more} \wedge \neg \bigcirc p \leftrightarrow \mathit{more} \wedge \bigcirc \neg p)$ is tautologically valid. Hence, by Theorem 2.1, $(\mathit{more} \rightarrow (\neg \bigcirc p \leftrightarrow \bigcirc \neg p) \equiv (\mathit{more} \wedge \neg \bigcirc p \leftrightarrow \mathit{more} \bigcirc \neg p)$. Moreover, from (1), $\models \Box(\mathit{more} \rightarrow (\neg \bigcirc p \leftrightarrow \bigcirc \neg p)$, so by Theorem 2.7, we have $\models \Box(\mathit{more} \wedge \neg \bigcirc p \leftrightarrow \mathit{more} \wedge \bigcirc \neg p)$. That is, $\mathit{more} \wedge \neg \bigcirc p \equiv \mathit{more} \wedge \bigcirc \neg p$.

□

Theorem 2.11 Duality Laws

$FDU1$	$\neg \odot p \equiv \bigcirc \neg p$	$PDU1$	$\neg \ominus p \equiv \underline{\odot} \neg p$
$FDU2$	$\neg \bigcirc p \equiv \odot \neg p$	$PDU2$	$\neg \underline{\odot} p \equiv \ominus \neg p$
$FDU3$	$\neg \Diamond p \equiv \Box \neg p$	$PDU3$	$\neg \boxminus p \equiv \underline{\Diamond} \neg p$
$FDU4$	$\neg \Box p \equiv \Diamond \neg p$	$PDU4$	$\neg \underline{\Diamond} p \equiv \boxminus \neg p$
$FDU5$	$\neg \mathit{more} \equiv \mathit{empty}$	$PDU5$	$\neg \mathit{elaps} \equiv \mathit{first}$

Proof

The proof of FDU5 is straightforward; the proofs of FDU2 and FDU4 are similar to the proofs of FDU1 and FDU3. The proofs of PDU1 - PDU5 are analogous to FDU1 - FDU5. We prove only FDU1 and FDU3.

The proof of FDU1

$\odot p$	$\equiv$	$\mathit{empty} \vee \bigcirc p$	Abb-wnext
$\neg \odot p$	$\equiv$	$\neg(\mathit{empty} \vee \bigcirc p)$	theorem 2.7
	$\equiv$	$\neg \mathit{empty} \wedge \neg \bigcirc p$	theorem 2.1
	$\equiv$	$\mathit{more} \wedge \neg \bigcirc p$	Abb-m
	$\equiv$	$\mathit{more} \wedge \bigcirc \neg p$	lemma 2.10 (2)
	$\equiv$	$\bigcirc \mathit{true} \wedge \bigcirc \neg p$	Abb-e, Abb-m
	$\equiv$	$\bigcirc(\mathit{true} \wedge \neg p)$	FD3 (theorem 2.17)
	$\equiv$	$\bigcirc \neg p$	theorem 2.1

Note that FD3 can be used in the above proof before its proof in Theorem 2.17 since the proof of FD3 does not depend on FDU1.

The proof of FDU3

$\neg \Diamond \neg q \equiv \Box q$	Abb-alw
$\neg \Diamond p \equiv \Box \neg p$	let p be $\neg q$, theorems 2.1, 2.7

□

Theorem 2.12 Reflexivity Laws

$FR1$	$\Box p \supset p$	$PR1$	$\boxminus p \supset p$
$FR2$	$p \supset \Diamond p$	$PR2$	$p \supset \Diamondminus p$

Proof Straightforward. □

Theorem 2.13 Laws about the 'strength' of the operators

$FS1$	$\Box p \supset \odot p$	$PS1$	$\boxminus p \supset \underset{\cdot}{\ominus} p$
$FS2$	$\Box p \supset \Diamond p$	$PS2$	$\boxminus p \supset \Diamondminus p$
$FS3$	$\bigcirc p \supset \Diamond p$	$PS3$	$\ominus p \supset \Diamondminus p$
$FS4$	$\bigcirc p \supset \mathit{more}$	$PS4$	$\ominus p \supset \mathit{elaps}$
$FS5$	$\Diamond\Box p \supset \Box\Diamond p$	$PS5$	$\Diamondminus\boxminus p \supset \boxminus\Diamondminus p$
$FS6$	$(p;q) \supset \Diamond q$	$PS6$	$(p\mathbin{\bar{;}}q) \supset \Diamondminus p$
$FS7$	$(p;q_1\wedge q_2) \supset p;q_1$	$PS7$	$(p_1\wedge p_2\mathbin{\bar{;}}q) \supset p_1\mathbin{\bar{;}}q$
$FS8$	$(p;q_1\wedge q_2) \supset p;q_2$	$PS8$	$(p_1\wedge p_2\mathbin{\bar{;}}q) \supset p_2\mathbin{\bar{;}}q$
$FS9$	$\Box(p\to q)\wedge(w;p) \supset (w;q)$	$PS9$	$\boxminus(p\to q)\wedge(p\mathbin{\bar{;}}w) \supset (q\mathbin{\bar{;}}w)$
$FS10$	$\Box p\wedge(w;q) \supset (w;p\wedge q)$	$PS10$	$\boxminus p\wedge(q\mathbin{\bar{;}}w) \supset (p\wedge q\mathbin{\bar{;}}w)$

Proof

We prove only $FS5$ and $PS8$. Let σ be an interval and k an integer, $0 \le k \preceq |\sigma|$.

The proof of FS5

$$\begin{array}{ll}
 & (\sigma,0,k,|\sigma|) \models \Diamond\Box p \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \Box p \text{ for some } r, k \le r \preceq |\sigma| \\
\Longleftrightarrow & (\sigma,0,r_1,|\sigma|) \models p \text{ for some } r \text{ and for all } r_1, k \le r \le r_1 \preceq |\sigma| \\
\Longrightarrow & (\sigma,0,h_1,|\sigma|) \models p \text{ for all } h \text{ and for some } h_1, k \le h \le h_1 \preceq |\sigma| \\
\Longleftrightarrow & (\sigma,0,h,|\sigma|) \models \Diamond p \text{ for all } h,\ k \le h \preceq |\sigma| \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|) \models \Box\Diamond p
\end{array}$$

The proof of PS8

$$\begin{array}{ll}
 & (\sigma,0,k,|\sigma|) \models p_1 \wedge p_2\mathbin{\bar{;}}q \\
\Longleftrightarrow & (\sigma,0,l,l) \models p_1 \wedge p_2 \text{ and } (\sigma,l,k,|\sigma|) \models q \text{ for some } l, 0 \le l \le k \\
\Longrightarrow & (\sigma,0,l,l) \models p_2 \text{ and } (\sigma,l,k,|\sigma|) \models q \text{ for some } l, 0 \le l \le k \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p_2\mathbin{\bar{;}}q
\end{array}$$

□

Theorem 2.14 Laws about weak and strong operators

$FW1$	$\bigcirc p \equiv \odot p \wedge \mathit{more}$	$PW1$	$\ominus p \equiv \underset{\cdot}{\ominus} p \wedge \mathit{elaps}$
$FW2$	$\odot p \equiv \bigcirc p \vee \mathit{empty}$	$PW2$	$\underset{\cdot}{\ominus} p \equiv \ominus p \vee \mathit{first}$
$FW3$	$\bigcirc p \supset \odot p$	$PW3$	$\ominus p \supset \underset{\cdot}{\ominus} p$

Proof

FW2 and PW2 follow directly from definitions; FW3 and PW3 follow from FW1 and PW1; the proof of PW1 is similar to the proof of FW1. So we prove only FW1.

$$\begin{array}{rcll}
\odot p \wedge more & \equiv & (\bigcirc p \vee empty) \wedge more & \text{theorem 2.7, FW2} \\
 & \equiv & \bigcirc p \wedge more & \text{theorem 2.1, } more \wedge empty \equiv false \\
 & \equiv & \bigcirc p \wedge \bigcirc true & \text{Abb-e, theorem 2.7} \\
 & \equiv & \bigcirc (p \wedge true) & \text{FD3} \\
 & \equiv & \bigcirc p & \text{theorem 2.7}
\end{array}$$

□

Theorem 2.15 Idempotency Laws

$FI1$	$\Box\Box p \equiv \Box p$	$PI1$	$\underline{\Box}\underline{\Box} p \equiv \underline{\Box} p$
$FI2$	$\Diamond\Diamond p \equiv \Diamond p$	$PI2$	$\underline{\Diamond}\underline{\Diamond} p \equiv \underline{\Diamond} p$
$FI3$	$empty \wedge empty \equiv empty$	$PI3$	$first \wedge first \equiv first$
$FI4$	$more \wedge more \equiv more$	$PI4$	$elaps \wedge elaps \equiv elaps$

Proof Straightforward.

□

Theorem 2.16 Commutativity Laws

$FC1$	$\models_{Inf} \Box(\Box\bigcirc p \leftrightarrow \bigcirc\Box p)$		
$FC2$	$\models_{Inf} \Box(\Diamond\bigcirc p \leftrightarrow \bigcirc\Diamond p)$		
$FC3$	$\Box\odot p \equiv \odot\Box p$	$PC3$	$\underline{\Box}\underline{\odot} p \equiv \underline{\odot}\underline{\Box} p$
$FC4$	$\Diamond\odot p \rightarrow \odot\Diamond p$	$PC4$	$\underline{\Diamond}\underline{\odot} p \rightarrow \underline{\odot}\underline{\Diamond} p$

where $Inf = \{\sigma \mid |\sigma| = \omega\}$.

Proof

We prove only FC1 and FC3. Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of FC1:

Since $|\sigma| = \omega$, we have

$$\begin{array}{rl}
 & (\sigma, 0, k, \omega) \models \Box \bigcirc p \\
\Longleftrightarrow & (\sigma, 0, k + r, \omega) \models \bigcirc p \text{ for all } r, 0 \leq r < \omega \\
\Longleftrightarrow & (\sigma, 0, k + r + 1, \omega) \models p \text{ for all } r, 0 \leq r < \omega \\
\Longleftrightarrow & (\sigma, 0, k + 1, \omega) \models \Box p \\
\Longleftrightarrow & (\sigma, 0, k, \omega) \models \bigcirc \Box p
\end{array}$$

The proof of FC3:

If $k = |\sigma|$, $(\sigma, 0, k, |\sigma|) \models \Box \odot p \leftrightarrow \odot \Box p$ is trivially true; in the case of $k < |\sigma|$, we have

$$\begin{array}{rl}
 & (\sigma, 0, k, |\sigma|) \models \odot \Box p \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \bigcirc \Box p \\
\Longleftrightarrow & (\sigma, 0, k + 1, |\sigma|) \models \Box p \\
\Longleftrightarrow & (\sigma, 0, k + r + 1, |\sigma|) \models p \text{ for all } r, 0 \leq r \text{ and } r + k + 1 \preceq |\sigma|
\end{array}$$

$$
\begin{aligned}
&\Longleftrightarrow && (\sigma,0,k+r,|\sigma|)\models \bigcirc p \text{ for that } r\\
&\Longleftrightarrow && (\sigma,0,k+r,|\sigma|)\models \bigcirc p \text{ for that } r\\
& && \text{and } (\sigma,0,k+r_1,|\sigma|)\models \mathit{empty} \text{ for } r_1=|\sigma|-k \text{ (if } |\sigma|\neq\omega)\\
&\Longleftrightarrow && (\sigma,0,k+r,|\sigma|)\models \odot p \text{ for all } r, 0\le r \text{ and } k+r\preceq|\sigma|\\
&\Longleftrightarrow && (\sigma,0,k,|\sigma|)\models \Box\odot p
\end{aligned}
$$

□

Theorem 2.17 Distributivity Laws

$FD1$	$\Box(p\wedge q)$	$\equiv$	$\Box p\wedge\Box q$	$PD1$	$\boxminus(p\wedge q)$	$\equiv$	$\boxminus p\wedge\boxminus q$
$FD2$	$\Diamond(p\vee q)$	$\equiv$	$\Diamond p\vee\Diamond q$	$PD2$	$\diamondminus(p\vee q)$	$\equiv$	$\diamondminus p\vee\diamondminus q$
$FD3$	$\bigcirc(p\wedge q)$	$\equiv$	$\bigcirc p\wedge\bigcirc q$	$PD3$	$\ominus(p\wedge q)$	$\equiv$	$\ominus p\wedge\ominus q$
$FD4$	$\bigcirc(p\vee q)$	$\equiv$	$\bigcirc p\vee\bigcirc q$	$PD4$	$\ominus(p\vee q)$	$\equiv$	$\ominus p\vee\ominus q$
$FD5$	$\bigcirc(p\rightarrow q)$	$\equiv$	$\odot p\rightarrow\bigcirc q$	$PD5$	$\ominus(p\rightarrow q)$	$\equiv$	$\underset{\cdot}{\ominus} p\rightarrow\ominus q$
$FD6$	$\odot(p\wedge q)$	$\equiv$	$\odot p\wedge\odot q$	$PD6$	$\underset{\cdot}{\ominus}(p\wedge q)$	$\equiv$	$\underset{\cdot}{\ominus} p\wedge\underset{\cdot}{\ominus} q$
$FD7$	$\odot(p\vee q)$	$\equiv$	$\odot p\vee\odot q$	$PD7$	$\underset{\cdot}{\ominus}(p\vee q)$	$\equiv$	$\underset{\cdot}{\ominus} p\vee\underset{\cdot}{\ominus} q$
$FD8$	$\odot(p\rightarrow q)$	$\equiv$	$\bigcirc p\rightarrow\odot q$	$PD8$	$\underset{\cdot}{\ominus}(p\rightarrow q)$	$\equiv$	$\ominus p\rightarrow\underset{\cdot}{\ominus} q$
$FD9$	$(w;p\vee q)$	$\equiv$	$(w;p)\vee(w;q)$	$PD9$	$(w\,\bar{;}\,p\vee q)$	$\equiv$	$(w\,\bar{;}\,p)\vee(w\,\bar{;}\,q)$
$FD10$	$(p\vee q;w)$	$\equiv$	$(p;w)\vee(q;w)$	$PD10$	$(p\vee q\,\bar{;}\,w)$	$\equiv$	$(p\,\bar{;}\,w)\vee(q\,\bar{;}\,w)$

Proof

We prove FD2 - FD6, FD9, and PD10. In the proofs, we use some interpretation rules, abbreviations, and proved logic laws without declaration. Let σ be an interval, and k an integer, $0\le k\preceq|\sigma|$.

The proof of FD2

$$
\begin{aligned}
& && (\sigma,0,k,|\sigma|)\models\Diamond p\vee\Diamond q\\
&\Longleftrightarrow && (\sigma,0,j,|\sigma|)\models p \text{ for some } j, k\le j\preceq|\sigma| \text{ or}\\
& && (\sigma,0,i,|\sigma|)\models q \text{ for some } i, k\le i\preceq|\sigma|\\
&\Longleftrightarrow && (\sigma,0,r,|\sigma|)\models p\vee q \text{ for some } r, k\le r\preceq|\sigma|\\
&\Longleftrightarrow && (\sigma,0,k,|\sigma|)\models\Diamond(p\vee q)
\end{aligned}
$$

The proof of FD3

$$
\begin{aligned}
& && (\sigma,0,k,|\sigma|)\models\bigcirc p\wedge\bigcirc q\\
&\Longleftrightarrow && (\sigma,0,k+1,|\sigma|)\models p \text{ and } (\sigma,0,k+1,|\sigma|)\models q \text{ and } k<|\sigma|\\
&\Longleftrightarrow && (\sigma,0,k+1,|\sigma|)\models p\wedge q\\
&\Longleftrightarrow && (\sigma,0,k,|\sigma|)\models\bigcirc(p\wedge q)
\end{aligned}
$$

The proof of FD4

$$
\begin{aligned}
& && (\sigma,0,k,|\sigma|)\models\bigcirc p\vee\bigcirc q\\
&\Longleftrightarrow && ((\sigma,0,k+1,|\sigma|)\models p \text{ or } (\sigma,0,k+1,|\sigma|)\models q) \text{ and } k<|\sigma|\\
&\Longleftrightarrow && (\sigma,0,k+1,|\sigma|)\models p\vee q\\
&\Longleftrightarrow && (\sigma,0,k,|\sigma|)\models\bigcirc(p\vee q)
\end{aligned}
$$

The proof of FD5

$$
\begin{aligned}
& && \bigcirc(p\rightarrow q) && \\
&\equiv && \bigcirc(\neg p\vee q) && \text{theorem 2.1, theorem 2.7}\\
&\equiv && \bigcirc\neg p\vee\bigcirc q && \text{FD4}\\
&\equiv && \neg\odot p\vee\bigcirc q && \text{FDU2}\\
&\equiv && \odot p\rightarrow\bigcirc q && \text{theorem 2.1}
\end{aligned}
$$

The proof of FD6

$$\begin{array}{cll}
 & \odot p\wedge\odot q & \\
\equiv & (empty\vee\bigcirc p)\wedge(empty\vee\bigcirc q) & \text{Abb-wnext} \\
\equiv & (empty\wedge empty)\vee(empty\wedge\bigcirc p)\vee(empty\wedge\bigcirc q)\vee(\bigcirc p\wedge\bigcirc q) & \text{theorem 2.1} \\
\equiv & empty\vee\bigcirc p\wedge\bigcirc q & \text{FI3, } empty\wedge\bigcirc p\equiv false \\
\equiv & empty\vee\bigcirc(p\wedge q) & \text{FD3} \\
\equiv & \odot(p\wedge q) & \text{Abb-wnext}
\end{array}$$

The proof of FD9

$$\begin{array}{cl}
 & (\sigma,0,k,|\sigma|)\models w;p\vee q \\
\Longleftrightarrow & (\sigma,0,k,r)\models w \text{ and } ((\sigma,r,r,|\sigma|)\models p \text{ or } (\sigma,r,r,|\sigma|)\models q) \text{ for some } r,\ k\leq r\preceq|\sigma|. \\
\Longleftrightarrow & (\sigma,0,k,r)\models w \text{ and } (\sigma,r,r,|\sigma|)\models p \text{ for some } r,\ k\leq r\preceq|\sigma|. \\
 & \text{or} \\
 & (\sigma,0,k,r)\models w \text{ and } (\sigma,r,r,|\sigma|)\models q \text{ for some } r,\ k\leq r\preceq|\sigma|. \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models(w;p)\vee(w;q)
\end{array}$$

The proof of PD10

$$\begin{array}{cl}
 & (\sigma,0,k,|\sigma|)\models p\vee q\bar{;}w \\
\Longleftrightarrow & ((\sigma,0,l,l)\models p \text{ or } (\sigma,0,l,l)\models q) \text{ and } (\sigma,l,k,|\sigma|)\models w \text{ for some } l,0\leq l\leq k. \\
\Longleftrightarrow & (\sigma,0,l,l)\models p \text{ and } (\sigma,l,k,|\sigma|)\models p \text{ for some } l,\ 0\leq l\leq k. \\
 & \text{or} \\
 & (\sigma,0,l,l)\models q \text{ and } (\sigma,l,k,|\sigma|)\models w \text{ for some } l,\ 0\leq l\leq k. \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models(p;w)\vee(q;w)
\end{array}$$

□

Theorem 2.18 Weak Distributivity Laws

$FWD1$	$\Box(p\rightarrow q)\supset(\Box p\rightarrow\Box q)$	$PWD1$	$\boxminus(p\rightarrow q)\supset(\boxminus p\rightarrow\boxminus q)$
$FWD2$	$\Box p\vee\Box q\supset\Box(p\vee q)$	$PWD2$	$\boxminus p\vee\boxminus q\supset\boxminus(p\vee q)$
$FWD3$	$(\Diamond p\rightarrow\Diamond q)\supset\Diamond(p\rightarrow q)$	$PWD3$	$(\underset{-}{\Diamond}p\rightarrow\underset{-}{\Diamond}q)\supset\underset{-}{\Diamond}(p\rightarrow q)$
$FWD4$	$\Diamond(p\wedge q)\supset\Diamond p\wedge\Diamond q$	$PWD4$	$\underset{-}{\Diamond}(p\wedge q)\supset\underset{-}{\Diamond}p\wedge\underset{-}{\Diamond}q$

Proof

We prove only FWD4. Let σ be an interval, and k an integer, $0\leq k\preceq|\sigma|$.

$$\begin{array}{cl}
 & (\sigma,0,k,|\sigma|)\models\Diamond(p\wedge q) \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|)\models p\wedge q \text{ for some } r,k\leq r\preceq|\sigma| \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|)\models p \text{ and } (\sigma,0,r,|\sigma|)\models q \text{ for that } r \\
\Longrightarrow & (\sigma,0,k,|\sigma|)\models\Diamond p \text{ and } (\sigma,0,k,|\sigma|)\models\Diamond q \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models\Diamond p\wedge\Diamond q
\end{array}$$

□

Theorem 2.19 Expansion Laws

$FE1$	$\Diamond p\equiv p\vee\bigcirc\Diamond p$	$PE1$	$\underset{-}{\Diamond}p\equiv p\vee\ominus\underset{-}{\Diamond}p$
$FE2$	$\Box p\equiv p\wedge\odot\Box p$	$PE2$	$\boxminus p\equiv p\wedge\underset{-}{\odot}\boxminus p$

These laws are very useful for both theorem proving and the reduction of programs.

Proof

FE1 and PE1 follow directly from the definitions. The proof of PE2 is similar to the proof of FE2. We prove only FE2.

$$\begin{array}{rcll} \Box p & \equiv & \neg\Diamond\neg p & \text{Abb-alw} \\ & \equiv & \neg(\neg p \vee \bigcirc\Diamond\neg p) & \text{FE1, theorem 2.7} \\ & \equiv & \neg\neg p \wedge \odot\neg\Diamond\neg p & \text{FDU2, theorems 2.1, 2.7} \\ & \equiv & p \wedge \odot\Box p & \text{Abb-alw, theorems 2.1, 2.7} \end{array}$$

□

To formulate further chop laws and end point laws, some definitions such as *left end closed* and *right end closed* formulas are required.

Definition 2.2 A formula p is called *left end closed* (*lec*-formula) if $(\sigma, k, k, j) \models p \iff (\sigma, i, k, j) \models p$ for any interpretation (σ, i, k, j).

A formula p is called *right end closed* (*rec*-formula) if $(\sigma, i, k, k) \models p \iff (\sigma, i, k, j) \models p$ for any interpretation (σ, i, k, j). □

Intuitively, a formula p being left end closed means that if p holds over a subinterval $\sigma_{(k..j)}$ resulting from $\sigma_{(i..j)}$ by chopping (;) it at the state s_k, then p does not refer to any left state beyond the state s_k, while a formula p being right end closed means that if p holds over a subinterval $\sigma_{(i..k)}$ obtained by chopping ($\bar{;}$) $\sigma_{(i..j)}$ at the state s_k, then p does not refer to (or depend on) any right state beyond the state s_k. For instance, $\Box(more \rightarrow \bigcirc(p \leftrightarrow \ominus q))$ is a *lec*-formula; whereas $\Box(elaps \rightarrow \odot(p \leftrightarrow \bigcirc q))$ is a *rec*-formula (p and q are state formulas). In particular, *empty* is a *lec*-formula and *first* is a *rec*-formula.

The importance of the set of the left end closed formulas lies in guaranteeing $empty; p \equiv p$ and $q \wedge empty; p \equiv p \wedge q$ (p is a lec-formula and q is a rec-formula), which are the basic laws for reduction of *chop* and *while* statements in a program.

We first prove a useful lemma given below:

Lemma 2.20 A non-past formula is a lec-formula, and a non-future formula is a rec-formula.

Proof

We prove only a non-past formula is a *lec*-formula. The proof runs by induction on the structure of p. Since p is a non-past formula, the past operators need not to be considered.

Let (σ, i, k, j) be an interpretation. Then

1. If p is a proposition q.

$$\begin{array}{rl} & (\sigma, i, k, j) \models q \text{ iff } s_k[q] = \text{true} \\ \iff & (\sigma, k, k, j) \models q \end{array}$$

2. If p is $\neg q$.

$$\begin{array}{rl} & (\sigma, i, k, j) \models p \\ \iff & (\sigma, i, k, j) \models \neg q \\ \iff & (\sigma, i, k, j) \not\models q \\ \iff & (\sigma, k, k, j) \not\models q \\ \iff & (\sigma, k, k, j) \models \neg q \\ \iff & (\sigma, k, k, j) \models p \end{array}$$

3. If p is $q_1 \wedge q_2$.

$$\begin{array}{rl} & (\sigma,i,k,j) \models p \\ \Longleftrightarrow & (\sigma,i,k,j) \models q_1 \wedge q_2 \\ \Longleftrightarrow & (\sigma,i,k,j) \models q_1 \text{ and } (\sigma,i,k,j) \models q_2 \\ \Longleftrightarrow & (\sigma,k,k,j) \models q_1 \text{ and } (\sigma,k,k,j) \models q_2 \\ \Longleftrightarrow & (\sigma,k,k,j) \models q_1 \wedge q_2 \\ \Longleftrightarrow & (\sigma,k,k,j) \models p \end{array}$$

4. If p is $\bigcirc q$.

$$\begin{array}{rl} & (\sigma,i,k,j) \models p \\ \Longleftrightarrow & (\sigma,i,k,j) \models \bigcirc q \\ \Longleftrightarrow & (\sigma,i,k+1,j) \models q \text{ and } k < j \\ \Longleftrightarrow & (\sigma,k+1,k+1,j) \models q \\ \Longleftrightarrow & (\sigma,k,k+1,j) \models q \\ \Longleftrightarrow & (\sigma,k,k,j) \models \bigcirc q \\ \Longleftrightarrow & (\sigma,k,k,j) \models p \end{array}$$

5. If p is $q_1;q_2$.

$$\begin{array}{rl} & (\sigma,i,k,j) \models p \\ \Longleftrightarrow & (\sigma,i,k,j) \models q_1;q_2 \\ \Longleftrightarrow & (\sigma,i,k,r) \models q_1 \text{ and } (\sigma,r,r,j) \models q_2 \text{ for some } k \le r \preceq j \\ \Longleftrightarrow & (\sigma,k,k,r) \models q_1 \text{ and } (\sigma,r,r,j) \models q_2 \text{ for some } k \le r \preceq j \\ \Longleftrightarrow & (\sigma,k,k,j) \models q_1;q_2 \\ \Longleftrightarrow & (\sigma,k,k,j) \models p \end{array}$$

6. If p is q^+.

$(\sigma,i,k,j) \models q^+$ iff there are finitely many $r_0, r_1, ..., r_n \in N_\omega$ $(n \ge 1)$ such that $k = r_0 \le r_1 \le ... \le r_{n-1} \preceq r_n = j$ and $(\sigma,i,r_0,r_1) \models q$ and for all $1 < l \le n$, $(\sigma,r_{l-1},r_{l-1},r_l) \models q$; or $j = \omega$ and there are infinitely many integers $k = r_0 \le r_1 \le r_2 \le ...$ such that $\lim_{i\to\infty} r_i = \omega$, and $(\sigma,i,r_0,r_1) \models q$ and, for all $l > 1$, $(\sigma,r_{l-1},r_{l-1},r_l) \models q$.

By hypothesis, the above means that, $(\sigma,i,k,j) \models q^+$ iff there are finitely many $r_0, r_1, ..., r_n \in N_\omega$ $(n \ge 1)$ such that $k = r_0 \le r_1 \le ... \le r_{n-1} \preceq r_n = j$ and $(\sigma,k,r_0,r_1) \models q$ and for all $1 < l \le n$, $(\sigma,r_{l-1},r_{l-1},r_l) \models q$; or $j = \omega$ and there are infinitely many integers $k = r_0 \le r_1 \le r_2 \le ...$ such that $\lim_{i\to\infty} r_i = \omega$, and $(\sigma,k,r_0,r_1) \models q$ and, for all $l > 1$, $(\sigma,r_{l-1},r_{l-1},r_l) \models q$. This is equivalent to $(\sigma,k,k,|\sigma|) \models q^+$.

□

Theorem 2.21 Further Chop Laws

Let w be a rec-formula. Then

$FCH1$	$\bigcirc p;q \equiv \bigcirc(p;q)$	$PCH1$	$p\bar{;}\ominus q \equiv \ominus(p\bar{;}q)$
$FCH2$	$w \wedge (p;q) \equiv (w \wedge p;q)$	$PCH2$	$w \wedge (p\bar{;}q) \equiv (w \wedge p\bar{;}q)$

Proof

We prove only FCH1 and FCH2. Let σ be a model and k an integer $k, 0 \le k \preceq |\sigma|$.

The proof of FCH1

$$
\begin{array}{ll}
 & (\sigma,0,k,|\sigma|)\models\bigcirc p;q \\
\Longleftrightarrow & (\sigma,0,k,r)\models\bigcirc p \text{ and } (\sigma,r,r,|\sigma|)\models q \text{ for some } r, k\leq r\preceq|\sigma| \\
\Longleftrightarrow & (\sigma,0,k+1,r)\models p \text{ and } k<r \text{ and } (\sigma,r,r,|\sigma|)\models q \text{ for some } r, k\leq r\preceq|\sigma|. \\
\Longleftrightarrow & (\sigma,0,k+1,|\sigma|)\models p;q \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models\bigcirc(p;q)
\end{array}
$$

The proof of FCH2

$$
\begin{array}{ll}
 & (\sigma,0,k,|\sigma|)\models w\wedge(p;q) \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models w \text{ and } (\sigma,0,k,|\sigma|)\models p;q \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models w \text{ and } (\sigma,0,k,r)\models p \text{ and } (\sigma,r,r,|\sigma|)\models q \text{ for some } r, k\leq r\preceq|\sigma| \\
\Longleftrightarrow & (\sigma,0,k,r)\models w \text{ and } (\sigma,0,k,r)\models p \text{ and } (\sigma,r,r,|\sigma|)\models q \text{ for some } r, k\leq r\preceq|\sigma| \text{ (lemma 2.20)} \\
\Longleftrightarrow & (\sigma,0,k,r)\models w\wedge p \text{ and } (\sigma,r,r,|\sigma|)\models q \text{ for some } r, k\leq r\preceq|\sigma| \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models w\wedge p;q
\end{array}
$$

□

Example 2.3 The condition of w being a rec-formula for FCH2 cannot be left out. For instance, $len(3)\wedge(len(1);len(2))$ is satisfiable, but $len(3)\wedge len(1);len(2)$ is not. □

Theorem 2.22 Non-End Point Laws

NFE	$\neg first\wedge\neg empty\supset(\bigcirc\ominus p\leftrightarrow\ominus\bigcirc p)$
NF	$\neg first\supset(\ominus p\leftrightarrow\underline{\odot} p)$
NE	$\neg empty\supset(\bigcirc p\leftrightarrow\odot p)$

Proof Straightforward.

□

Theorem 2.23 End Point Laws

$FEP1$	$empty;p\approx p$	$PEP1$	$p\,\bar{;}\,first\approx p$ if p is a rec-formula
$FEP2$	$empty;p\equiv p$ if p is a lec-formula	$PEP2$	$p\,\bar{;}\,first\equiv p$ if p is a rec-formula
$FEP3$	$(p\wedge empty;q)\approx p\wedge q$ if p is a rec-formula	$PEP3$	$(p\,\bar{;}\,q\wedge first)\approx p\wedge q$ if p is a rec-formula
$FEP4$	$(p\wedge empty;q)\supset q$ if q is a lec-formula	$PEP4$	$(p\,\bar{;}\,q\wedge first)\supset q$ if q is a lec-formula
$FEP5$	$(p\wedge empty;q)\equiv p\wedge q$ if p is a rec-formula and q a lec-formula.	$PEP5$	$(p\,\bar{;}\,q\wedge first)\equiv p\wedge q$ if p is a rec-formula and q a lec-formula.
$FEP6$	$p;empty\equiv p\wedge\Diamond empty$	$PEP6$	$first\,\bar{;}\,p\equiv p$
$FEP7$	$\models_{fi}\Box(p;empty\leftrightarrow p)$	$PEP7$	$(p\wedge first\,\bar{;}\,q)\supset q$
$FEP8$	$(p;q\wedge empty)\equiv p\wedge\Box(empty\rightarrow q)$, if q is a lec-formula.	$PEP8$	$(p\wedge first\,\bar{;}\,q)\approx p\wedge q$, if p is a rec-formula.

where $fi=\{\sigma|\ |\sigma|<\omega\}$.

Note that PEP2 simply implies PEP1. PEP1 is added here only for symmetry.

Proof

Let σ be an interval and k an integer, $0 \le k \preceq |\sigma|$.

The proofs of FEP1 and FEP2

$$\begin{array}{ll} & (\sigma,0,k,|\sigma|) \models empty;p \\ \Longleftrightarrow & (\sigma,0,k,r) \models empty \text{ and } (\sigma,r,r,|\sigma|) \models p \text{ for some } r, k \le r \preceq |\sigma| \\ \Longleftrightarrow & k = r \text{ and } (\sigma,r,r,|\sigma|) \models p \\ \Longleftrightarrow & (\sigma,k,k,|\sigma|) \models p \end{array}$$

If p is a lec-formula, then $(\sigma,0,k,|\sigma|) \models empty;p \Longleftrightarrow (\sigma,0,k,|\sigma|) \models p$.

Let $k = 0$ in the proof, it is obvious that $(\sigma,0,0,|\sigma|) \models empty;p$ iff $(\sigma,0,0,|\sigma|) \models p$. Hence, $empty;p \approx p$.

The proofs of FEP3, FEP4 and FEP5

$$\begin{array}{ll} & (\sigma,0,k,|\sigma|) \models p \wedge empty;q \\ \Longleftrightarrow & (\sigma,0,k,r) \models p \wedge empty \text{ and } (\sigma,r,r,|\sigma|) \models q \text{ for some } r, k \le r \preceq |\sigma| \\ \Longleftrightarrow & (\sigma,0,k,r) \models p \text{ and } r = k \text{ and } (\sigma,r,r,|\sigma|) \models q \\ \Longleftrightarrow & (\sigma,0,k,k) \models p \text{ and } (\sigma,k,k,|\sigma|) \models q \end{array}$$

If q is a lec-formula, then $(\sigma,0,k,|\sigma|) \models p \wedge empty;q \Longrightarrow (\sigma,k,k,|\sigma|) \models q \Longleftrightarrow (\sigma,0,k,|\sigma|) \models q$. Hence, $p \wedge empty;q \supset q$.

Furthermore, if p is a rec-formula, then $(\sigma,0,k,|\sigma|) \models p \wedge empty;q \Longleftrightarrow (\sigma,0,k,|\sigma|) \models p \wedge q$.

In the proof, let $k = 0$ and p be a rec-formula. We obtain $(\sigma,0,0,|\sigma|) \models p \wedge empty;q$ iff $(\sigma,0,0,0) \models p$ and $(\sigma,0,0,|\sigma|) \models q$ iff $(\sigma,0,0,|\sigma|) \models p$ and $(\sigma,0,0,|\sigma|) \models q$ iff $(\sigma,0,0,|\sigma|) \models p \wedge q$. That is, $p \wedge empty;q \approx p \wedge q$.

The proof of FEP6

$$\begin{array}{ll} & (\sigma,0,k,|\sigma|) \models p \wedge \Diamond empty \\ \Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p \text{ and } (\sigma,0,k,|\sigma|) \models \Diamond empty \\ \Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p \text{ and } (\sigma,0,r,|\sigma|) \models empty \text{ for some } r, k \le r \preceq |\sigma| \\ \Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p \text{ and } (\sigma,0,r,|\sigma|) \models empty \text{ and } k \le r = |\sigma| < \omega \\ \Longleftrightarrow & (\sigma,0,k,r) \models p \text{ and } (\sigma,r,r,|\sigma|) \models empty \text{ and } k \le r = |\sigma| < \omega \\ \Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p; empty. \end{array}$$

The proof of FEP7

$$\begin{array}{ll} & (\sigma,0,k,|\sigma|) \models p; empty \\ \Longleftrightarrow & (\sigma,0,k,r) \models p \text{ and } (\sigma,r,r,|\sigma|) \models empty \text{ for some } r, k \le r \preceq |\sigma| \\ \Longleftrightarrow & (\sigma,0,k,r) \models p \text{ and } r = |\sigma| < \omega \\ \Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p \text{ and } |\sigma| < \omega. \end{array}$$

The proof of FEP8 will be given in Chapter 4.

The proof of PEP2

$$\begin{array}{ll} & (\sigma,0,k,|\sigma|) \models p\bar{;} first \\ \Longleftrightarrow & (\sigma,0,l,l) \models p \text{ and } (\sigma,l,k,|\sigma|) \models first \text{ for some } l, 0 \le l \le k \\ \Longleftrightarrow & (\sigma,0,l,l) \models p \text{ and } l = k \\ \Longleftrightarrow & (\sigma,0,k,k) \models p \end{array}$$

If p is a rec-formula, then $(\sigma, 0, k, |\sigma|) \models p\bar{;} first \iff (\sigma, 0, k, |\sigma|) \models p$

The proofs of PEP3, PEP4 and PEP5

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models p\bar{;} q \wedge first \\ \iff & (\sigma, 0, l, l) \models p \text{ and } (\sigma, l, k, |\sigma|) \models q \wedge first \text{ for some } l, 0 \leq l \leq k \\ \iff & l = k \text{ and } (\sigma, 0, l, l) \models p \text{ and } (\sigma, l, k, |\sigma|) \models q \\ \iff & (\sigma, 0, k, k) \models p \text{ and } (\sigma, k, k, |\sigma|) \models q \end{array}$$

If q is a lec-formula, then $(\sigma, 0, k, |\sigma|) \models p\bar{;} q \wedge first \Longrightarrow (\sigma, k, k, |\sigma|) \models q \iff (\sigma, 0, k, |\sigma|) \models q$. Hence, $p\bar{;} q \wedge first \supset q$. Furthermore, if p is a rec-formula, then $(\sigma, 0, k, |\sigma|) \models p\bar{;} q \wedge first \iff (\sigma, 0, k, |\sigma|) \models p \wedge q$. Hence $p\bar{;} q \wedge first \equiv p \wedge q$.

In the proof, let $k = 0$ and p be a rec-formula. We obtain $(\sigma, 0, 0, |\sigma|) \models p\bar{;} q \wedge first$ iff $(\sigma, 0, 0, 0) \models p$ and $(\sigma, 0, 0, |\sigma|) \models q$ iff $(\sigma, 0, 0, |\sigma|) \models p$ and $(\sigma, 0, 0, |\sigma|) \models q$ iff $(\sigma, 0, 0, |\sigma|) \models p \wedge q$. That is, $p\bar{;} q \wedge first \approx p \wedge q$.

The proof of PEP6

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models first\bar{;} p \\ \iff & (\sigma, 0, l, l) \models first \text{ and } (\sigma, l, k, |\sigma|) \models p \text{ for some } l, 0 \leq l \leq k \\ \iff & l = 0 \text{ and } (\sigma, l, k, |\sigma|) \models p \\ \iff & (\sigma, 0, k, |\sigma|) \models p \end{array}$$

The proofs of PEP7 and PEP8

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models p \wedge first\bar{;} q \\ \iff & (\sigma, 0, l, l) \models p \wedge first \text{ and } (\sigma, l, k, |\sigma|) \models q \text{ for some } l, 0 \leq l \leq k \\ \iff & l = 0 \text{ and } (\sigma, 0, l, l) \models p \text{ and } (\sigma, l, k, |\sigma|) \models q \\ \iff & (\sigma, 0, 0, 0) \models p \text{ and } (\sigma, 0, k, |\sigma|) \models q \\ \Longrightarrow & (\sigma, 0, k, |\sigma|) \models q \end{array}$$

Let $k = 0$ and p be a rec-formula in the proof. We have $(\sigma, 0, 0, |\sigma|) \models p \wedge first\bar{;} q$ iff $(\sigma, 0, 0, 0) \models p$ and $(\sigma, 0, 0, |\sigma|) \models q$ iff $(\sigma, 0, 0, |\sigma|) \models p$ and $(\sigma, 0, 0, |\sigma|) \models q$ iff $(\sigma, 0, 0, |\sigma|) \models p \wedge q$. That is, $p \wedge first\bar{;} q \approx p \wedge q$. □

Theorem 2.24 Laws about Termination and Absorption

$$TER \quad p \equiv (p; empty) \vee p \wedge \Box more \qquad ABS \quad p \equiv (p; empty) \vee p$$

Proof

The proof of absorption law is straightforward. We prove the termination law only.

The proof of TER

$$\begin{array}{rcll} p & \equiv & p \wedge (\Diamond empty \vee \neg \Diamond empty) & \\ & \equiv & p \wedge \Diamond empty \vee p \wedge \Box more & \text{Abb-alw} \\ & \equiv & (p; empty) \vee p \wedge \Box more & \text{FEP6} \end{array}$$

□

Theorem 2.25 Laws about Chop-Plus

*FPS*1	$p^+ \equiv p \vee (p; p^+)$	*FPS*2	$p \wedge p^+ \equiv p$
*FPS*3	$p \vee p^+ \equiv p^+$	*FPS*4	$p^+; p \supset p; p^+$
*FPS*5	$p; p^+ \supset p^+; p^+$	*FPS*6	$p^+; p^+ \supset p^+$
*FPS*7	$p^{++} \equiv p^+$		

Proof

The proofs of FPS4, FPS5, FPS6, and FPS7 are given in the Appendix. We prove only FPS1, FPS2 and FPS3.

The proof of FPS1

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

Suppose $(\sigma, 0, k, |\sigma|) \models p^+$. Thus, there are finitely many $r_0, ..., r_n \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \preceq r_n = |\sigma|$ and $(\sigma, 0, r_0, r_1) \models p$ and for all $1 < l \leq n$ $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$; or $|\sigma| = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq r_2 \leq ...$ such that $\lim_{r_i \to \infty} r_i = \omega$, and $(\sigma, 0, r_0, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

In the former case, if $n = 1$, then $(\sigma, 0, r_0, r_1) \models p$, i.e. $(\sigma, 0, k, |\sigma|) \models p$; if $n \geq 2$ then $(\sigma, 0, r_0, r_1) \models p$ and for all $1 < l \leq n$ $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$. That is, $(\sigma, 0, r_0, r_1) \models p$ and $(\sigma, r_1, r_1, |\sigma|) \models p^+$. We have $(\sigma, 0, k, |\sigma|) \models p; p^+$.

In the latter case, $(\sigma, 0, k, |\sigma|) \models p^+$ amounts to $(\sigma, 0, r_0, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$. That is, $(\sigma, 0, r_0, r_1) \models p$ and $(\sigma, r_1, r_1, |\sigma|) \models p^+$. We obtain $(\sigma, 0, k, |\sigma|) \models p; p^+$.

Conversely, suppose $(\sigma, 0, k, |\sigma|) \models p \vee (p; p^+)$. If $(\sigma, 0, k, |\sigma|) \models p$ then $(\sigma, 0, k, |\sigma|) \models p^+$ is trivially true since we can simply choose $n = 1$, $r_0 = k$ and $r_1 = |\sigma|$ ignoring σ being finite or infinite.

If $(\sigma, 0, k, |\sigma|) \models p; p^+$, then there is an integer r_1 such that $k \leq r_1 \preceq |\sigma|$ and $(\sigma, 0, k, r_1) \models p$ and $(\sigma, r_1, r_1, |\sigma|) \models p^+$. The latter tells us that either there are finitely many $r_1, r_2, ..., r_n \in N_\omega$ $(n \geq 1)$such that $r_1 \leq r_2 \leq ... \leq r_{n-1} \preceq r_n = |\sigma|$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$ for all $l > 1$; or $|\sigma| = \omega$ and there are infinitely many integers $r_1 \leq r_2 \leq ...$ such that $\lim_{i \to \infty} r_i = \omega$, and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$. Thus, in the former case, we obtain finitely many $r_0, r_1, ..., r_n \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \leq r_{n-1} \preceq r_n = |\sigma|$ and $(\sigma, 0, k, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$; whereas in the latter case, we obtain infinitely many integers $k = r_0 \leq r_1 \leq r_2 \leq ...$ such that $\lim_{i \to \infty} r_i = \omega$, and $(\sigma, 0, k, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$. Hence, in both cases, we obtain $(\sigma, 0, k, |\sigma) \models p^+$.

The proof of FPS2

$$\begin{aligned} p \wedge p^+ &\equiv p \wedge (p \vee (p; p^+)) \\ &\equiv p \wedge p \vee p \wedge (p; p^+) \\ &\equiv p \vee p \wedge (p; p^+) \\ &\equiv p \end{aligned}$$

The proof of FPS3

$$\begin{aligned} p \vee p^+ &\equiv p \vee (p \vee (p; p^+)) \\ &\equiv p \vee (p; p^+) \\ &\equiv p^+ \end{aligned}$$

□

The relationship expressed by the equivalence $p^+ \equiv p \vee (p; p^+)$ could also be discussed within a model supporting fixpoint semantics of logic formulas [8]. However, a detailed evaluation of the problem lies outside the scope of the thesis and is left for future research.

We now further investigate some logic laws regarding the chop star operator. These laws are useful for the reduction of the while statement.

Theorem 2.26 Laws about Chop-Star

$FST1$	$p^* \equiv empty \vee (p; p^*) \vee p \wedge \Box more$	$FST2$	$p^*; p \approx p \vee (p^+; p)$
$FST3$	$p^*; p \supset (empty; p) \vee p^+$	$FST4$	$p^*; p \supset p^+$ if p is a lec-formula.
$FST5$	$p; p^* \supset p^+$	$FST6$	$p^*; p^* \approx p^*$
$FST7$	$p^*; p^* \equiv p^*$ if p is a lec-formula.	$FST8$	$p^{**} \equiv p^*$ if p is a lec-formula.

Proof

The proof of FST1

$$\begin{array}{rcll} p^* & \equiv & empty \vee p^+ & \text{Abb-star} \\ & \equiv & empty \vee (p \vee (p; p^+)) & \text{FPS1} \\ & \equiv & empty \vee ((p; empty) \vee p \wedge \Box more \vee (p; p^+)) & \text{TER} \\ & \equiv & empty \vee (p; (empty \vee p^+)) \vee p \wedge \Box more & \text{FD9} \\ & \equiv & empty \vee (p; p^*) \vee p \wedge \Box more & \text{Abb-star} \end{array}$$

The proof of FST3

$$\begin{array}{rcll} p^*; p & \equiv & (empty \vee p^+); p & \text{Abb-star} \\ & \equiv & (empty; p) \vee (p^+; p) & \text{FD9} \\ & \supset & (empty; p) \vee p^+ & \text{FPS4} \end{array}$$

The proof of FST4

$$\begin{array}{rcll} p^*; p & \equiv & (empty \vee p^+); p & \\ & \equiv & (empty; p) \vee (p^+; p) & \text{FD9} \\ & \equiv & p \vee (p^+; p) & \text{FEP2, } p \text{ is a lec-formula} \\ & \supset & p^+ & \text{FPS4} \end{array}$$

The proof of FST2

$$\begin{array}{rcll} p^*; p & \equiv & (empty \vee p^+); p & \\ & \equiv & (empty; p) \vee (p^+; p) & \text{FD9} \\ & \approx & p \vee (p^+; p) & \text{FEP1} \end{array}$$

The proof of FST5

$$\begin{array}{rcll} p; p^* & \equiv & p; (empty \vee p^+) & \text{Abb-star} \\ & \equiv & (p; empty) \vee (p; p^+) & \text{FD9} \\ & \supset & (p^+; empty) \vee p^+ & p \supset p^+\text{, FPS4} \\ & \equiv & p^+ & \text{ABS} \end{array}$$

The proofs of FST6 and FST7

$$\begin{array}{lll}
p^*;p^* & \equiv (\mathit{empty} \vee p^+);(\mathit{empty} \vee p^+) & \text{Abb-star} \\
 & \equiv (\mathit{empty};\mathit{empty}) \vee (p^+;\mathit{empty}) \vee (\mathit{empty};p^+) \vee (p^+;p^+) & \text{FD9} \\
 & \equiv \mathit{empty} \vee (p^+;\mathit{empty}) \vee (\mathit{empty};p^+) \vee (p^+;p^+) & \text{EMP1} \\
 & \equiv \mathit{empty} \vee (p^+;\mathit{empty}) \vee p^+ \vee (p^+;p^+) & \text{FEP2 and } p \text{ is a lec-formula.} \\
 & (\approx \mathit{empty} \vee (p^+;\mathit{empty}) \vee p^+ \vee (p^+;p^+) & \text{FEP1)} \\
 & \equiv \mathit{empty} \vee p^+ \vee (p^+;p^+) & \text{ABS} \\
 & \equiv \mathit{empty} \vee p^+ & \text{FPS4} \\
 & \equiv p^* & \text{Abb-star}
\end{array}$$

The proof of FST8 is given in the Appendix.

□

Theorem 2.27 Associativity Laws

$$FA1 \quad (p;(q;w)) \equiv ((p;q);w) \qquad PA1 \quad (p\bar{;}(q\bar{;}w)) \equiv ((p\bar{;}q)\bar{;}w)$$

Proof

We prove only FA1. Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of FA1

$$\begin{array}{ll}
 & (\sigma,0,k,|\sigma|) \models (p;(q;w)) \\
\Longleftrightarrow & (\sigma,0,k,r) \models p \text{ and } (\sigma,r,r,|\sigma|) \models q;w \text{ for some } k \leq r \preceq |\sigma|. \\
\Longleftrightarrow & (\sigma,0,k,r) \models p \text{ and } (\sigma,r,r,r') \models q \text{ and } (\sigma,r',r',|\sigma|) \models w \text{ for some } r,r', k \leq r \leq r' \preceq |\sigma|. \\
\Longleftrightarrow & (\sigma,0,k,r') \models p;q \text{ and } (\sigma,r',r',|\sigma|) \models w \text{ for some } r', k \leq r' \leq |\sigma|. \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|) \models (p;q);w
\end{array}$$

□

Finally, we discuss some laws about the negation of the chop construct [24].

Theorem 2.28 Laws about negation of chop construct

$$\begin{array}{llll}
FCHN1 & (p;q) \wedge \neg(p;r) \supset (p;q \wedge \neg r) & PCHN1 & (p\bar{;}q) \wedge \neg(p\bar{;}r) \supset (p\bar{;}q \wedge \neg r) \\
FCHN2 & (p;q) \wedge \neg(r;q) \supset (p \wedge \neg r;q) & PCHN2 & (p\bar{;}q) \wedge \neg(r\bar{;}q) \supset (p \wedge \neg r\bar{;}q) \\
FCHN3 & (len(n);p) \supset \neg(len(n);\neg p) & PCHN3 & (l\bar{e}n(n)\bar{;}p) \supset \neg(l\bar{e}n(n)\bar{;}\neg p) \\
FCHN4 & (p;len(n)) \supset \neg(\neg p;len(n)) & PCHN4 & (p\bar{;}l\bar{e}n(n)) \supset \neg(\neg p\bar{;}l\bar{e}n(n))
\end{array}$$

Proof

We prove only FCHN1 and FCHN3. Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of FCHN1

$$\begin{array}{ll}
 & (\sigma,0,k,|\sigma|) \models (p;q) \wedge \neg(p;r) \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|) \models (p;q) \text{ and } (\sigma,0,k,|\sigma|) \models \neg(p;r) \\
\Longleftrightarrow & (\sigma,0,k,h) \models p \text{ and } (\sigma,h,h,|\sigma|) \models q \text{ for some } k \leq h \preceq |\sigma|\text{, and} \\
 & (\sigma,0,k,h') \not\models p \text{ or } (\sigma,h',h',|\sigma|) \not\models r \text{ for all } k \leq h' \preceq |\sigma| \\
\Longrightarrow & (\sigma,0,k,h) \models p \text{ and } (\sigma,h,h,|\sigma|) \models q \\
 & (\sigma,0,k,h) \models p \text{ and } (\sigma,h,h,|\sigma|) \not\models r \text{ for some } k \leq h \preceq |\sigma| \\
\Longleftrightarrow & (\sigma,0,k,h) \models p \text{ and } (\sigma,h,h,|\sigma|) \models q \wedge \neg r \text{ for some } k \leq h \preceq |\sigma| \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|) \models p;q \wedge \neg r
\end{array}$$

The proof of FCHN3

$$\begin{array}{ll}
 & (\sigma,0,k,|\sigma|) \models (len(n);p) \\
\Longleftrightarrow & (\sigma,0,k,h) \models len(n) \text{ and } (\sigma,h,h,|\sigma|) \models p \text{ for some } k \leq h \preceq |\sigma| \\
\Longleftrightarrow & n = h-k \text{ and } (\sigma,h,h,|\sigma|) \models p \text{ for some } k \leq h \preceq |\sigma| \\
\Longleftrightarrow & (\sigma,n+k,n+k,|\sigma|) \models p
\end{array}$$

Since $(\sigma,0,k,|\sigma|) \not\models \neg(len(n);\neg p)$ is equivalent to $(\sigma,0,k,|\sigma|) \models len(n);\neg p$ which, by the above, is equivalent to $(\sigma,n+k,n+k,|\sigma|) \models \neg p$, we obtain a contradiction. Therefore, FCHN3 holds.

□

Theorem 2.29 Laws about always and negation of chop construct

$FCHAN1$	If $\Box p$ is valid then $\Box(\neg(\neg p;q))$ is valid.
$FCHAN2$	If $\Box p$ is valid then $\Box(\neg(q;\neg p))$ is valid.

Proof

We prove only FCHAN1. Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

Suppose $\Box p$ is valid. Thus, for every model σ, $\sigma \models \Box p$. That is, by Theorem A.8, for every interpretation $\mathcal{I}$, $\mathcal{I} \models p$.

If $\Box(\neg(\neg p;q))$ is not valid, then there is a σ and integer k such that $(\sigma,0,k,|\sigma|) \not\models \neg(\neg p;q)$. This is equivalent to $(\sigma,0,k,|\sigma|) \models \neg p;q$. By I-chop, there exists a r, $k \leq r \preceq |\sigma|$ such that $(\sigma,0,k,r) \models \neg p$ and $(\sigma,r,r,|\sigma|) \models q$ leading to $(\sigma,0,k,r) \models \neg p$. This is a contradiction.

□

In this chapter, we have investigated logic laws which will provide a basis of the underlying logic. In particular, these logic laws allow us to establish a model theory for the first order extended ITL (see Chapter 3) and to reduce (framed) programs built from the extended Tempura (see Chapters 4, 6,7). However, we have not established an axiom system as this is beyond the scope of this thesis.

Chapter 3

First Order Temporal Logic

Summary: An extended first order interval temporal logic is formalized by adding past operators and infinite models to ITL. The syntax and semantics of the extended logic are presented. The logic laws regarding variables, the equality, and quantifications are formulated and proved.

Temporal logic can be developed from its propositional basis to a first order predicate logic in a way analogous to the way in which this is done in the classical case. In this chapter, we present an extended ITL (EITL) which combines linear time temporal logic [66, 51] with chop [78, 42, 11], and is an extension of the interval temporal logic [61].

This chapter is organized as follows: Section 3.1 presents the syntax of EITL. Section 3.2 presents the semantics of EITL. To this end, first, states and intervals are defined. Then interpretations of terms and formulas are given in detail. Section 3.3 defines satisfaction and validity of formulas in EITL. In Section 3.4, a considerable collection of logic laws concerning variables, the equality and quantifications are formalized and proved. The proofs of a number of useful theorems exploited later in the thesis are based on these laws.

3.1 Syntax

1. Terms

   Let *Prop* be a countable set of propositions, and V a countable set of typed variables. We assume the variables are partitioned into static and dynamic ones. $B = \{\text{true}, \text{false}\}$ represents the boolean domain, D denotes all data needed by us including integers, lists, sets etc. Z denotes all integers, N denotes positive integers, N_0 stands for non-negative integers, and $N_\omega = N_0 \cup \{\omega\}$. Terms are inductively constructed as follows:

   1) Individual typed static variables: $a, b, c, u, v, \ldots$ possibly with subscripts.
   2) Individual typed dynamic (or state) variables: $x, y, z, X, Y, Z, \ldots$ possibly with subscripts.
   3) Functions: if f is a function of arity m ($m \geq 0$) and $e_1, \ldots, e_m$ are terms over V of types compatible with the types of arguments of f, then $f(e_1, \ldots, e_m)$ is a term over V with the type defined by f. In particular, when $m = 0$, f is a constant term.
   4) If e is a term, then $\bigcirc e, \ominus e$, $beg(e)$ and $end(e)$ are terms with definite types.

2. Atomic Formulas

    1) Every proposition $p \in Prop$ is an atomic formula.
    2) If P is a primitive predicate of arity m $(m > 0)$ and $e_1, ..., e_m$ are terms over V of types compatible with the types of arguments of P, then $P(e_1, ..., e_m)$ is an atomic formula over V.
    3) If e_1, e_2 are terms of the same type, then $e_1 = e_2$ is an atomic formula.

3. Basic Formulas

    1) Every atomic formula is a basic formula.
    2) If p and q are basic formulas, so are the following constructs:

$$\neg p, p \wedge q, \exists x : p, \bigcirc p, \Diamond p, p ; q, \ominus p, \underline{\Diamond} p, p \bar{;} q, p^{+}.$$

    where $\bigcirc$, $\Diamond$ and $;$ are the basic future operators whereas $\ominus$, $\underline{\Diamond}$ and $\bar{;}$ are the basic past operators.

Note that, in the following, the types of terms will not be considered in detail, and all terms are assumed to be 'well-formed'.

The constants, functions, and predicates are concrete individual elements, concrete functions, and concrete predicates over their respective domains [66]. Similarly as in [66], we have constants true, false over B; -1, -2, 0, 1, 2,... over Z; ϵ (the empty list) over lists; and ϕ (the empty set) over sets. We have the function symbols +, - over Z; . (the concatenation of two lists), $\circ$ (the fusion of two lists), *hd* (taking the head of a nonempty list), *tl* (taking the tail of a nonempty list), *lt* (taking the last element of a nonempty list) over lists (see Chapter 4 for the details); and $\cup$, $\cap$ over sets. We also have predicates $>$, $<$ over Z; and $\subseteq$, $\in$ over sets.

3.2 Semantics

1. States

    We define a state s over $V \cup Prop$ to be a pair (I_v, I_p) of state interpretations I_v and I_p. I_v assigns each variable $x \in V$ a value in D or *nil* (undefined) and the total domain is denoted by $D' = D \cup \{nil\}$, whereas I_p assigns each proposition $p \in Prop$ a truth value in B. $s[v]$ denotes the value of v at state s.

2. Intervals

    Basically, the notation for intervals is the same as that defined in Chapter 2. Some notations will also be used without declaration under the assumption that they are the same as defined in Chapter 2. *It is assumed that a static variable remains the same over an interval whereas a dynamic variable can have different values at different states.* To evaluate the existential quantification, an equivalence relation is required and given below. We use I_v^k and I_p^k to denote the state interpretations at state s_k.

Definition 3.1 Two intervals, σ and σ', are x-equivalent, denoted by $\sigma' \overset{x}{=} \sigma$, if $|\sigma| = |\sigma'|$, $I_v^h[y] = I_v'^h[y]$ for all $y \in V - \{x\}$, and $I_p^h[p] = I_p'^h[p]$ for all $p \in Prop$ $(0 \leq h \preceq |\sigma|)$. □

Note that, (1) $\overset{x}{=}$ is an equivalence relation over the set of all intervals; that is, $\overset{x}{=}$ is reflexive, symmetric, and transitive; (2) $\overset{x}{=}$ relation is extensible; that is, if $< s_i, ..., s_l >$ $\overset{x}{=}$ $< s'_i, ..., s'_l >$ and $< s_l, ..., s_j >$ $\overset{x}{=}$ $< s'_l, ..., s'_j >$ then $< s_i, ..., s_j >$ $\overset{x}{=}$ $< s'_i, ..., s'_j >$ $(i \leq l \preceq j)$; (3) $\overset{x}{=}$ relation is partitionable; that is, if $< s_i, ..., s_j >$ $\overset{x}{=}$ $< s'_i, ..., s'_j >$, then $< s_i, ..., s_l >$ $\overset{x}{=}$ $< s'_i, ..., s'_l >$ and $< s_l, ..., s_j >$ $\overset{x}{=}$ $< s'_l, ..., s'_j >$ for all l, $i \leq l \preceq j$.

3. Interpretations

An interpretation, as for EPITL, is a quadruple $\mathcal{I} = (\sigma, i, k, j)$, where σ is an interval, $i, k \in N_0$ and $j \in N_\omega$, and $0 \leq i \leq k \preceq j \leq |\sigma|$. We use notation (σ, i, k, j) to mean that a formula or term is interpreted over a subinterval $< s_i, ..., s_j >$ of σ with the current state being s_k.

For every term e, the evaluation of e relative to interpretation $\mathcal{I} = (\sigma, i, k, j)$, denoted by $\mathcal{I}[e]$, is defined by induction on terms in the following way:

$$\begin{array}{lll}
I - varu & \mathcal{I}[a] & = s_k[a] = I_v^k[a] = I_v^i[a] \text{ if } a \text{ is a static variable.} \\
I - varx & \mathcal{I}[x] & = s_k[x] = I_v^k[x] \text{ if } x \text{ is a dynamic variable.} \\
I - fun & \mathcal{I}[f(e_1, ..., e_m)] & = \begin{cases} nil & \text{if } \mathcal{I}[e_h] = nil, \text{ for some } h \in \{1, ..., m\} \\ f(\mathcal{I}[e_1], ..., \mathcal{I}[e_m]) & \text{otherwise} \end{cases} \\
I - enext & \mathcal{I}[\bigcirc e] & = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases} \\
I - eprev & \mathcal{I}[\circleddash e] & = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases} \\
I - beg & \mathcal{I}[beg(e)] & = (\sigma, i, i, j)[e] \\
I - end & \mathcal{I}[end(e)] & = \begin{cases} (\sigma, i, j, j)[e] & \text{if } j < \omega \\ nil & \text{otherwise} \end{cases}
\end{array}$$

The meaning of formulas is given by the satisfaction relation, $\models$, which is inductively defined as follows:

$$\begin{array}{lll}
I - prop & \mathcal{I} \models & p \text{ iff } I_p^k[p] = \text{true, for any given proposition } p. \\
I - pred & \mathcal{I} \models & P(e_1, ..., e_m) \text{ iff } P \text{ is a primitive predicate other than } = \text{ and, for} \\
 & & \text{all } h,\ 0 \leq h \leq m, \mathcal{I}[e_h] \neq nil \text{ and } P(\mathcal{I}[e_1], ..., \mathcal{I}[e_m]) = \text{true.} \\
I - equal & \mathcal{I} \models & e_1 = e_2 \text{ iff } e_1 \text{ and } e_2 \text{ are terms and } \mathcal{I}[e_1] = \mathcal{I}[e_2]. \\
I - neg & \mathcal{I} \models & \neg p \text{ iff } \mathcal{I} \not\models p. \\
I - and & \mathcal{I} \models & p \wedge q \text{ iff } \mathcal{I} \models p \text{ and } \mathcal{I} \models q. \\
I - next & \mathcal{I} \models & \bigcirc p \text{ iff } k < j \text{ and } (\sigma, i, k+1, j) \models p. \\
I - som & \mathcal{I} \models & \Diamond p \text{ iff there exists } r \text{ such that } k \leq r \preceq j \text{ and } (\sigma, i, r, j) \models p. \\
I - chop & \mathcal{I} \models & (p; q) \text{ iff there exists } r \text{ such that } k \leq r \preceq j \text{ and } (\sigma, i, k, r) \models p \text{ and} \\
 & & (\sigma, r, r, j) \models q.
\end{array}$$

I − prev $\quad \mathcal{I} \models \ominus p$ iff $i < k$ and $(\sigma, i, k-1, j) \models p$.

I − somp $\quad \mathcal{I} \models \diamondminus p$ iff there exists l such that $i \leq l \leq k$ and $(\sigma, i, l, j) \models p$.

I − chopp $\quad \mathcal{I} \models (p \bar{;} q)$ iff there exists l such that $i \leq l \leq k$ and $(\sigma, i, l, l) \models p$ and $(\sigma, l, k, j) \models q$.

I − plus $\quad \mathcal{I} \models p^+$ iff there are finitely many $r_0, ..., r_n \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \leq r_{n-1} \preceq r_n = j$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $1 < l \leq n, (\sigma, r_{l-1}, r_{l-1}, r_l) \models p$; or $j = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq r_2 \leq ...$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

I − exists $\quad \mathcal{I} \models \exists x : p$ iff there exists an interval σ' such that $\sigma'_{(i..j)} \overset{x}{=} \sigma_{(i..j)}$ and $(\sigma', i, k, j) \models p$.

4. Abbreviations

The abbreviations for *true*, *false*, $\neq$, $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. We also define *chop-star*:

$$(Abb - star) \quad p^* \quad \overset{\text{def}}{=} \quad empty \vee p^+$$

The semantics of the chop star construct is the same as in EPITL. The universal quantifier is defined as usual:

$$(Abb - all) \qquad \forall x : p \overset{\text{def}}{=} \neg \exists x : \neg p$$

Its semantics is given by

$$(\sigma, i, k, j) \models \forall x : p \text{ iff } (\sigma', i, k, j) \models p \text{ for every } \sigma', \ \sigma'_{(i..j)} \overset{x}{=} \sigma_{(i..j)}.$$

The usual convention of abbreviating multiple quantifiers of the same kind, e.g. $\forall x : \forall y : p$ to $\forall xy : p$ is also used. Furthermore, we use the abbreviations introduced in Chapter 2.

5. Precedence Rules

In order to avoid an excessive number of parentheses, the following precedence rules are used:

1	$\neg$
2	$\bigcirc, \odot, \Diamond, \Box, \ominus, \odot, \diamondminus, \boxminus, +, *$
3	$\exists, \forall$
4	$=$
5	$\wedge, \vee$
6	$\rightarrow, \leftrightarrow$
7	$;, \bar{;}$

where 1=highest and 7=lowest.

A formula (or term) is called a state formula (or term) if it does not contain any temporal operators; otherwise it is a temporal formula (or term).

3.3 Satisfaction and Validity

A model is an interval which can be finite or infinite.

1. Satisfaction and Validity

   A formula p is satisfied by an interval σ, denoted by $\sigma\models p$, if $(\sigma,0,0,|\sigma|)\models p$. A formula p is called satisfiable if $\sigma \models p$ for some σ. A formula p is valid, denoted by $\models p$, if $\sigma\models p$ for all σ.

2. Special Models

   In some cases, we need to define a set R of models satisfying some properties. A formula p is called R-satisfiable, if there exists a model $\sigma\in R$ such that $\sigma\models p$. A formula p is said to be R-valid if for all intervals σ in R, $\sigma \models p$. In this case, the satisfaction relation $\models_R$ will be used instead of $\models$.

Like Kripke's structure for EPITL, in a model of EITL, I_v^k can be given by a set of pairs in the form $x_i : e_i$, where $x_i \in V$, $e_i \in D'$. The pair $x_i : e_i$ means that $I_v^k[x_i] = e_i$. In the examples, when the value of a variable x_i is irrelevant, the pair $x_i : e_i$ may not be shown. Thus, a empty set ϕ of such pairs means that all variables are irrelevant.

Example 3.1 Evaluate the formula p: $x = 1\land\bigcirc x = x + 1; \bigcirc(y = 3 * \ominus x)$ according to the interval given in Fig.3.1, i.e. $\sigma =< (I_v^0, I_p^0), \ldots >, I_v^0 = \{x : 1, y : 2\}, I_v^1 = \{x : 2, y : 4\}, \ldots$. We do not need to specify I_p^i in this example.

k	0	1	2	3 ...
x	1	2	3	4 ...
y	2	4	6	8 ...

Fig.3.1 An interval

$(\sigma,0,0,|\sigma|)\models p \iff (\sigma,0,0,r)\models x = 1\land\bigcirc x = x+1$ and $(\sigma,r,r,|\sigma|)\models\bigcirc(y = 3*\ominus x)$ for some $r, 0\le r\preceq|\sigma|$

Let $r = 1$. We have

$$\begin{array}{ll}
 & (\sigma,0,0,1)\models x = 1\land\bigcirc x = x+1 \\
\iff & (\sigma,0,0,1)\models x = 1 \text{ and } (\sigma,0,0,1)\models\bigcirc x = x+1 \\
\iff & (\sigma,0,0,1)[x] = (\sigma,0,0,1)[1] \text{ and } (\sigma,0,0,1)[\bigcirc x] = (\sigma,0,0,1)[x] + (\sigma,0,0,1)[1] \\
\iff & s_0[x] = 1 \text{ and } (\sigma,0,1,1)[x] = s_0[x] + 1 \\
\iff & I_v^0[x] = 1 \text{ and } I_v^1[x] = I_v^0[x] + 1 \\
\iff & 1 = 1 \text{ and } 2 = 1 + 1 \\
\iff & \mathit{true} \\
 & (\sigma,1,1,|\sigma|)\models\bigcirc(y = 3*\ominus x) \\
\iff & (\sigma,1,2,|\sigma|) \models y = 3*\ominus x \\
\iff & (\sigma,1,2,|\sigma|)[y] = 3*(\sigma,1,2,|\sigma|)[\ominus x] \\
\iff & s_2[y] = 3*(\sigma,1,1,|\sigma|)[x] \\
\iff & I_v^2[y] = 3*I_v^1[x] \\
\iff & 6 = 3*2 \\
\iff & \mathit{true}
\end{array}$$

Therefore $\sigma\models p$. $\square$

3.4 Logic Laws

The logic laws provided in Chapter 2 for EPITL are all available in the first order temporal logic EITL. In this chapter, we present some new logic laws concerning variables, the equality and quantifications.

3.4.1 Basic Theorems

First, we claim that Theorems (Corollary) 2.1 - 2.7 still hold for EITL. Their proofs are similar as in the preceding chapter but quantifications, predicates and the equality in the proofs have to be considered. We refer to them as Theorems (Corollary) 3.1 - 3.7, respectively. Note that the additional proof for Theorem 3.7 is given in the Appendix.

Theorem 3.1 If p is a tautologically valid formula, then $\models \Box p$. □

Theorem 3.2 p is valid iff $\neg p$ is not satisfiable. □

Theorem 3.3 Let $\mathcal{I}$ be an interpretation. If $\mathcal{I}\models p$ and $\mathcal{I}\models p\rightarrow q$ then $\mathcal{I}\models q$. □

Theorem 3.4 Let $\mathcal{I}$ be an interpretation. If $\mathcal{I}\models q$ implies $\mathcal{I}\models p$, then $\mathcal{I} \models q\rightarrow p$. □

Theorem 3.5 If p_1, p_2, q_1, q_2 are formulas, then

$FM0$	If $(p_1 \supset p_2)$ and $(q_1 \hookrightarrow q_2)$ then $((p_1;q_1) \supset (p_2;q_2))$
$PM0$	If $(p_1 \supset p_2)$ and $(q_1 \supset q_2))$ then $((p_1 \bar{;} q_1) \supset (p_2 \bar{;} q_2))$

□

Corollary 3.6

$FM1$	If $p_1 \supset p_2$ then $(p_1;q) \supset (p_2;q)$	$PM1$	If $p_1 \supset p_2$ then $(p_1 \bar{;} q) \supset (p_2 \bar{;} q)$
$FM2$	If $p_1 \hookrightarrow p_2$ then $(q;p_1) \supset (q;p_2)$	$PM2$	If $p_1 \supset p_2$ then $(q \bar{;} p_1) \supset (q \bar{;} p_2)$
$FM3$	If $p_1 \equiv p_2$ and $q_1 \approx q_2$ then $(p_1;q_1) \equiv (p_2;q_2)$	$PM3$	If $p_1 \equiv p_2$ and $q_1 \equiv q_2$ then $(p_1 \bar{;} q_1) \equiv (p_2 \bar{;} q_2)$

□

Theorem 3.7 If F is a formula in EITL involving the subformula g, and f is a formula such that $f \equiv g$, then

$$F \equiv F[f/g]$$

where $F[f/g]$ denotes the formula given by replacing some occurrences of g in F by f. □

3.4.2 Quantifications and Temporal Operators

Let us first introduce the notion of variable binding [51]. An occurrence of a variable x in some formula p is called *bound* if it occurs in a sub-formula $\exists x : q$ (or $\forall x : q$) of p. Otherwise it is called *free*. If t is a term then $p[t/x]$ denotes the result of a simultaneous substitution of t for every free occurrence of x in p. When writing $p[t/x]$ we always assume implicitly that t does not contain a variable which occurs bound in p (this can be achieved by replacing the bound variables of p by fresh variables).

Theorem 3.8 states some logic laws concerning quantifications and future temporal operators.

Theorem 3.8 The following formulas hold:

$$\begin{array}{llcllll}
FQT1 & \bigcirc(\exists x : p) & \equiv & \exists x : \bigcirc p & FQT5 & \Diamond\exists x : p \equiv & \exists x : \Diamond p \\
FQT2 & \bigcirc(\forall x : p) & \equiv & \forall x : \bigcirc p & FQT6 & \Box\forall x : p \equiv & \forall x : \Box p \\
FQT3 & \odot(\exists x : p) & \equiv & \exists x : \odot p & FQT7 & \exists x : \Box p \supset & \Box\exists x : p \\
FQT4 & \odot(\forall x : p) & \equiv & \forall x : \odot p & FQT8 & \Diamond\forall x : p \supset & \forall x : \Diamond p
\end{array}$$

Proof

The proofs of FQT2, FQT4, FQT6 and FQT8 are similar to FQT1, FQT3, FQT5 and FQT7, respectively. So, we prove only FQT1, FQT3, FQT5 and FQT7. Let σ be a model and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of FQT1:

We need to prove $(\sigma, 0, k, |\sigma|) \models \bigcirc(\exists x : p) \leftrightarrow \exists x : \bigcirc p$. If $k = |\sigma|$, the conclusion is vacuously true. If $k < |\sigma|$, we have,

$$\begin{array}{lll}
 & (\sigma, 0, k, |\sigma|) \models \bigcirc(\exists x : p) & \\
\Longleftrightarrow & (\sigma, 0, k+1, |\sigma|) \models \exists x : p & \text{I-next} \\
\Longleftrightarrow & (\sigma', 0, k+1, |\sigma'|) \models p \text{ for some } \sigma' \overset{x}{=} \sigma & \text{I-exists} \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models \bigcirc p \text{ for some } \sigma' \overset{x}{=} \sigma & \text{I-next} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \exists x : \bigcirc p & \text{I-exists}
\end{array}$$

The proof of FQT3:

We need to prove $(\sigma, 0, k, |\sigma|) \models \odot\exists x : p \leftrightarrow \exists x : \odot p$.

$$\begin{array}{lll}
 & (\sigma, 0, k, |\sigma|) \models \odot\exists x : p & \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models empty \vee \bigcirc\exists x : p & \text{Abb-wnext} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \bigcirc\exists x : p \text{ or } k = |\sigma| & \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \exists x : \bigcirc p \text{ or } k = |\sigma| & \text{theorem 3.8 (1)} \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models \bigcirc p \text{ or } k = |\sigma| \text{ for some } \sigma' \overset{x}{=} \sigma & \text{I-exists} \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models \bigcirc p \vee empty \text{ for some } \sigma' \overset{x}{=} \sigma & \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models \odot p \text{ for some } \sigma' \overset{x}{=} \sigma & \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \exists x : \odot p & \text{I-exists}
\end{array}$$

The proof of FQT5:

$$
\begin{array}{lll}
 & (\sigma,0,k,|\sigma|)\models\Diamond(\exists x:p) & \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|)\models\exists x:p \text{ for some } r, k\leq r\preceq|\sigma| & \text{I-som} \\
\Longleftrightarrow & (\sigma',0,r,|\sigma'|)\models p \text{ for some } r,\ k\leq r\preceq|\sigma|, \text{ and for some } \sigma', \sigma'\overset{x}{=}\sigma & \text{I-exists} \\
\Longleftrightarrow & (\sigma',0,k,|\sigma'|)\models\Diamond p \text{ for some } \sigma\overset{x}{=}\sigma' & \text{I-som} \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|)\models\exists x:\Diamond p & \text{I-exists}
\end{array}
$$

The proof of FQT7:

$$
\begin{array}{lll}
 & (\sigma,0,k,|\sigma|)\models\exists x:\Box p & \\
\Longleftrightarrow & (\sigma',0,k,|\sigma'|)\models\Box p \text{ for some } \sigma', \sigma'\overset{x}{=}\sigma & \text{I-exists} \\
\Longleftrightarrow & (\sigma',0,r,|\sigma'|)\models p \text{ for some } \sigma'\overset{x}{=}\sigma, \text{ for all } r, k\leq r\preceq|\sigma'| & \text{Abb-alw} \\
\Longrightarrow & (\sigma,0,r,|\sigma|)\models\exists x:p \text{ for all } r, k\leq r\preceq|\sigma| & \text{I-exists} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models\Box\exists x:p & \text{Abb-alw}
\end{array}
$$

□

Example 3.2 shows that the always operator (□) does not distribute over the existential quantification.

Example 3.2 Relative to the interval given in Fig 3.2, the formula $\Box\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$ holds but $\exists x:\Box(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$ is false. The justification is as follows:

$$
\begin{array}{ll}
 & \sigma\models\Box\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2) \\
\Longleftrightarrow & (\sigma,0,0,|\sigma|)\models\Box\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2) \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|)\models\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2) \text{ for all integer } r, 0\leq r\preceq|\sigma|
\end{array}
$$

Since $|\sigma|=2$, $r=0,1,2$. When $r=2$, the formula is 'vacuously' true. So, we need only to consider the cases $r=0$ and $r=1$.

Case 1: $r=0$

$$
\begin{array}{ll}
 & (\sigma,0,0,2)\models\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2) \\
\Longleftrightarrow & (\sigma',0,0,2)\models more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2 \text{ for some } \sigma',\ \sigma'\overset{x}{=}\sigma
\end{array}
$$

We can construct such a σ' given in Fig 3.3. We have

$$(\sigma',0,0,2)\models more$$

and

$$
\begin{array}{ll}
 & (\sigma',0,0,2)\models x>0\wedge x^2+\bigcirc x^2=y^2 \\
\Longleftrightarrow & (\sigma',0,0,2)\models s'_0[x]>0 \text{ and } (s'_0[x])^2+(s'_1[x])^2=(s'_0[y])^2 \\
\Longleftrightarrow & (\sigma',0,0,2)\models 3>0 \text{ and } 3^2+4^2=5^2 \\
\Longleftrightarrow & (\sigma',0,0,2)\models true
\end{array}
$$

Therefore,

$$(\sigma,0,0,2)\models\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$$

Case 2: $r=1$

$$(\sigma,0,1,2)\models\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$$
$$\Longleftrightarrow\ (\sigma'',0,1,2)\models more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2 \text{ for some } \sigma'',\ \sigma''\overset{x}{=}\sigma.$$

We can construct such a σ'' given in Fig 3.4. We have

$$(\sigma'',0,1,2)\models more$$

and

$$(\sigma'',0,1,2)\models x>0\wedge x^2+\bigcirc x^2=y^2$$
$$\Longleftrightarrow\ (\sigma'',0,1,2)\models s_1''[x]>0 \text{ and } (s_1''[x])^2+(s_2''[x])^2=(s_1''[y])^2$$
$$\Longleftrightarrow\ (\sigma'',0,1,2)\models 5>0 \text{ and } 5^2+12^2=13^2$$
$$\Longleftrightarrow\ (\sigma'',0,1,2)\models true$$

Therefore,

$$(\sigma,0,1,2)\models\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$$

By cases 1,2, we have

$$\sigma\models\Box\exists x:(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$$

However,

$$\sigma\not\models\exists x:\Box(more\rightarrow x>0\wedge x^2+\bigcirc x^2=y^2)$$

because we cannot construct one model σ' such that $(s_0'[x])^2+(s_1'[x])^2=(s_0'[y])^2$ and $(s_1'[x])^2+(s_2'[x])^2=(s_1'[y])^2$ and $s_0'[x]>0$ and $s_1'[x]>0$ and $s_2'[x]>0$ and $\sigma'\overset{x}{=}\sigma$ since there is no positive integer solution for the equation system: $a^2+b^2=25$ and $b^2+c^2=169$.

```
   |------|------|
x=1       2      1
y=5       13     5
```

Fig 3.2

```
   |------|------|
x=3       4      5
y=5       13     5
```

Fig 3.3

```
   |------|------|
x=3       5      12
y=5       13     5
```

Fig 3.4

$\square$

Example 3.3 We now show that the sometimes operator does not distribute over the universal quantification, that is, in general,

$$\Diamond(\forall x : p) \equiv \forall x : \Diamond p$$

does not hold. Consider the two formulas, $\Diamond(\forall u : p)$ and $\forall u : \Diamond p$, in the case that p stands for $x \neq u$ and where variable u is static, and variable x is dynamic.

Consider the interval relative to Fig 3.5. It is easy to show that $\sigma \models \forall u : \Diamond(x \neq u)$. To see this, we have to show that, for any $\sigma', \sigma' \overset{u}{=} \sigma$, i.e. $\sigma' =< (I_v'^0, I_p'^0), ... >$ for $I_v'^i = \{x : i, u : a\}$, and an arbitrary I_p^i, for all $i, 0 \leq i \preceq |\sigma'|$, and for every possible value of a, σ' satisfies $\Diamond(x \neq u)$.

We consider two cases:

(1) if $a = 0$ then $(\sigma', 0, 1, |\sigma'|) \models x \neq u$ as $x = 1, u = 0$ at s_1' of σ'.
(2) if $a \neq 0$ then $(\sigma', 0, 0, |\sigma'|) \models x \neq u$ as $x = 0, u = a \neq 0$ at s_0' of σ'.

On the other hand, the state formula $\forall u : x \neq u$ is false at any state of σ, since we can always take $u = x$ at that state. It follows that $\sigma \not\models \Diamond\forall u : (x \neq u)$, which shows that the formulas are not equivalent.

```
|------------|------------|--
x=0          1            2 ...
u=10         10           10 ...
```

Fig 3.5

□

Theorem 3.9 concerns the past operators, its proof is similar to Theorem 3.8.

Theorem 3.9 The following formulas are valid:

$PQT1$	$\ominus(\exists x : p) \equiv \exists x : \ominus p$	$PQT5$	$\Diamond\exists x : p \equiv \exists x : \Diamond p$
$PQT2$	$\ominus(\forall x : p) \equiv \forall x : \ominus p$	$PQT6$	$\boxminus\forall x : p \equiv \forall x : \boxminus p$
$PQT3$	$\odot(\exists x : p) \equiv \exists x : \odot p$	$PQT7$	$\exists x : \boxminus p \supset \boxminus\exists x : p$
$PQT4$	$\odot(\forall x : p) \equiv \forall x : \odot p$	$PQT8$	$\Diamond\forall x : p \supset \forall x : \Diamond p$

Theorem 3.11 is devoted to developing the laws which are relevant to the chop operator. First, we prove an auxiliary Lemma.

Lemma 3.10 If $\sigma_{(i..j)} \overset{v}{=} \sigma'_{(i..j)}$, $0 \leq i \leq k \preceq j \leq |\sigma|$, and q is a formula without free occurrences of the (dynamic or static) variable v, then $(\sigma, i, k, j) \models q$ iff $(\sigma', i, k, j) \models q$.

Proof

We need to prove that, any bound occurrence of v in the form of $\exists v : w$ contained in the formula q is evaluated to the same truth value over any interpretations (σ, i_1, k_1, j_1) and (σ', i_1, k_1, j_1) $(i_1 \geq i, j_1 \leq j)$. Since $\sigma_{(i..j)} \overset{v}{=} \sigma'_{(i..j)}$, $\sigma_{(i_1..j_1)} \overset{v}{=} \sigma'_{(i_1..j_1)}$.

If q contains $\exists v : w$ as a subformula, then

$$\begin{array}{ll} & (\sigma, i_1, k_1, j_1) \models \exists v : w \\ \Longleftrightarrow & (\sigma'', i_1, k_1, j_1) \models w \text{ for some } \sigma'',\ \sigma''_{(i_1..j_1)} \overset{v}{=} \sigma_{(i_1..j_1)} \\ \Longleftrightarrow & (\sigma'', i_1, k_1, j_1) \models w \text{ for some } \sigma'',\ \sigma''_{(i_1..j_1)} \overset{v}{=} \sigma'_{(i_1..j_1)}\ (\sigma'_{(i_1..j_1)} \overset{x}{=} \sigma_{(i_1..j_1)}) \\ \Longleftrightarrow & (\sigma', i_1, k_1, j_1) \models \exists v : w \end{array}$$

□

Theorem 3.11 Let p, q be formulas. The following hold:

$FQC1$	$(\exists x : p; q)$	$\equiv$	$\exists x : (p; q)$	x does not occur freely in q
$FQC2$	$(p; \exists x : q)$	$\equiv$	$\exists x : (p; q)$	x does not occur freely in p
$FQC3$	$(\forall x : p; q)$	$\supset$	$\forall x : (p; q)$	x does not occur freely in q
$FQC4$	$(p; \forall x : q)$	$\supset$	$\forall x : (p; q)$	x does not occur freely in p

Proof

The proofs of FQC2 and FQC4 are similar to FQC1 and FQC3. So, we prove only FQC1 and FQC3. Let σ be a model, and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of FQC1:

$(\sigma, 0, k, |\sigma|) \models (\exists x : p; q)$

$\Longleftrightarrow$ $(\sigma, 0, k, r) \models \exists x : p$ and $(\sigma, r, r, |\sigma|) \models q$ for some $r, k \leq r \preceq |\sigma|$. I-chop

$\Longleftrightarrow$ $(\sigma', 0, k, r) \models p$ and $(\sigma, r, r, |\sigma|) \models q$ for some r, $k \leq r \preceq |\sigma|$ and for some σ', $\sigma'_{(0..r)} \overset{x}{=} \sigma_{(0..r)}$. I-exists

$\Longleftrightarrow$ $(\sigma', 0, k, r) \models p$ and $(\sigma', r, r, |\sigma'|) \models q$ for some r, $k \leq r \preceq |\sigma|$ and for some σ', $\sigma' \overset{x}{=} \sigma$. lemma 3.10

$\Longleftrightarrow$ $(\sigma', 0, k, |\sigma'|) \models p; q$ for some σ', $\sigma' \overset{x}{=} \sigma$. I-chop

$\Longleftrightarrow$ $(\sigma, 0, k, |\sigma|) \models \exists x : (p; q)$ lemma 3.10

The proof of FQC3:

$(\sigma, 0, k, |\sigma|) \models \forall x : p; q$.

$\Longleftrightarrow$ $(\sigma, 0, k, r) \models \forall x : p$ and $(\sigma, r, r, |\sigma|) \models q$ for some integer $r, k \leq r \preceq |\sigma|$

$\Longleftrightarrow$ $(\sigma', 0, k, r) \models p$ and $(\sigma, r, r, |\sigma|) \models q$ for some r, $k \leq r \preceq |\sigma|$, and for every σ', $\sigma'_{(0..r)} \overset{x}{=} \sigma_{(0..r)}$.

$\Longrightarrow$ $(\sigma', 0, k, r) \models p$ and $(\sigma', r, r, |\sigma'|) \models q$ for every σ', $\sigma' \overset{x}{=} \sigma$ and some r, $k \leq r \preceq |\sigma|$ (lemma 3.10).

$\Longleftrightarrow$ $(\sigma', 0, k, |\sigma'|) \models p; q$ for every σ', $\sigma' \overset{x}{=} \sigma$.

$\Longleftrightarrow$ $(\sigma, 0, k, |\sigma|) \models \forall x : (p; q)$.

□

Example 3.4 According to Theorem 3.11, the following formula holds:

$$\exists x : \bigcirc(x = 7); \bigcirc(y = 8) \equiv \exists x : (\bigcirc(x = 7); \bigcirc(y = 8))$$

□

Example 3.5 and Example 3.6 show that the universal quantifier does not distribute over the chop operator.

Example 3.5 Relative to the interval given in Fig 3.5, the formula $\forall u : (\Diamond(x = u \wedge u \geq 0); true)$ holds, but the formula $\forall u : \Diamond x = u \wedge u \geq 0; true$ is false, where u is a static variable. We prove this as follows

$$\begin{array}{ll} & \sigma\models\forall u : (\Diamond(x = u \wedge u \geq 0); true) \\ \Longleftrightarrow & (\sigma,0,0,|\sigma|)\models\forall u : (\Diamond(x = u \wedge u \geq 0); true) \\ \Longleftrightarrow & (\sigma',0,0,|\sigma'|)\models\Diamond(x = u \wedge u \geq 0); true \text{ for every } \sigma',\ \sigma'\overset{u}{=}\sigma. \\ \Longleftrightarrow & (\sigma',0,0,r_{\sigma'})\models\Diamond(x = u \wedge u \geq 0) \text{ and } (\sigma',r_{\sigma'},r_{\sigma'},|\sigma'|)\models true \text{ for every } \sigma',\ \sigma'\overset{u}{=}\sigma, \text{ and} \\ & \text{some } r_{\sigma'} \text{ relevant to the } \sigma', 0\leq r_{\sigma'}\preceq|\sigma'|. \\ \Longleftrightarrow & (\sigma',0,0,r_{\sigma'})\models\Diamond(x = u \wedge u \geq 0) \text{ for every } \sigma',\ \sigma'\overset{u}{=}\sigma \text{ and some } r_{\sigma'}. \\ \Longleftrightarrow & (\sigma',0,i,r_{\sigma'})\models x = u \wedge u \geq 0 \text{ for every } \sigma',\ \sigma'\overset{u}{=}\sigma, \text{ and some } r_{\sigma'}, i, 0\leq i \leq r_{\sigma'}. \\ \Longleftrightarrow & (\sigma',0,i,r_{\sigma'})\models s'_i[x] = s'_i[u] = s'_0[u] \geq 0 \text{ for every } \sigma'\overset{u}{=}\sigma, \text{ and some } r_{\sigma'}, i, 0 \leq i \preceq |\sigma|. \end{array}$$

For each given u, $\sigma'\overset{u}{=}\sigma$, we can choose an $r_{\sigma'}$ to satisfy the above condition. In fact, let $r_{\sigma'} = s'_0[u]$ and $i = r_{\sigma'}$, then $s'_i[x] = i = r_{\sigma'} = s'_0[u] = s'_i[u]$. However, we cannot construct one interval $\sigma'\overset{u}{=}\sigma$ for all u to satisfy the above condition.

$$\begin{array}{ll} & \sigma\models\forall u : \Diamond(x = u \wedge u \geq 0); true \\ \Longleftrightarrow & (\sigma,0,0,|\sigma|)\models\forall u : \Diamond(x = u \wedge u \geq 0); true \\ \Longleftrightarrow & (\sigma,0,0,r)\models\forall u : \Diamond(x = u \wedge u \geq 0) \text{ and } (\sigma,r,r,|\sigma|)\models true, \text{ for some } r, 0\leq r\preceq|\sigma|. \\ \Longleftrightarrow & (\sigma,0,0,r)\models\forall u : \Diamond(x = u \wedge u \geq 0) \text{ for some } r,\ 0 \leq r \preceq |\sigma|. \\ \Longleftrightarrow & (\sigma',0,0,r)\models\Diamond(x = u \wedge u \geq 0) \text{ for some } r,\ 0 \leq r \preceq |\sigma|, \text{ and every } \sigma', \sigma'_{(0..r)}\overset{u}{=}\sigma_{(0..r)}. \\ \Longleftrightarrow & (\sigma',0,i,r)\models x = u \wedge u \geq 0 \text{ for some } r, 0 \leq r \preceq |\sigma|, \text{ for every } \sigma', \sigma'_{(0..r)}\overset{u}{=}\sigma_{(0..r)}, \text{ and for} \\ & \text{some } i, 0\leq i\leq r \preceq |\sigma|. \\ \Longleftrightarrow & s'_i[x] = s'_i[u] \geq 0 \text{ for some } r, 0 \leq r \preceq |\sigma|, \text{ for every } \sigma',\ \sigma'_{(0..r)}\overset{u}{=}\sigma_{(0..r)}, \text{ and some} \\ & i, 0 \leq i \leq r \preceq |\sigma|. \\ \Longleftrightarrow & s'_i[x] = s'_0[u] \geq 0 \text{ for some } r, 0 \leq r \preceq |\sigma|, \text{ for every } \sigma',\ \sigma'_{(0..r)}\overset{u}{=}\sigma_{(0..r)}, \text{ and some} \\ & i, 0 \leq i \leq r \preceq |\sigma|. \\ \Longleftrightarrow & i = s'_0[u] \geq 0 \text{ for some } r, 0 \leq r \preceq |\sigma|, \text{ for every } \sigma',\ \sigma'_{(0..r)}\overset{u}{=}\sigma_{(0..r)}, \text{ and some} \\ & i, 0 \leq i \leq r \preceq |\sigma|. \end{array}$$

Since $0\leq i\leq r$, when $s'_0[u] > r$, the above condition cannot be satisfied. □

Example 3.6 As proved in Theorem 3.11 (FQC4), $p; \forall x : q \supset \forall x : (p; q)$, x is not free in p; but the reverse does not hold. The justification is simple: if $\forall x : (p; q) \supset p; \forall x : q$ held, then by taking p to be *true*, we would have

$$\forall x : (true; q) \ \supset\ (true; \forall x : q)$$

If q and $\forall x : q$ are lec-formulas, then we can rewrite this as

$$\forall x : \Diamond q \supset \Diamond\forall x : q$$

Thus, taking x to be a static variable u, and q to be $x \neq u$, the above contradicts Example 3.3. □

Theorem 3.12 is similar to Theorem 3.11 but is concerned with chop in the past operator.

Theorem 3.12 Let p, q be formulas. The following formulas are valid:

$$\begin{array}{lllll} PQC1 & (\exists x : p\bar{;}q) & \equiv & \exists x : (p\bar{;}q) & x \text{ does not occur freely in } q. \\ PQC2 & (p\bar{;}\exists x : q) & \equiv & \exists x : (p\bar{;}q) & x \text{ does not occur freely in } p. \\ PQC3 & (\forall x : p\bar{;}q) & \supset & \forall x : (p\bar{;}q) & x \text{ does not occur freely in } q. \\ PQC4 & (p\bar{;}\forall x : q) & \supset & \forall x : (p\bar{;}q) & x \text{ does not occur freely in } p. \end{array}$$

3.4.3 Values of Terms

In this section, we present some logic laws concerning the values of variables at the previous, next, first and last states over an interval. First, we introduce some notations used in the sequel.

Definition 3.2

1. $keep(p) \overset{\text{def}}{=} \Box(\neg empty \rightarrow p)$
2. $remain(p) \overset{\text{def}}{=} \Box(\neg first \rightarrow p)$
3. $inner(p) \overset{\text{def}}{=} \Box(\neg first \wedge \neg empty \rightarrow p)$

$\Box$

keep(p) holds over an interval as long as p holds at all states ignoring the last one, while *remain*(p) holds over an interval as long as p holds at all states ignoring the first one, and *inner*(p) holds over an interval if p holds at all states ignoring the first and last ones.

Theorem 3.13 Let e_1, e_2 be terms. Then

$$\begin{array}{lll} EQ1 & \models & keep(\bigcirc e_1 = \bigcirc e_2 \leftrightarrow \bigcirc(e_1 = e_2)) \\ EQ2 & \models & remain(\ominus e_1 = \ominus e_2 \leftrightarrow \ominus(e_1 = e_2)) \\ EQ3 & \models & inner(\bigcirc\ominus e = \ominus\bigcirc e) \end{array}$$

Proof

We prove only EQ1 here. To this end, we need to prove $(\sigma, 0, r, |\sigma|) \models more \rightarrow (\bigcirc e_1 = \bigcirc e_2 \leftrightarrow \bigcirc(e_1 = e_2))$ for every σ and r, $0 \leq r \preceq |\sigma|$. If $r = |\sigma|$, the conclusion is trivially true. If $r < |\sigma|$, then $(\sigma, 0, r, |\sigma|) \models more$. Hence,

$$\begin{array}{ll} & (\sigma, 0, r, |\sigma|) \models \bigcirc e_1 = \bigcirc e_2 \\ \Longleftrightarrow & (\sigma, 0, r, |\sigma|)[\bigcirc e_1] = (\sigma, 0, r, |\sigma|)[\bigcirc e_2] \\ \Longleftrightarrow & (\sigma, 0, r+1, |\sigma|)[e_1] = (\sigma, 0, r+1, |\sigma|)[e_2] \\ \Longleftrightarrow & (\sigma, 0, r+1, |\sigma|) \models e_1 = e_2 \\ \Longleftrightarrow & (\sigma, 0, r, |\sigma|) \models \bigcirc(e_1 = e_2) \end{array}$$

$\Box$

Lemma 3.14 Let e be a term and f a function of arity one. Then

$$\begin{array}{ll} Fun1 & \models \Box(f(\bigcirc e) = \bigcirc f(e)) \\ Fun2 & \models \Box(f(\ominus e) = \ominus f(e)) \end{array}$$

Proof

We prove only Fun1. Let σ be an interval and r an integer, $0 \leq r \preceq |\sigma|$. If $r = |\sigma|$, $(\sigma, 0, r, |\sigma|) \models f(\bigcirc e) = nil = \bigcirc f(e)$. If $r < |\sigma|$, then we have

$$\begin{array}{ll} & (\sigma, 0, r, |\sigma|) \models (f(\bigcirc e) = \bigcirc f(e)) \\ \Longleftrightarrow & (\sigma, 0, r, |\sigma|)[f(\bigcirc e)] = (\sigma, 0, r, |\sigma|)[\bigcirc f(e))] \\ \Longleftrightarrow & f((\sigma, 0, r, |\sigma|)[\bigcirc e)]) = (\sigma, 0, r+1, |\sigma|)[f(e)] \\ \Longleftrightarrow & f((\sigma, 0, r+1, |\sigma|)[e]) = f((\sigma, 0, r+1, |\sigma|)[e])] \\ \Longleftrightarrow & true \end{array}$$

$\Box$

Corollary 3.15 Let $e_1, ..., e_m$ be terms and f a function of arity m $(m > 0)$. Then

$$\begin{array}{ll} Fun3 & \models \Box(f(\bigcirc e_1, ..., \bigcirc e_m) = \bigcirc f(e_1, ..., e_m)) \\ Fun4 & \models \Box(f(\ominus e_1, ..., \ominus e_m) = \ominus f(e_1, ..., e_m)) \end{array}$$

□

Notice that although we consider only the name and arity of a function, a function (called *function-expression*) can be defined compositionally in a complicated way. For example, +, -, *, / are functions of arity two over Z, a composite function f of arity two over Z can be defined as $f(x,y) \overset{\text{def}}{=} (x + y) * ((x - y)/(x * y))$. Therefore, in what follows, whenever a proof involves a function, it is sufficient to take only its name and arity into account. Note also that, in the definition of a composite function, the parameters of the function are only the free variables; whereas in the invocation of such a function, the parameters are replaced by terms (arguments). In a similar way, we can define a *term-expression* and *formula-expression* with only free variables as their parameters. In the applications of term-expressions and formula-expressions, the parameters can be replaced only by terms (arguments). Certainly, a term-expression (formula-expression) is also a term (formula).

Lemma 3.16 Let u be a static variable and x a dynamic or static variable. Let f be a function of arity two. Then

$$\begin{array}{ll} Fun5 & \models \Box(f(u, \bigcirc x) = \bigcirc f(u, x)) \\ Fun6 & \models \Box(f(u, \ominus x) = \ominus f(u, x)) \end{array}$$

Proof

We prove only Fun5. Let σ be an interval and r an integer, $0 \leq r \preceq |\sigma|$. If $r = |\sigma|$, $(\sigma, 0, r, |\sigma|) \models f(u, \bigcirc x) = nil = \bigcirc f(u, x)$. If $r < |\sigma|$, then we have

$$\begin{array}{ll} & (\sigma, 0, r, |\sigma|) \models (f(u, \bigcirc x) = \bigcirc f(u, x)) \\ \Longleftrightarrow & (\sigma, 0, r, |\sigma|)[f(u, \bigcirc x)] = (\sigma, 0, r, |\sigma|)[\bigcirc f(u, x))] \\ \Longleftrightarrow & f((\sigma, 0, r, |\sigma|)[u], (\sigma, 0, r, |\sigma|)[\bigcirc x)]) = (\sigma, 0, r+1, |\sigma|)[f(u, x)] \\ \Longleftrightarrow & f((\sigma, 0, r+1, |\sigma|)[u], (\sigma, 0, r+1, |\sigma|)[x]) = f((\sigma, 0, r+1, |\sigma|)[u], (\sigma, 0, r+1, |\sigma|)[x]) \\ \Longleftrightarrow & true \end{array}$$

□

Lemma 3.17 If u is a static variable, then

$$\begin{array}{ll} VarS1 & \models keep(u = \bigcirc u) \\ VarS2 & \models remain(u = \ominus u) \end{array}$$

Proof Straightforward.

□

Lemma 3.18 Let u be a static variable and x a static or dynamic variable. Let $e(u, x)$ denote a state term-expression which can be a constant or a variable u or x or a function-expression $f(u)$, or $f(x)$ or $f(u, x)$, then

$$\begin{array}{ll} TerS1 & \models keep(e(u, \bigcirc x) = \bigcirc e(u, x)) \\ TerS2 & \models remain(e(u, \ominus x) = \ominus e(u, x)) \end{array}$$

Proof

We prove only TerS1. The proof proceeds by case analysis of the term $e(u,x)$.

1. If $e(u,x)$ is a constant c, then the conclusion is obvious.
2. If $e(u,x)$ is a static variable u, then by Lemma 3.17, the conclusion holds.
3. If $e(u,x)$ is a dynamic variable x, then the conclusion trivially holds.
4. If $e(u,x)$ contains u and x, then $e(u,x)$ is a function-expression $f(x)$, $f(u)$ or $f(u,x)$ then by Lemma 3.14, 3.16 and 3.17, the conclusion holds.

□

Theorem 3.19 Let u be a static variable and x a static or dynamic variable. Let $\varphi(u,x)$ denote a state formula-expression that contains free variables u or x or both as its only variables. Then

$$\begin{array}{lll} ForS1 & \models & keep(\varphi(u, \bigcirc x) \leftrightarrow \bigcirc\varphi(u,x))) \\ ForS2 & \models & remain(\varphi(u, \ominus x) \leftrightarrow \ominus\varphi(u,x))) \end{array}$$

Proof

We prove only ForS1. Let σ be an interval and r an integer, $0 \leq r \preceq |\sigma|$. If $r = |\sigma|$, the conclusion is obvious. We assume $|\sigma| > r$. The proof proceeds by induction on the structure of state formulas. Since $\varphi(u,x)$ contains free variables u, or x or both as its only variables, it is impossible for $\varphi(u,x)$ to contain quantifiers. Also, it does not contain any temporal operators. Thus,

1. $\varphi(u,x)$ is a primitive predicate $p(u,x)$.
The proofs for the cases of $p(u)$ and $p(x)$ are similar.

$$\begin{array}{lll} & (\sigma,0,r,|\sigma|) \models \varphi(u,\bigcirc x) & \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models p(u,\bigcirc x) & \\ \Longleftrightarrow & p((\sigma,0,r,|\sigma|)[u],(\sigma,0,r,|\sigma|)[\bigcirc x]) = \text{true} & \\ \Longleftrightarrow & p((\sigma,0,r,|\sigma|)[u],(\sigma,0,r+1,|\sigma|)[x]) = \text{true} & \\ \Longleftrightarrow & p((\sigma,0,r+1,|\sigma|)[u],(\sigma,0,r+1,|\sigma|)[x]) = \text{true} & u \text{ is static} \\ \Longleftrightarrow & (\sigma,0,r+1,|\sigma|) \models p(u,x) & \text{I-pred} \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc p(u,x) & \text{I-next} \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc\varphi(u,x) & \end{array}$$

2. $\varphi(u,x)$ is $e_1(u,x) = e_2(u,x)$

$$\begin{array}{lll} & (\sigma,0,r,|\sigma|) \models \varphi(u,\bigcirc x) & \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models e_1(u,\bigcirc x) = e_2(u,\bigcirc x) & \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc e_1(u,x) = \bigcirc e_2(u,x) & \text{lemma 3.18} \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc(e_1(u,x) = e_2(u,x)) & \text{theorem 3.13} \end{array}$$

3. $\varphi(u,x)$ is $\neg\psi(u,x)$

$$\begin{array}{lll} & (\sigma,0,r,|\sigma|) \models \varphi(u,\bigcirc x) & \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \neg\psi(u,\bigcirc x) & \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \neg\bigcirc\psi(u,x) & \text{hypothesis} \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc\neg\psi(u,x) & r > 0 \\ \Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc\varphi(u,x) & \end{array}$$

4. $\varphi(u,x)$ is $\psi_1(u,x)\wedge\psi_2(u,x)$

$$\begin{array}{ll}
 & (\sigma,0,r,|\sigma|) \models \varphi(u,\bigcirc x) \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \psi_1(u,\bigcirc x)\wedge\psi_2(u,\bigcirc x) \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc\psi_1(u,x)\wedge\bigcirc\psi_2(u,x) \qquad \text{hypothesis} \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc(\psi_1(u,x)\wedge\psi_2(u,x)) \qquad \text{FD3} \\
\Longleftrightarrow & (\sigma,0,r,|\sigma|) \models \bigcirc\varphi(u,x)
\end{array}$$

□

Theorem 3.20 Let $u_1,...,u_m$ be static variables and $x_1,...,x_n$ be static or dynamic variables. Let $\varphi(u_1,...,u_m,x_1,...,x_n)$ denote a state formula-expression that contains free variables u_i $(1 \le i \le m)$ or x_j $(1 \le j \le n)$ or all of them as its only variables. Then

$$\begin{array}{lll}
ForS3 & \models & keep(\varphi(u_1,...,u_m,\bigcirc x_1,...,\bigcirc x_n) \leftrightarrow \bigcirc\varphi(u_1,...,u_m,x_1,...,x_n)) \\
ForS4 & \models & remain(\varphi(u_1,...,u_m,\odot x_1,...,\odot x_n) \leftrightarrow \odot\varphi(u_1,...,u_m,x_1,...,x_n))
\end{array}$$

Proof

Similar to the proof of Theorem 3.19 which can easily be generalised. □

3.4.4 Replacement of Variables

Definition 3.3 A formula (or term) is called static if it does not refer to any dynamic variable.

□

Definition 3.4 Let τ be a formula or term. If t is a term and x a variable used in τ, then $\tau[t/x]$ denotes the result of simultaneous replacement of all free occurrences of x by t in τ. The replacement is called compatible if either x and t are static or x is dynamic. □

In the following, we write $\tau(x)$ to imply that τ has one or more occurrences of variable x and that there is no quantification over x.

Definition 3.5 The replacement $\tau[t/x]$ is called admissible for $\tau(x)$ if it is compatible and none of the variables appearing in t is quantified in τ. We also say that t is admissible for x in $\tau(x)$ and write $\tau(t)$ to denote $\tau[t/x]$. □

Example 3.7 Let x be a dynamic variable and u a static variable, then the substitution of x for u in $\exists x : (x \neq u)$ is not admissible. □

Theorem 3.21 (replacement of equals by equals in terms)

For a state term-expression $e(x)$ and terms, t_1, t_2, that are admissible for x in $e(x)$, we have

$$STer1 \quad t_1 = t_2 \supset e(t_1) = e(t_2)$$

Proof

The proof proceeds by induction on the structure of $e(x)$. Obviously, we do not need to consider constant terms. Let $\mathcal{I} = (\sigma, 0, k, |\sigma|)$ be an interpretation.

1. $e(x)$ is a variable.

$$\begin{array}{rl} & \mathcal{I} \models t_1 = t_2 \\ \iff & \mathcal{I}[t_1] = \mathcal{I}[t_2] \\ \iff & \mathcal{I}[e(t_1)] = \mathcal{I}[e(t_2)] \\ \iff & \mathcal{I} \models e(t_1) = e(t_2) \end{array}$$

2. $e(x)$ is a (possibly composite) function $f(x)$.

$$\begin{array}{rl} & \mathcal{I} \models t_1 = t_2 \\ \iff & \mathcal{I}[t_1] = \mathcal{I}[t_2] \\ \implies & f(\mathcal{I}[t_1]) = f(\mathcal{I}[t_2]) \\ \iff & \mathcal{I}[f(t_1)] = \mathcal{I}[f(t_2)] \\ \iff & \mathcal{I} \models f(t_1) = f(t_2) \end{array}$$

□

Corollary 3.22 For state term-expressions $e_1(x), e_2(x)$ and terms t_1, t_2, that are admissible for x in $e_1(x), e_2(x)$, we have

$$STer2 \quad t_1 = t_2 \land e_1(t_1) = e_2(t_1) \supset e_1(t_2) = e_2(t_2)$$

Proof

Let σ be an interval and r an integer, $0 \leq r \preceq |\sigma|$. Suppose $(\sigma, 0, r, |\sigma|) \models t_1 = t_2 \land e_1(t_1) = e_2(t_1), 0 \leq r \preceq |\sigma|$. By Theorem 3.21, we have

$$(\sigma, 0, r, |\sigma|) \models e_1(t_1) = e_1(t_2) \text{ and } (\sigma, 0, r, |\sigma|) \models e_2(t_1) = e_2(t_2)$$

Since $(\sigma, 0, r, |\sigma|) \models e_1(t_1) = e_2(t_1)$, we obtain

$$(\sigma, 0, r, |\sigma|) \models e_1(t_2) = e_2(t_2)$$

□

Note that the condition, $e(x)$ being a state term, is necessary for Theorem 3.21. For example, taking $e(x)$ to be $\bigcirc x$, and t_1, t_2 to be x, y , respectively, $x = y$ does not imply $\bigcirc x = \bigcirc y$. However, if $t_1 = t_2$ holds throughout an interval, then the conclusion can be changed.

Theorem 3.23 If $e(x)$ is a term-expression, and t_1, t_2 are terms that are admissible for x in $e(x)$, then

$$STer3 \quad \text{if } \models \Box(t_1 = t_2) \text{ then } \models \Box(e(t_1) = e(t_2))$$

Proof

The proof is by induction on the structure of term e. Let σ be an interval and k an integer $0 \leq k \preceq |\sigma|$. Then, $(\sigma, 0, k, |\sigma|) \models t_1 = t_2$.

1. If $e(x)$ is a state term, by Theorem 3.21, the conclusion holds.
2. If $e(x)$ is $\bigcirc e_1(x)$, we need to prove $(\sigma, 0, k, |\sigma|) \models \bigcirc e_1(t_1) = \bigcirc e_1(t_2)$. If $k = |\sigma|$, then the conclusion is obvious. Suppose $k < |\sigma|$. Then $(\sigma, 0, k, |\sigma|) \models \bigcirc e_1(t_1) = \bigcirc e_1(t_2)$ iff $(\sigma, 0, k+1, |\sigma|) \models e_1(t_1) = e_1(t_2)$. By the induction hypothesis, $(\sigma, 0, k+1, |\sigma|) \models e_1(t_1) = e_1(t_2)$.

3. If $e(x)$ is $\ominus e_1(x)$, a similar proof as that for $\bigcirc e_1$ can be made.
4. If $e(x)$ is $beg(e_1(x))$. We need to prove $(\sigma,0,k,|\sigma|)\models beg(e_1(t_1)) = beg(e_1(t_2))$.

$$\begin{aligned}
& (\sigma,0,k,|\sigma|) \models beg(e_1(t_1)) = beg(e_1(t_2)) \\
\Longleftrightarrow\ & (\sigma,0,k,|\sigma|)[beg(e_1(t_1))] = (\sigma,0,k,|\sigma|)[beg(e_1(t_2))] \\
\Longleftrightarrow\ & (\sigma,0,0,|\sigma|)[e_1(t_1)] = (\sigma,0,0,|\sigma|)[e_1(t_2)] \\
\Longleftrightarrow\ & (\sigma,0,0,|\sigma|) \models e_1(t_1) = e_1(t_2)
\end{aligned}$$

By the induction hypothesis, $(\sigma,0,k,|\sigma|) \models e_1(t_1) = e_1(t_2)$ for all k, $0 \le k \preceq |\sigma|$. So, $(\sigma,0,0,|\sigma|) \models e_1(t_1) = e_1(t_2)$.

5. If $e(x)$ is $end(e_1(x))$, a similar proof as that for $beg(e_1(x))$ can be made.
6. If $e(x)$ is a function $f(x)$, then it is already discussed in 1.

□

Theorem 3.24 (replacement of equals by equals in formulas)

If $p(x)$ is a state formula-expression and t_1, t_2 are terms which are admissible for $p(x)$, then

$$SFor1 \quad t_1 = t_2 \supset (p(t_1) \leftrightarrow p(t_2))$$

Proof

Let σ be a model and r an integer, $0 \le r \preceq |\sigma|$. The proof proceeds by induction on the structure of state formula p.

Suppose $(\sigma,0,r,|\sigma|)\models t_1 = t_2$. We need to prove $(\sigma,0,r,|\sigma|)\models p(t_1) \leftrightarrow (\sigma,0,r,|\sigma|)\models p(t_2)$.

(1) $p(x)$ is a primitive predicate $q(x)$.

Since $(\sigma,0,r,|\sigma|) \models t_1 = t_2$, then

$$\begin{aligned}
& (\sigma,0,r,|\sigma|) \models p(t_1) \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models q(t_1) \\
\Longleftrightarrow\ & q((\sigma,0,r,|\sigma|)[t_1]) = \text{true} \\
\Longleftrightarrow\ & q((\sigma,0,r,|\sigma|)[t_2]) = \text{true} \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models q(t_2) \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models p(t_2)
\end{aligned}$$

(2) $p(x)$ is $e(x) = c$ and c is a constant in D. Then,

$$\begin{aligned}
& (\sigma,0,r,|\sigma|) \models p(t_1) & \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models e(t_1) = c & \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models e(t_2) = e(t_1) = c & \text{theorem 3.21} \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models p(t_2) &
\end{aligned}$$

(3) $p(x)$ is $e_1(x) = e_2(x)$. Then,

$$\begin{aligned}
& (\sigma,0,r,|\sigma|) \models p(t_1) & \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models e_1(t_1) = e_2(t_1) & \\
\Longleftrightarrow\ & (\sigma,0,r,|\sigma|) \models e_1(t_2) = e_2(t_2) & \text{corollary 3.22}
\end{aligned}$$

(4) $p(x)$ is $\neg q(x)$. Then,

$$(\sigma,0,r,|\sigma|) \models p(t_1)$$

$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models\neg q(t_1)$
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models\neg q(t_2)$ hypothesis, theorem 3.7
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models p(t_2)$

(5) $p(x)$ is $q_1(x)\wedge q_2(x)$. Then,

$(\sigma,0,r,|\sigma|)\models p(t_1)$
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models q_1(t_1)\wedge q_2(t_1)$
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models q_1(t_2)\wedge q_2(t_2)$ hypothesis, theorem 3.7
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models p(t_2)$

(6) $p(x)$ is $\exists y:q(x)$. Then,

$(\sigma,0,r,|\sigma|)\models p(t_1)$
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models\exists y:q(t_1)$
$\Longleftrightarrow$ $(\sigma',0,r,|\sigma|)\models q(t_1)$ for some $\sigma'\overset{y}{=}\sigma$
$\Longleftrightarrow$ $(\sigma',0,r,|\sigma|)\models q(t_2)$ for some $\sigma'\overset{y}{=}\sigma$ hypothesis
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models\exists y:q(t_2)$
$\Longleftrightarrow$ $(\sigma,0,r,|\sigma|)\models p(t_2)$

□

In the above proof, it is important that y is not used by t_1 and t_2 since this allows us to have $(\sigma',0,r,|\sigma|)\models t_1=t_2$, so induction hypothesis can be applied.

We will show that the two restrictions required by the theorem are essential for its validity. Consider first the case $p(x)$ is not a state formula. We may then take $p(x)$ to be $\bigcirc(x=10)$ and t_1,t_2 to be y,z, respectively, and obtain

$$\models\Box(y=z\rightarrow(\bigcirc(y=10)\leftrightarrow\bigcirc(z=10)))$$

To see the above is not valid, it is sufficient to consider the sequence

$$\sigma=<(I_v^0,I_p^0),(I_v^1,I_p^1)>\ \text{ and }I_v^0=\{y:10,z:10\},I_v^1=\{y:10,z:20\}$$

We observe that σ satisfies $y=z$ and $\bigcirc(y=10)$ but not $\bigcirc(z=10)$ at state s_0.

Next consider the case that formula $p(x)$ quantifies over some of the variables of t_1,t_2. For example, take $p(x)$ to be $\exists y:(y>x)$, and t_1,t_2 to be y,z, respectively, then

$$\models\Box(y=z\rightarrow(\exists y:y>y\leftrightarrow\exists y:y>z))$$

Clearly, $\exists y:y>y$ is always false, while if x and y range over the integers, $\exists y:y>z$ is always true.

We can obtain a stronger notion of substitutivity by requiring that $e_1=e_2$ holds not only at a single position, but throughout the interval.

Theorem 3.25 Let $p(x)$ be a formula with a free variable x and let t_1,t_2 be terms that are admissible for $p(x)$. Then the following holds:

$$SFor2\quad\text{if }\models\Box(t_1=t_2)\text{ then }p(t_1)\equiv p(t_2)$$

Proof Similar to the proof of Theorem 3.24.

□

Theorem 3.26 For a state formula $p(x_1, ..., x_m)$, a formula $q(x_1, ..., x_m)$, a state term $e(x_1, ..., x_m)$, and terms $t_1, ..., t_m, e_1, ..., e_m$, that are admissible for $x_1, ..., x_m$ in $p(x_1, ..., x_m)$ and $e(x_1, ..., x_m)$, the following hold:

$$STer4 \quad (t_1 = e_1 \wedge ... \wedge t_m = e_m) \supset (e(t_1, ..., t_m) = e(e_1, ..., e_m))$$

$$STer5 \quad (t_1 = e_1 \wedge ... \wedge t_m = e_m) \supset (p(t_1, ..., t_m) \leftrightarrow p(e_1, ..., e_m))$$

$$SFor6 \quad if \models \Box(t_1 = e_1 \wedge ... \wedge t_m = e_m) \text{ then } (q(t_1, ..., t_m) \equiv q(e_1, ..., e_m))$$

Proof Similar to the proof in the case $m = 1$.

□

3.4.5 Quantifications

The logic laws presented in this section are concerned with the universal quantification.

Theorem 3.27 For a (static or dynamic) variable v, a formula $p(v)$, and a state term t that is admissible for v in $p(v)$ we have

$$UQgo1 \quad \forall v : p(v) \supset p(t)$$

Proof

Let σ be a model and k an integer, $0 \leq k \preceq |\sigma|$.

$$\begin{array}{llll} & (\sigma, 0, k, |\sigma|) & \models & \forall v : p(v) \\ \Longleftrightarrow & (\sigma', 0, k, |\sigma'|) & \models & p(v) \text{ for any } \sigma', \sigma' \overset{v}{=} \sigma. \quad \text{Abb-all} \\ \Longrightarrow & (\sigma', 0, k, |\sigma'|) & \models & p(v) \text{ for some } \sigma' \text{ with } (\sigma', 0, k, |\sigma'|)[v] = (\sigma, 0, k, |\sigma|)[t]. \\ \Longrightarrow & (\sigma', 0, k, |\sigma'|) & \models & p(v) \text{ iff } (\sigma, 0, k, |\sigma|) \models p(t) \text{ for this } \sigma'. \\ \Longrightarrow & (\sigma, 0, k, |\sigma|) & \models & p(t) \end{array}$$

□

It is easy to come up with counterexamples for cases in which the required restrictions are violated.

Example 3.8 For a static variable v and a variable x, $\forall v \exists x : (x \neq v)$ is valid.

Taking t to be x which is not admissible for $\exists x : (x \neq v)$, we obtain for $p(t)$ the contradictory formula $\exists x : (x \neq x)$. □

Example 3.9 Let v be a static variable and x a dynamic variable. The formula

$$\forall v : \Diamond(x \neq v)$$

is satisfiable (see Example 3.3). If we take t to be x which is not admissible for v, we obtain an invalid formula $p(x) : \Diamond x \neq x$. Therefore, the condition of the term t being admissible for v in $p(v)$ in Theorem 3.27 is necessary.

□

Theorem 3.27 can be strengthened. In fact, a static variable, say u, can be replaced by a dynamic state term as long as the replacement of u does not create new occurrences of dynamic variable in the scope of temporal operators. We discuss this in the following.

Example 3.10

$$\forall u : (u = x \rightarrow u = \bigcirc x) \supset (y = x \rightarrow y = \bigcirc x)$$

holds, where u is a static variable and y is a dynamic variable. □

Definition 3.6 A state term t with the property that its substitution for a static variable u in a formula p does not create new occurrences of dynamic variables in the scope of temporal operators and none of the variables in t is bound in p is called ***substitutable for u in p***. □

Theorem 3.28 Let t be substitutable for u in p. Then

$$UQgo2 \quad \forall u : p(u) \supset p[t/u]$$

Proof Similar ro the proof of Theorem 3.27.

□

In Theorem 3.28, the condition for t to be substitutable for u in p is necessary. A counter-example is given in Example 3.11.

Example 3.11 Consider the case that p is $u = x \rightarrow \bigcirc(u = x)$, taking t to be a dynamic variable y, and σ to be such that

$$(\sigma, 0, 0, |\sigma|)[x] = (\sigma, 0, 0, |\sigma|)[y] = (\sigma, 0, 1, |\sigma|)[x] \neq (\sigma, 0, 1, |\sigma|)[y]$$

then$(\sigma, 0, 0, |\sigma|) \models \forall u : (u = x \rightarrow \bigcirc(u = x))$ since $(\sigma, 0, 0, |\sigma|)[x] = (\sigma, 0, 1, |\sigma|)[x]$, and $(\sigma, 0, 0, |\sigma|) \models y = x$ and $(\sigma, 0, 1, |\sigma|) \not\models y = x$.

Thus, we obtain

$$(\sigma, 0, 0, |\sigma|) \not\models \forall u : (u = x \rightarrow \bigcirc(u = x)) \rightarrow (y = x \rightarrow \bigcirc(y = x))$$

□

Theorem 3.29 Let u be a variable which is not quantified in $p(u)$, and e be a state term which is admissible for u in $p(u)$. Then

$$EQadd \quad p(e) \supset \exists u : p(u)$$

Proof

Let σ be an interval and r an integer, $0 \leq r \preceq |\sigma|$. Suppose $(\sigma, 0, r, |\sigma|) \models p(e)$ and $(\sigma, 0, r, |\sigma|) \not\models \exists u : p(u)$. Then

$$\begin{array}{lll} & (\sigma, 0, r, |\sigma|) \models \neg\exists u : p(u) & \\ \Longleftrightarrow & (\sigma, 0, r, |\sigma|) \models \forall u : \neg p(u) & \text{Abb-all} \\ \Longrightarrow & (\sigma, 0, r, |\sigma|) \models \neg p(e) & \text{theorem 3.27} \end{array}$$

This is a contradiction.

□

Theorem 3.30 Let v be a variable and p and $q(v)$ be formulas such that v has no free occurrences in p. Then

$$UQadd \quad p \supset q(v) \implies p \supset \forall v : q(v)$$

Proof

Let σ be an interval and r an integer, $0 \leq r \preceq |\sigma|$. Suppose $p \supset q(v)$, and $(\sigma, 0, r, |\sigma|) \models p$. We need to prove that $(\sigma, 0, r, |\sigma|) \models \forall v : q(v)$.

Consider any σ', $\sigma' \overset{v}{=} \sigma$. Since v has no free occurrences in p, by Lemma 3.10, we have $(\sigma', 0, r, |\sigma'|) \models p$. Since $p \supset q(v)$, $(\sigma', 0, r, |\sigma'|) \models p \rightarrow q(v)$. Hence, $(\sigma', 0, r, |\sigma'|) \models q(v)$. Therefore

$$(\sigma, 0, r, |\sigma|) \models \forall v : q(v)$$

□

Theorem 3.31 Let v be a variable, and $p(v)$ be a formula.

1 $\models \Box(p(v) \rightarrow q) \rightarrow \Box(\exists v : p(v) \rightarrow q)$, if v has no free occurrences in q.
2 $\models \Box(p(v) \rightarrow q(v)) \rightarrow \Box(\forall v : p(v) \rightarrow \forall v : q(v))$

Proof

We prove only (2). Let σ be a model, and r an integer, $0 \leq r \preceq |\sigma|$.

$$\begin{array}{lll}
 & (\sigma, 0, r, |\sigma|) \models p(v) \rightarrow q(v) & \\
\Longleftrightarrow & (\sigma, 0, r, |\sigma|) \models \forall v : p(v) \rightarrow p(v) \text{ and } (\sigma, 0, r, |\sigma|) \models p(v) \rightarrow q(v) & \text{theorem 3.27} \\
\Longrightarrow & (\sigma, 0, r, |\sigma|) \models \forall v : p(v) \rightarrow q(v) & \text{theorem 3.3 (MP)} \\
\Longrightarrow & (\sigma, 0, r, |\sigma|) \models \forall v : p(v) \rightarrow \forall v : q(v) & \text{theorem 3.30}
\end{array}$$

□

The law ChV1 in Theorem 3.32 is useful for proving the substitution law given in Theorem 3.7. ChV1 is proved using only the rule I-exists.

Theorem 3.32 If variables x, y both are either static or dynamic, and neither does x appear in $p(y)$ nor does y appear in $p(x)$, then the following hold:

$$\begin{array}{ll}
ChV1 & \exists x : p(x) \equiv \exists y : p(y) \\
ChV2 & \forall x : p(x) \equiv \forall y : p(y) \\
AddQ1 & p(x) \equiv q(x) \implies \exists x : p(x) \equiv \exists x : q(x) \\
AddQ2 & p(x) \equiv q(x) \implies \forall x : p(x) \equiv \forall x : q(x)
\end{array}$$

Proof

We prove only ChV1 and AddQ1. Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of ChV1

$$\begin{array}{ll}
 & (\sigma, 0, k, |\sigma|) \models \exists x : p(x) \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models p(x) \text{ for some } \sigma' \overset{x}{=} \sigma
\end{array}$$

We can construct an interval σ'' by $|\sigma''| = |\sigma|$ and $(\sigma'', 0, k, |\sigma''|)[y] = (\sigma', 0, k, |\sigma'|)[x]$ and $(\sigma'', 0, k, |\sigma''|)[z] = (\sigma, 0, k, |\sigma|)[z]$ for all $z \in V - \{y\}$. Then, it is obvious that $(\sigma'', 0, k, |\sigma''|) \models p(y)$, and $\sigma'' \overset{y}{=} \sigma$. Hence, $(\sigma, 0, k, |\sigma|) \models \exists y : p(y)$.

The proof of AddQ1

$$\begin{array}{lll}
 & (\sigma, 0, k, |\sigma|) \models \exists x : p(x) & \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models p(x), \text{ for some } \sigma' \overset{x}{=} \sigma & \text{I-exists} \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models q(x), \text{ for some } \sigma' \overset{x}{=} \sigma & p(x) \equiv q(x) \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \exists x : q(x) & \text{I-exists}
\end{array}$$

□

Theorem 3.33 can be proved by Theorem 3.32. It is useful for substitution for a dynamic (or static) variable in a bound formula.

Theorem 3.33 If the variables x and y both are either dynamic or static, and y does not appear in $p(x)$, then

$$UQgo3 \quad \forall x : p(x) \supset p(y)$$

Proof

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

$$\begin{array}{lll}
 & (\sigma, 0, k, |\sigma|) \models \forall x : p(x) & \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models \forall y : p(y) & \text{ChV2} \\
\Longleftrightarrow & (\sigma', 0, k, |\sigma'|) \models p(y) \text{ for every } \sigma' \overset{y}{=} \sigma & \text{Abb-all} \\
\Longrightarrow & (\sigma, 0, k, |\sigma|) \models p(y) & \sigma \overset{y}{=} \sigma
\end{array}$$

□

In this chapter, we presented a collection of logic laws regarding variables, equality, and quantifications. They are useful for theorem proving and reduction of programs. However, we have not worked out a deductive system for EITL as such a system is beyond the scope of this thesis.

Chapter 4

Programming Language

Summary: An extended Tempura language, which is a basis for us to develop a framing technique for temporal logic programming, is formalized; the syntax and semantics of the extended Tempura are presented; a normal form of programs is described and proved.

Tempura language was introduced in [61] as an executable subset of ITL. The principal restriction is that Tempura programs must be deterministic and the intervals over which the programs are executed must be finite. Therefore, the following constructs are not allowed in basic Tempura.

1. Disjunction.

   The disjunction is non-deterministic, so, in general, it is not permitted in Tempura. For instance,

   $$x := 1 \vee y := 2$$

   is not a well-formed program in Tempura.

2. Negation.

   Basically, the negation is non-deterministic, for example,

   $$\neg skip \equiv empty \ \vee \ len(2) \vee len(3) \vee \ldots$$

   $$\neg(x := 2 \wedge y := 3) \equiv \neg(x := 2) \vee \neg(y := 3)$$

   Therefore, the negation of a program is not syntactically permitted. However, negation is indispensable for boolean expressions which may appear in conditional and iterative statements. For instance, $\neg(x = 2)$ and $\neg(x > 3 \wedge y < 5)$ are well-formed boolean expressions.

3. State formulas.

   A state formula such as $x = 2$ is not a program in Tempura because no definite interval has been specified.

4. Universal quantification.

   It is impossible to execute a program with universal quantification but a restricted form of universal quantification can be defined (see [61]).

Tempura will be extended in several respects in this thesis:

1. Infinite intervals.

   The reactive systems, such as operating systems and control software of safety critical systems, etc. require non-terminating executions of software. To make Tempura more useful in practice, we extend Tempura to permit a program to be executed over an infinite interval.

2. Previous operator.

   A variable can refer to its previous value but we do not permit the previous operator to appear freely in programs because this is very expensive in practice.

3. Projection.

   A construct $(p_1, ..., p_m)\ prj\ q$, called projection, is used in the extended language and will be discussed in Chapter 5.

4. Framing.

   A framing technique is introduced and discussed in Chapters 6,7,8 in detail.

5. Await.

   After the framing technique is introduced, a communication and synchronization operator, *await*, is defined. It is discussed in Chapter 8.

This chapter is organized as follows: Section 4.1 presents the syntax of the extended Tempura, derived structs and data structures. Section 4.2 presents the semantics of the extended Tempura. Section 4.3 defines P-models corresponding to a program P. In Section 4.4, the normal form of programs is formalised and proved.

4.1 Syntax

This section is devoted to presenting the syntax of the extended Tempura language. First, the grammar of statements is given, then expressions are defined.

4.1.1 Programs

The programming language we use is a subset of the underlying logic. As mentioned earlier, we augment Tempura with framing, parallel, projection and await operators [20, 22]. In addition, the variables within a program can refer to their previous values.

A program can be terminable or non-terminable. A program is terminable if it has a finite model. A program is non-terminable if it has no finite model. If a program is terminable (non-terminable) then it can be executed over a finite (infinite) interval. The language provides some statements to specify the interval over which a program is executed. In fact, the primitive statement *empty* and the derived statements $len(k)$ and $halt(c)$ are used for this purpose.

The extended language also intends to be as close to a deterministic language as possible. Being deterministic means that a program has only one model from the point of view of semantics; whereas, from the point of view of syntax, being deterministic requires that, roughly speaking, the disjunction be generally unavailable. The negation of a temporal formula, being fundamentally non-deterministic, is not a primitive operator of the language. Instead, the conditional statement (see below) and *empty*, both defined in terms of the negation, are taken as primitives. This implies that some of the derived operators of ITL cannot be defined at all in Tempura. For instance, universal quantification and an arbitrary choice cannot be defined in full generality although a restricted form of the universal quantification can be defined and a mutually exclusive disjunction can be used in a program. Moreover, since the execution of a program proceeds over a series of states, a deterministic program also requires that only one choice, *more* or *empty*, be derived at each state so that the execution of the program can determine whether to continue or terminate.

The previous operator facilitates referring to the previous values of variables which is useful in a framed environment. But using the previous operator is very expensive in practice because extra memory and computation time are required. Therefore, the previous operator is permitted only within expressions.

In this chapter, it is sufficient to introduce the basic constructs of Tempura. Later, once the framing technique has been formalized, an await statement is defined in Chapter 8. A projection construct is discussed in Chapter 5. At the end of the thesis, as a comparison, a more powerful framed concurrent temporal logic programming language (FTLL) is briefly presented in Chapter 8.

Programs are constructed inductively from the operators shown below, together with a suitable choice of expressions which may involve the previous operator.

There are ten elementary statements, six of them are basic constructs directly taken from EITL, i.e. the equality (=), conjunction ($\wedge$), always ($\Box$) (since $\Diamond$ is not permitted), existential quantification ($\exists x : p$), next ($\bigcirc$) and chop (;); the other four, conditional statement, while statement, parallel and *empty* are derived constructs in EITL. From the point of view of the programming language, the ten statements are all primitives since the negation ($\neg$), sometimes ($\Diamond$), disjunction ($\vee$) and chop-plus (+) are absent in the definition of the extended Tempura. This also implies that some operators cannot be derived in the way in which they were derived before. Note that in an induction proof of a property of programs, the assignment and *empty* statements can be thought of as basic statements and the others can be treated as composite statements. As usual, in the following, x denotes a variable, e stands for an arbitrary arithmetic expression, b stands for a boolean expression, and p and q stand for programs.

$S-ass$	**Assignment (Unification) :**	$x = e$
$S-and$	**Conjunction statement :**	$p \wedge q$
$S-if$	**Conditional statement :**	$if\ b\ then\ p\ else\ q \overset{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
$S-loc$	**Existential quantification :**	$\exists x : p$
$S-next$	**Next statement :**	$\bigcirc p$
$S-alw$	**Always statement :**	$\Box p$
$S-seq$	**Sequential statement :**	$p; q$

$S-while$	**While statement :**	$while\ b\ do\ p \overset{\text{def}}{=} (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$
$S-par$	**Parallel statement :**	$p\|q \overset{\text{def}}{=} p \wedge (q; true) \vee q \wedge (p; true)$
$S-end$	**Termination :**	$empty$

where b is a boolean state expression consisting of propositions, variables, and boolean connectives.

The meaning of these statements can be captured by the interpretations within the logic.

The assignment $x = e$ is a special case of equality and means that the value of the variable x is equal to the value of the expression e. Since it is an equality, its interpretation is subject to I-equal. Whenever an assignment $x = e$ is encountered, we evaluate x and e by $\mathcal{I}[x]$ and $\mathcal{I}[e]$ to see whether or not $\mathcal{I}[x] = \mathcal{I}[e]$. Therefore, if e is evaluated to a constant in D and x has not been specified (or has been specified as the same value as e) before, then we say e is assigned to x. In this case, the equality $x = e$ is satisfied otherwise it is false. It is really that x is unified with e as in Prolog.

Note that the equality in Tempura has two functions: assignment and comparison. The former is a statement in a program while the latter is in a condition, boolean expression, associated with the conditional statements or iterative statements. An assignment is true as long as it is satisfiable whereas a condition is true if all the variables w.r.t. the condition are specified and the condition is evaluated to true.

The conditional statement *if b then p else q*, as in the conventional programming language, means that if the condition b is evaluated to true then the process (i.e. sub-program) p is executed otherwise the process q is executed.

As in the logic, the next statement $\bigcirc p$ means that p holds at the next state while $\Box p$ means that p holds in all states from now. The terminal statement *empty* simply means that the current state has reached the final state of the interval over which a program is executed.

The sequential statement $p; q$ means that p holds from now until some point in time in the future and from that time point q holds. Intuitively, the program p is executed from the current state until its termination, then the program q is executed.

The conjunction statement $p \wedge q$ is executed in a parallel manner. The processes p and q start at the same state but may terminate at different states. They share all the states and variables during the mutual execution.

The statement *while b do p* allows process p to be repeatedly executed a finite (or an infinite) number of times over a finite (or an infinite) interval as long as condition b holds at the beginning of each execution. If condition b becomes false, then the *while* statement terminates, otherwise, p is executed. For instance, *while true do p* allows p over an infinite interval to be executed a finite or an infinite number of times, each time on a finite subinterval, or an infinite subinterval for the last execution. However, this statement is obviously *false* within any finite interval if the execution of p requires a non-singleton interval. Another extreme case is the statement *while b do empty*. One can show that it is simply equivalent to $\neg b \wedge empty$ (see Theorem 4.7).

The parallel computation presented here proceeds synchronously, and may be modelled by true concurrency. It is weaker than the asynchronous parallel computation modelled by interleaving. In fact, the parallel operator presented here is very close to the conjunction. The basic difference between $p\|q$ and $p \wedge q$ is that the former allows both processes p and q to be able to specify their own intervals while the latter does not. For instance, $len(2)||len(3)$ holds but $len(2) \land len(3)$ is obviously false.

The existential quantification statement $\exists x : p$ intends to hide the variable x within the process p. It may permit a process p to use a local variable x. This idea can be realized in an operational semantics. However, within temporal semantics, the concept of a local variable is not effective (see Chapter 7 for details).

Although the language given here rules out the disjunction and the negation as well as the universal quantification constructs as basic statements, the language does not guarantee that programs built from it are deterministic. The problem is that an immediate assignment, say $x = e$, is non-deterministic since $x = e \equiv x = e \land empty \lor x = e \land more$, so are $\bigcirc(x = e)$, $\Box(x = e)$ etc. If a unit assignment $x := e$ rather than an immediate assignment were used as a primitive statement, the language would be deterministic. However, an immediate assignment has its advantages, e.g. it can easily be used to initialise variables and it corresponds to the equality taken directly from the underlying logic. So, it is necessary to keep the immediate assignment as a primitive. Therefore, to build a deterministic program in the language, $len(k)$ ($k \geq 0$) or $halt(b)$ (b is a boolean expression) (see below) or $\Box more$ (only for a non-terminably deterministic program) construct must be provided to specify a definite interval for the program.

Hereafter, we will use the term 'programs' to mean programs belonging to the extended Tempura. Most of them are deterministic and terminable programs which may involve the previous operator only in expressions. However, some results are discussed in a broader scope in which non-deterministic programs and/or infinite intervals are considered. To avoid ambiguity, whenever only a deterministic program or only an finite interval is involved, we clarify it in an explicit manner.

Example 4.1 Here is a simple program: $x = 0 \land while\ (x < 3)\ do\ (\bigcirc(x = \ominus x + 1) \land \bigcirc empty)$.

$\Box$

4.1.2 Derived Constructs

The following constructs can be built from the basic statements and used in programs.

1. **Termination and the final state**

   The formula $halt(p)$ is true over an interval if and only if p is true at the final state. In other words, the computation terminates as soon as the formula p becomes true. The other relevant operators are fin and $keep$. The formula $fin(p)$ is true as long as p is true at the final state while $keep(p)$ is true if p is true at every state ignoring the final one.

$$\begin{array}{lll} S-halt & halt(p) & \overset{\text{def}}{=} \Box(empty \leftrightarrow p) \\ S-fin & fin(p) & \overset{\text{def}}{=} \Box(empty \rightarrow p) \\ S-keep & keep(p) & \overset{\text{def}}{=} \Box(\neg empty \rightarrow p) \end{array}$$

2. **Assignment Operators**

   Based on the equality, several assignment operators can be defined. In the following, let x be a variable, u a static variable, and e an expression.

1) Temporal assignment

$$S - tem - ass \quad x \leftarrow e \overset{\text{def}}{=} \exists u : u = e \wedge \Box(empty \rightarrow x = u)$$

Such an assignment has the meaning: the value of x at the end of an interval equals the value of e at the current state.

2) Next assignment

$$S - next - ass \quad x \circ\!= e \overset{\text{def}}{=} \bigcirc x = e$$

This assignment has the meaning: the value of x at the next state equals the value of e at the current state. Note that the next assignment does not specify the length of the interval although it takes one unit of time.

3) Unit assignment

$$S - unit - ass \quad x \ := \ e \overset{\text{def}}{=} skip \wedge x \ \circ = e$$

The unit assignment asserts that the value of x at the next state equals the value of e at the current state the same as the next assignment does and it specifies the length of the interval is one unit of time.

4) Multiple assignments

$$S - mult - ass \quad (x_1, ..., x_n) \ op \ (e_1, ..., e_2) \ \overset{\text{def}}{=} \ x_1 \ op \ e_1 \ \wedge ... \wedge \ x_n \ op \ e_n$$

where $op ::= \circ = \mid := \mid \leftarrow \mid =$

3. Iterative Statements

1) For-loops

A particularly simple form of loop is *for n times do p* which just denotes n iteration of p, i.e. $p; ...; p$ (n *times*). It is inductively defined as follows:

$$\begin{array}{lcl} for\ 0\ times\ do\ p & \overset{\text{def}}{=} & empty \\ for\ n+1\ times\ do\ p & \overset{\text{def}}{=} & (for\ n\ times\ do\ p); p \end{array}$$

where $n \in N_0$. Loops with control variables can also be defined.

$$\begin{array}{lcl} for\ i < 0\ do\ p & \overset{\text{def}}{=} & empty \\ for\ i < n+1\ do\ p & \overset{\text{def}}{=} & (for\ i < n\ do\ p); p[n/i] \end{array}$$

where $n \in N_0$ and i is a static variable, and n is substitutable for i in p as defined in Chapter 3.

2) Repeat-loops

Finally, a repeat-loop is defined in terms of the while-loop in the usual way.

$$repeat\ p\ until\ b \overset{\text{def}}{=} p; while\ \neg b\ do\ p$$

Other loops could be defined in a similar way. For example, in the loop *for* $v \in l$ *do* p the control variable v takes successive value from the list l.

4.1.3 Expressions

The choice of permissible expressions is wider but only constants, variables, array elements, strings, and list expressions, as well as restricted temporal expressions are used in this thesis.

1. Constants.

   Constants include integers and boolean constants.

2. Variables.

   In the underlying programming language, variables are, as in EITL, partitioned into two parts: static variables and dynamic variables. From the point of view of data structure, variables are divided into simple variables (say x) and structured variables (say $x[1]$). The values of a variable at the previous and next states can be referred to over an interval.

3. Arithmetic expressions.

   Arithmetic operators are: +, -, *, /, *mod*. Arithmetic expressions are built from integers, variables and arithmetic operators.

   - An integer is an arithmetic expression.
   - If e_1 and e_2 are expressions, so are the following operations:

   $$e_1 + e_2,\ e_1 - e_2,\ e_1 * e_2,\ e_1/e_2,\ e_1\ mod\ e_2$$

   - If e is an expressions, so is the $\bigcirc e$ at a state different from the final one.
   - If e is an expressions, so is the $\circleddash e$ at a state different from the first one.

4. Boolean expressions.

   Boolean expressions include relational expressions. Relational expressions are built from arithmetic expressions and relational operators: $>, <, =, \leq, \neq$ etc. Boolean expressions are built from relational expressions and boolean connectives such as $\wedge, \vee, \neg, \rightarrow$.

   - If e_1 and e_2 are arithmetic expressions, then the following constructs are relational expressions:

   $$e_1 > e_2,\ e_1 \geq e_2,\ e_1 < e_2,\ e_1 \leq e_2,\ e_1 = e_2\ e_1 \neq e_2$$

   - A relational expression is a boolean expression.
   - Boolean constants 'true' and 'false' are boolean expressions.
   - Temporal constructs *more* and *empty* are boolean expressions.
   - If b_1 and b_2 are boolean expressions, so are the following constructs:

   $$\neg b_1,\ b_1 \wedge b_2,\ b_1 \vee b_2,\ b_1 \rightarrow b_2,\ b_1 \leftrightarrow b_2$$

4.1.4 Data Structures

In addition to constants and simple variables, we also use *lists*, *strings*, and *arrays*.

1. **Lists.**

   Lists are sequences of elements separated by commas and enclosed in angle brackets, such as $< 2,4,6,8 >$. The length of a list l, denoted by $|l|$, is the number of the elements in l minus 1; the i^{th} element is denoted by $l[i]$, the sublist from element i to element j by $l(i..j)$. The smallest subscript of a list is 0, and the empty list is denoted by ϵ. To manipulate lists, we use the following operators: concatenation $(\cdot)$, fusion $(\circ)$, head (hd), tail (tl) and last (lt). Let l be a non-empty list. $hd(l)$ obtains the first element of l, $tl(l)$ obtains the sublist $l(1..|l|)$, i.e. the list l without its first element; $lt(l)$ obtains the last element of l; the concatenation of two lists l_1 and l_2, denoted by $l_1 \cdot l_2$, is the list whose elements are the elements of l_1 followed by the elements of l_2; the fusion of two lists l_1 and l_2, denoted by $l_1 \circ l_2$, is the list $l_1 \cdot l_2(1..|l_2|)$, we assume that the last element of l_1 and the first element of l_2 are the same and overlap in the resulting list. An empty list ϵ concatenated or fused with any list l gives l. Formally, we introduce the following definitions.

   Let $l =< s_0, s_1, ... >$ be a list, we define

   1) Length of a list

   $$|l| = \begin{cases} n & \text{if } l =< s_0, ..., s_n > \\ \omega & l \text{ is infinite} \end{cases}$$

   $|\epsilon| = -1$. If $|l| = 0$, then l is called a singleton list.

   2) Head of a list
      - $hd(l) =< s_0 >$
      - $hd(\epsilon) = \epsilon$

   3) Tail of a list
      - $tl(l) =< s_1, ... >$
      - $tl(\epsilon) = \epsilon$

   4) Last element of a list
      - $lt(l) =< s_{|l|} >$ if $|l| < \omega$.
      - $lt(\epsilon) = \epsilon$

      If $|l| = \omega$ then $lt(l)$ is undefined.

   5) Fusion $(\circ)$

   $$l_1 \circ l_2 = \begin{cases} l_1 & \text{if } |l_1| = \omega \text{ or } l_2 = \epsilon \\ l_2 & \text{if } l_1 = \epsilon \\ < s_0, ..., s_i ... > & \text{if } l_1 =< s_0, ..., s_i > \text{ and } l_2 =< s_i, ... > \text{ and } i \in N_0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

   6) Concatenation $(\cdot)$

   $$l_1 \cdot l_2 = \begin{cases} l_1 & \text{if } |l_1| = \omega \text{ or } l_2 = \epsilon \\ l_2 & \text{if } l_1 = \epsilon \\ < s_0, ..., s_i, s_{i+1}, ... > & \text{if } l_1 =< s_0, ..., s_i > \text{ and } l_2 =< s_{i+1}, ... > \\ & \text{and } i \in N_0 \end{cases}$$

2. Strings.

   Strings are surrounded by the double quotation marks, like "string". They may be indexed and manipulated in the same way as lists.

3. **Arrays.**

Arrays are sets of indexed elements of the same type. Arrays can be of one, two or three dimensions. An array can be expressed in terms of lists with row-first in an implementation.

4.1.5 Omitting Parentheses Precedence Rules

In order to avoid an excessive number of parentheses, the following precedence rules are used:

1	$\neg$
2	$\bigcirc, \odot, \Diamond, \Box, \ominus, \textcircled{$\underline{\cdot}$}$
3	$=, \circ\!\!=, :=, \leftarrow$
4	$\land, \lor$
5	$\rightarrow, \leftrightarrow$
6	;

where 1=highest and 6=lowest.

There is an important fact we need to point out here. Since an expression refers neither to the previous value at the beginning state nor to the next value at the ending state over an interval, and the past operators, in general, are forbidden in a program, the statements of the extended Tempura language are lec-formulas in the sense of the logic language. However, during the reduction of a program, a sub-part of the program can refer to a previous state as long as the previous state does not go beyond the first state of the interval.

4.2 Semantics of Programs

An expression e can be treated as a term and a program P can be viewed as a formula in EITL. Therefore, the evaluation of e and the interpretation of P can be done as in EITL. However, since the programming language is a subset of the underlying logic, a program may have its own characteristics and may be interpreted in a simple and manageable way. In particular, when a framing technique is introduced, some easily manageable models are required. We will use a minimal model (see Chapter 7) which is based on canonical models given below.

In order to interpret framed temporal logic programs, we assume that a program P contains a finite set S of variables and a finite set Φ of propositions. We interpret propositions over B and variables over D'. For a program P, there are three ways to interpret propositions contained in P, namely canonical, complete, and partial interpretations as defined for the semantics of logic programming language [13]. Here, we use the canonical interpretation only on propositions. That is, in a model $\sigma =< (I_v^0, I_p^0), ... >$, I_v^k is used as in the logic but I_p^k is changed to the canonical interpretation.

A canonical interpretation on propositions is a subset $I_p \subseteq \Phi$. Implicitly, propositions not in I_p are false. Note that I_p^k in the interpretation of the logic framework is an assignment of a truth value in B to each proposition $p \in Prop$ at state s_k; whereas in a canonical interpretation, I_p^k is a set of propositions, each of them has truth value true in B at s_k. Clearly, the two definitions are equivalent except that they refer to different sets of variables and propositions. Using canonical interpretation is useful for easy manipulation of minimal models. Let $\sigma =< (I_v^0, I_p^0), ... >$ be a model. We denote the sequence of interpretation on propositions of σ by $\sigma_p =< I_p^0, ... >$. σ_p is said to be canonical if each $I_p^i (i \geq 0)$ is a canonical interpretation on propositions.

If there exists a model σ with σ_p being a canonical interpretation sequence on propositions and $\sigma \models P$ as in the logic, then program P is said to be satisfiable under the canonical interpretation on propositions, denoted by $\sigma \models_c P$; and σ_p is said to be a canonical interpretation sequence (on propositions) of program P. If for all σ with σ_p being a canonical interpretation sequence, $\sigma \models P$, then program P is said to be valid under the canonical interpretation on propositions, denoted by $\models_c P$.

Note that the definition of the canonical interpretation of program P is independent of its syntax in the sense that the definition does not refer to the structure of the program. So the definition can be extended so that it can be applied to non-deterministic programs and temporal formulas.

Example 4.2 For the propositional formula, P_1: $\neg A \leftrightarrow \bigcirc B$, which can be treated as a non-deterministic program, we have $\Phi = \{A, B\}$, and P_1 has the following canonical interpretation sequences of length 2, $< \phi, \{B\} >$, $< \phi, \{A, B\} >$, $< \{B\}, \{B\} >$, $< \{B\}, \{A, B\} >$, $< \{A\}, \phi >$, $< \{A\}, \{A\} >$, $< \{A, B\}, \phi >$, and $< \{A, B\}, \{A\} >$.

P_1 is satisfiable but not valid under the canonical interpretation on propositions because a canonical interpretation sequence, $< \phi, \phi >$, does not satisfy it. □

Note that a program P can be satisfied by several different canonical models on propositions so program P has, possibly, different meanings under different models. Therefore, it is important to choose a model which satisfies the intended meaning of a program P, and this is the topic of Chapter 7.

Since the canonical model is basically equivalent to the basic model except that the latter acts on the fixed set V of variables and the fixed set of propositions, whereas the former acts on the set of variables and the set of propositions within a concrete program. Hence, in what follows, we do not distinguish between the relation $\models$ and $\models_c$. Whenever I_p is used to interpret a program P, it is treated as a subset of propositions Φ_p while whenever I_p is used to interpret a formula q, it is treated as a function (or set) over *Prop*.

4.3 Models Corresponding to Programs

A model of program P is a particular sequence of states that obey certain constraints imposed by P. Each model of program P is a finite (or infinite) sequence of states over S and Φ, the set of variables and the set of propositions associated with P, respectively. Let V be the set of variables and *Prop* the set of propositions specified as in Chapter 3. V contains S and *Prop* contains Φ. Consider a model

$$\sigma =< s_0, \ldots >$$

over V and *Prop*. We say that model σ is a P-model [66] if there is a model of P over S and Φ,

$$\sigma' =< s'_0, \ldots >$$

such that $|\sigma| = |\sigma'|$ and each state s_j is identical to s'_j, when restricted to the sets S and Φ; that is, for all $0 \leq j \preceq |\sigma|$, $s_j[p] = s'_j[p]$, for all $p \in \Phi$, and $s_j[x] = s'_j[x]$ for all $x \in S$.

Note that while a model of P interprets only the variables in S and propositions in Φ, a P-model may give interpretation to additional variables and propositions.

Definition 4.1 Let p be a temporal formula, V_p the set of the variables appearing in p, and Φ_p the set of the propositions appearing in p. Let $M(V, Prop, P)$ be the set of all P-models over V and $Prop$, where $V = V_p \cup S$ and $Prop = \Phi_p \cup \Phi$. p is satisfiable over P, denoted by $\sigma \models_{M(V,Prop,P)} p$, if there exists a model $\sigma \in M(V, Prop, P)$ and $\sigma \models p$. p is valid (also described as being P-valid) over program P, denoted by $\models_{M(V,Prop,P)} p$, if for every model $\sigma \in M(V, Prop, P)$, $\sigma \models p$.

□

It is not difficult to see that the above definition implies that every model in $M(V, Prop, P)$ satisfies p for all $V, Prop$ that contain $V_p \cup S$ and $\Phi_p \cup \Phi$, respectively.

Clearly, every valid formula is also a P-valid formula, but since $M(V, Prop, P)$ is only a subset of the universe of all possible models, there are formulas that are P-valid but not valid in general. Consider, for example, P given by $x = 0 \wedge \Box(more \rightarrow \bigcirc x = x + 1) \wedge len(2)$. Clearly, the formula $x = 0$ is P-valid, since it holds in the first state (i.e. s_0) of every model of P. On the other hand, $x = 0$ is obviously not a valid formula, as there are many models whose first state does not satisfy $x = 0$.

In the sequel, we need to employ, besides the canonical model, also the minimal model (see Chapter 7). To distinguish between them, we use $M_m(V, Prop, P)$ to denote the set of P-models in which program P is interpreted by the minimal model. If $V = S$ and $Prop = \Phi$, we refer to $M(V, Prop, P)$ and $M_m(V, Prop, P)$ simply as $M(P)$, and $M_m(P)$, respectively.

We capture the relationship between the P-models and models of the program P in Theorem 4.1.

Theorem 4.1 Let P be a program, and σ a P-model. If $\sigma \in M(P)$, then $\sigma \models P$.

Proof

Let S be the set of variables and Φ be the set of propositions contained in program P. Let V be the set of variables and $Prop$ be the set of propositions associated with the P-model σ. Since $\sigma \in M(P)$, by the definition of P-model, there exists a σ' such that $\sigma' \models P$ and $|\sigma| = |\sigma'|$, and for all $0 \leq j \preceq |\sigma|$, $s_j[p] = s'_j[p]$ for all $p \in \Phi$, and $s_j[x] = s'_j[x]$ for all $x \in S$; by the definition of notation $M(P)$, $V = S$ and $Prop = \Phi$. Hence, $\sigma = \sigma' \models P$.

□

4.4 Normal Form of Programs

In this section, we define a kind of normal form for programs and prove any program in the extended Tempura can be reduced to the normal form. We first prove Lemma 4.2, and then Theorems 4.3, 4.4, 4.5. They are useful for the reduction of the *while* statement. After that, we prove some conclusions regarding the *while* statement. Finally, we prove that a program built in extended Tempura can be reduced to the normal form.

Lemma 4.2 Let p be a formula.

$EMP1$	$empty; empty \equiv empty$
$EMP2$	$p \wedge empty; empty \equiv p \wedge empty$
$EMP3$	$empty \vee (p; empty) \equiv empty \vee (p \wedge more; empty)$
$EMP4$	$p \wedge empty; q \wedge empty \equiv p \wedge q \wedge empty$ if q is a lec-formula
$EMP5$	$p^+ \wedge empty \equiv p \wedge empty$

Proof

The proofs of EMP1 and EMP5 are straightforward.

The proof of EMP2

$$\begin{array}{rcll} p \wedge empty; empty & \equiv & p \wedge empty \wedge \Diamond empty & \text{FEP6} \\ & \equiv & p \wedge (empty; empty) & \text{FEP6} \\ & \equiv & p \wedge empty & \text{EMP1} \end{array}$$

The proof of EMP3

$$\begin{array}{rcll} empty \vee (p; empty) & \equiv & empty \vee ((p \wedge empty \vee p \wedge more); empty) & \text{tautology} \\ & \equiv & empty \vee (p \wedge empty; empty) \vee (p \wedge more; empty) & \text{FD9} \\ & \equiv & empty \vee p \wedge empty \vee (p \wedge more; empty) & \text{EMP2} \\ & \equiv & empty \vee (p \wedge more; empty) & \text{tautology} \end{array}$$

The proof of EMP4

Let σ be a model and k an integer, $0 \leq k \preceq |\sigma|$.

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models p \wedge empty; q \wedge empty \\ \Longleftrightarrow & (\sigma, 0, k, r) \models p \wedge empty \text{ and } (\sigma, r, r, |\sigma|) \models q \wedge empty \text{ for some } r, k \leq r \preceq |\sigma| \\ \Longleftrightarrow & (\sigma, 0, k, r) \models p \text{ and } r = k \text{ and } (\sigma, r, r, |\sigma|) \models q \text{ and } r = |\sigma| \\ \Longleftrightarrow & (\sigma, 0, k, k) \models p \text{ and } (\sigma, k, k, k) \models q \text{ and } k = |\sigma| \\ \Longleftrightarrow & (\sigma, 0, k, k) \models p \text{ and } (\sigma, 0, k, k) \models q \text{ and } k = |\sigma| \;\; q \text{ is a lec-formula} \\ \Longleftrightarrow & (\sigma, 0, k, k) \models p \wedge q \text{ and } k = |\sigma| \\ \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models p \wedge q \wedge empty \end{array}$$

□

Theorem 4.3 Let p be a lec-formula. Then

$$FPS7 \quad p^+ \equiv p \vee (p \wedge more; p^+)$$

Proof

$p \vee (p \wedge more; p^+) \supset p^+$ is obvious. We prove $p^+ \supset p \vee (p \wedge more; p^+)$ only. Let σ be an interval and k an integer, $0 \leq k \leq |\sigma|$.

Suppose $(\sigma, 0, k, |\sigma|) \models p^+$. Thus, either there are finitely many $r_0, ..., r_n \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \preceq r_n = |\sigma|$ and $(\sigma, 0, r_0, r_1) \models p$ and for all $1 < l \leq n$ $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$; or $|\sigma| = \omega$ and there are infinitely many integers $k = r_0 \leq r_1, ...$ such that $\lim_{r_i \to \infty} r_i = \omega$, and $(\sigma, 0, r_0, r_1) \models p$ and for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

In the former case, if $n = 1$, then $(\sigma, 0, k, |\sigma|) \models p^+$ amounts to $(\sigma, 0, r_0, r_1) \models p$, i.e. $(\sigma, 0, k, |\sigma|) \models p$; if $n \geq 2$ with $k = |\sigma|$ then $(\sigma, 0, k, |\sigma|) \models p^+$ is equivalent to $(\sigma, 0, r_0, r_1) \models p \wedge empty$ and $r_0 = k = r_1 = |\sigma|$ (EMP5) leading to $(\sigma, 0, k, |\sigma|) \models p$. If $n \geq 2$ with $k < |\sigma|$, there is the least integer h such that $r_0 = r_1 = ... = r_{h-1} < r_h \leq ... \leq r_n$. If $h = 1$, then $(\sigma, 0, k, r_1) \models more \wedge p$. That is, $(\sigma, 0, k, r_1) \models p \wedge more$ and $(\sigma, r_1, r_1, |\sigma|) \models p^+$. We have $(\sigma, 0, k, |\sigma|) \models p \wedge more; p^+$. If $h > 1$, then $(\sigma, 0, r_0, r_1) \models p \wedge empty$ and for all $1 < l < h$ $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p \wedge empty$ and $(\sigma, r_{h-1}, r_{h-1}, r_h) \models p \wedge more$. That is, $(\sigma, 0, k, r_h) \models p \wedge empty; ...; p \wedge empty; p \wedge more$ and $(\sigma, r_h, r_h, |\sigma|) \models p^+$. By EMP4 and FEP4, we obtain $(\sigma, 0, k, |\sigma|) \models p \wedge more; p^+$.

In the latter case, $(\sigma,0,k,|\sigma|) \models p^+$ amounts to $(\sigma,0,r_0,r_1) \models p$ and for all $l > 1$ $(\sigma,r_{l-1},r_{l-1},r_l) \models p$. Since $\lim_{i\to\infty} r_i = \omega$, there exists the least integer h such that $r_{h-1} < r_h$. Thus, a similar argument as in the former case can be given to prove $(\sigma,0,k,|\sigma|) \models p \wedge more; p^+$.

□

Corollary 4.4 Let p be lec-formula. Then

$$\begin{array}{ll} 1 & p;p^+ \supset p \vee (p \wedge more; p^+) \\ 2 & p \wedge empty; p^+ \supset p \vee (p \wedge more; p^+) \\ 3 & p^+ \wedge more \equiv p \wedge more \vee (p \wedge more; p^+) \end{array}$$

Proof Straightforward.

□

Theorem 4.5 Let p be a lec-formula. Then

$$FST9 \quad p^* \equiv empty \vee (p \wedge more; p^*) \vee p \wedge \Box more$$

Proof

Let p be a lec-formula. Then

$$\begin{array}{lll} p^* & \equiv empty \vee p^+ & \text{Abb-star} \\ & \equiv empty \vee (p \vee (p \wedge more; p^+)) & \text{theorem 4.3} \\ & \equiv empty \vee (p; empty) \vee p \wedge \Box more \vee (p \wedge more; p^+) & \text{TER} \\ & \equiv empty \vee ((p \wedge empty \vee p \wedge more); empty) \vee (p \wedge more; p^+) \vee p \wedge \Box more & \text{tautology} \\ & \equiv empty \vee (p \wedge empty; empty) \vee (p \wedge more; empty) \vee (p \wedge more; p^+) \vee p \wedge \Box more & \text{FD9} \\ & \equiv empty \vee p \wedge empty \vee (p \wedge more; empty) \vee (p \wedge more; p^+) \vee p \wedge \Box more & \text{EMP2} \\ & \equiv empty \vee (p \wedge more; empty) \vee (p \wedge more; p^+) \vee p \wedge \Box more & \text{tautology} \\ & \equiv empty \vee (p \wedge more; (empty \vee p^+) \vee p \wedge \Box more & \text{FD9} \\ & \equiv empty \vee (p \wedge more; p^*) \vee p \wedge \Box more & \text{Abb-star} \end{array}$$

□

We now prove some theorems regarding the *while* statement. The *while* statement can be expressed in a more operational way by means of an inductive description which is equivalent to its original definition. We show this in Theorem 4.7. Before proving it, we first prove lemma 4.6.

Lemma 4.6 If s is a state formula, then $fin(s) \wedge (p;q) \equiv (p; fin(s) \wedge q)$

Proof

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$. Suppose $(\sigma,0,k,|\sigma|) \models fin(s) \wedge (p;q)$. Then $(\sigma,0,k,|\sigma|) \models fin(s)$ and $(\sigma,0,k,|\sigma|) \models p;q$. The latter amounts to $(\sigma,0,k,r) \models p$ and $(\sigma,r,r,|\sigma|) \models q$ for some $r, k \leq r \preceq |\sigma|$, whereas the former means $(\sigma,0,k,|\sigma|) \models \Box(empty \rightarrow s)$. That is, $(\sigma,0,h,|\sigma|) \models empty \rightarrow s$ for every $h, k \leq h \preceq |\sigma|$. This implies that $(\sigma,0,k_1,|\sigma|) \models empty \rightarrow s$ for every $k_1, r \leq k_1 \preceq |\sigma|$. Moreover, since s is a state formula, $empty \rightarrow s$ is an lec-formula. $(\sigma,0,k_1,|\sigma|) \models empty \rightarrow s$ iff $(\sigma,k_1,k_1,|\sigma|) \models empty \rightarrow s$ for every $k_1, r \leq k_1 \preceq |\sigma|$. Hence, $(\sigma,0,k,r) \models p$ and $(\sigma,r,r,|\sigma|) \models \Box(empty \rightarrow s) \wedge q$. It follows that $(\sigma,0,k,|\sigma|) \models p; fin(s) \wedge q$.

Conversely, suppose $(\sigma, 0, k, |\sigma|) \models p; fin(s) \wedge q$. Then, $(\sigma, 0, k, r) \models p$ and $(\sigma, r, r, |\sigma|) \models fin(s)$ and $(\sigma, r, r, |\sigma|) \models q$ for some $r, k \leq r \preceq |\sigma|$. It turns out that $(\sigma, 0, k, |\sigma|) \models p; q$ and $(\sigma, r, h, |\sigma|) \models$ $empty \rightarrow s$ for every $h, r \leq h \preceq |\sigma|$. Since $empty \rightarrow s$ is an lec-formula, $(\sigma, r, h, |\sigma|) \models empty \rightarrow s$ iff $(\sigma, 0, h, |\sigma|) \models empty \rightarrow s$ for every $h, r \leq h \preceq |\sigma|$. It is easy to check that $(\sigma, 0, h, |\sigma|) \models empty \rightarrow$ s for every $h, 0 \leq h < r$ because the formula is vacuously true in this case. Thus, $(\sigma, 0, k, |\sigma|) \models empty$ $\rightarrow s$ for every $k, 0 \leq k \preceq |\sigma|$. That is, $\sigma \models \Box(empty \rightarrow s)$. Therefore, $\sigma \models fin(s) \wedge (p; q)$.

$\Box$

Theorem 4.7

$$\begin{array}{lll}
(1) \;\; (while\ b\ do\ p) & \equiv & ((\neg b \wedge empty) \vee (b \wedge p; while\ b\ do\ p)) \\
 & & \vee b \wedge p \wedge \Box more \\
(2) \;\; (while\ b\ do\ p) & \equiv & ((\neg b \wedge empty) \vee (b \wedge p \wedge more; while\ b\ do\ p)) \\
 & & \vee b \wedge p \wedge \Box more \\
 & & \text{if } p \text{ is a } lec\text{-formula.}
\end{array}$$

Proof

we prove only (2).

$$\begin{array}{lll}
 & while\ b\ do\ p \equiv (b \wedge p)^* \wedge fin(\neg b) & \text{S-while} \\
\equiv & (empty \vee (b \wedge p \wedge more; (b \wedge p)^*)) \wedge fin(\neg b) \vee b \wedge p \wedge \Box more \wedge fin(\neg b) & \text{theorem 4.5} \\
\equiv & empty \wedge \Box(empty \rightarrow \neg b) \vee (b \wedge p \wedge more; (b \wedge p)^*) \wedge fin(\neg b) \vee b \wedge p \wedge \Box more & (\Box more) \wedge fin(\neg b) \\
 & & \equiv \Box more \\
\equiv & empty \wedge (empty \rightarrow \neg b) \vee (b \wedge p \wedge more; (b \wedge p)^*) \wedge fin(\neg b) \vee b \wedge p \wedge \Box more & \text{FE2} \\
\equiv & (empty \wedge \neg b) \vee (b \wedge p \wedge more; fin(\neg b) \wedge (b \wedge p)^*)) \vee b \wedge p \wedge \Box more & \text{lemma 4.6} \\
\equiv & (empty \wedge \neg b) \vee (b \wedge p \wedge more; while\ b\ do\ p) \vee b \wedge p \wedge \Box more & \text{S-while}
\end{array}$$

$\Box$

The statement of Theorem 4.7 can be expressed in a simplified form in terms of the *if – then – else* construct if p always terminates, since then $b \wedge p \wedge \Box more$ is false:

$$\begin{array}{lll}
while\ b\ do\ p & \equiv & (empty \wedge \neg b) \vee (b \wedge p \wedge more; while\ b\ do\ p) \\
 & \equiv & (empty \wedge \neg b) \vee b \wedge (p \wedge more; while\ b\ do\ p) \\
 & \equiv & if\ b\ then\ (p \wedge more; while\ b\ do\ p)\ else\ empty \\
 & & \text{if } p \text{ is a lec-formula.}
\end{array}$$

Similarly, if p is a terminable program ignoring whether or not P is a lec-formula, we have

$$\begin{array}{lll}
while\ b\ do\ p & \equiv & (empty \wedge \neg b) \vee (b \wedge p; while\ b\ do\ p) \\
 & \equiv & (empty \wedge \neg b) \vee b \wedge (p; while\ b\ do\ p) \\
 & \equiv & if\ b\ then\ (p; while\ b\ do\ p)\ else\ empty
\end{array}$$

To reduce a program with existential quantification, we use a renaming method. Given a formula $\exists x : p(x)$ with a bound variable x, we can remove the existential quantification $(\exists x)$ from $\exists x : p(x)$ to obtain a formula $p(y)$ (or $p[y/x]$) with a free variable y by renaming x as y. To do so, we require that y not occur (free or bound) in $\exists x : p(x)$, y and x both be either dynamic or static, and y be substituted for x only within the bound scope of x in $\exists x : p(x)$. In this case, we call $p(y)$ a renamed formula of $\exists x : p(x)$. Now we discuss some facts concerning renamed formulas.

Lemma 4.8 Let $p(y)$ be a renamed formula of $\exists x : p(x)$. Then, $\exists x : p(x)$ is satisfiable if and only if $p(y)$ is satisfiable. Furthermore, any model of $p(y)$ is a model of $\exists x : p(x)$.

Proof

By I-exists, given a model σ, the following is true:

$$\sigma \models \exists y : p(y) \text{ iff there exists } \sigma',\ \sigma \overset{y}{=} \sigma', \text{ and } \sigma' \models p(y)$$

Thus, if $\sigma \models \exists y : p(y)$ then there is a σ', $\sigma' \models p(y)$. Conversely, for a model σ, if $\sigma \models p(y)$, then $\sigma \models \exists y : p(y)$ because σ is trivially y-equivalent to itself. Thus, $\exists y : p(y)$ is satisfiable iff $p(y)$ is satisfiable; and any model of $p(y)$ is a model of $\exists y : p(y)$. Moreover, by Theorem 3.32, $\exists y : p(y) \equiv \exists x : p(x)$. Hence, Lemma 4.8 is true.

□

The importance of Lemma 4.8 is that it guarantees the soundness of the renaming method for reducing programs involving existential quantifications. From Lemma 4.8, it is clear that $\exists x : p(x)$ and $p(y)$ are equivalent in satisfiability. To check whether or not $\exists x : p(x)$ is satisfiable amounts to checking whether or not $p(y)$ is satisfiable. Once we find a model for $p(y)$, this model is also a model for $\exists x : p(x)$. Therefore, to reduce $\exists x : p(x)$, it is sufficient to reduce $p(y)$.

One may object to the renaming method by saying that it offends the original spirit by using local variables in a program. However, we are investigating the temporal semantics of a program under the logic model theory. To interpret a formula with an existential quantification, the only interpretation law we can use is I-exists which does not seem to distinguish between local and global variables at all.

We now define the normal form of a program, as follows

Definition 4.2 A program q is in normal form if

$$q \overset{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj} \qquad (4.1)$$

where $0 \leq l \leq 1$, $t \geq 0$, and $l + t \geq 1$.

For $1 \leq j \leq t$, $\bigcirc q_{fj}$ are lec-formulas and q_{fj} are internal programs; whereas q_{ei} $(i = 1)$ and q_{cj} $(1 \leq j \leq t)$ are either equal to *true*, or are state formulas of the form:

$$(x_1 = e_1) \wedge \ldots \wedge (x_m = e_m)$$

where $e_k \in D$ $(1 \leq k \leq m)$. □

Note that, in the above definition, (1) $\bigvee_{i=1}^{0} p_i \equiv false$; (2) a program is an internal program if the variables involved in the program may refer to the previous state but not beyond the first state over the current interval. To ensure this condition, two adjacent previous operators are not permitted to be applied to a variable within a program. For instance, $\bigcirc\circleddash\circleddash x$ is not permitted but $\bigcirc\circleddash \bigcirc \circleddash x$ can be used in a program.

Note also that the normal form we provide is a semantics driven reduction for program rather than syntax determined normal form. Hence, in the proof of the normal form in the sequel, the logic laws presented in the thesis can be used and the evaluation of the expressions (including boolean expressions) can be involved.

If a program is deterministic then the normal form of the program has a simpler form with $l+t=1$. That is,

$$q_e \wedge empty \quad \text{or} \quad q_c \wedge \bigcirc q_f$$

This means that if q terminates at the current state it is reduced to $q_e \wedge empty$ otherwise it is reduced to $q_c \wedge \bigcirc q_f$.

Theorem 4.7, Lemma 4.8 and logic laws presented in Chapters 2 and 3 provide a basis for proving the existence of the normal form of a program. The normal form is closely related to the reduction of programs. A program p is reduced to q means that there are programs $p_1, ..., p_h$ such that $p \equiv p_1 \equiv ... \equiv p_h \equiv q$.

Theorem 4.9 Let P be a program. Then there is a normal form q as defined in (4.1) such that

$$P \equiv q$$

Proof

To simplify the proof, we assume that the expressions involved in a program are all well evaluated at every state during a reduction. The proof proceeds by induction on the structure of statements. The atomic statements are the immediate assignment $x = e$ and *empty*; whereas the compositional statements are $\bigcirc p$, $\Box p$, $p \wedge q$, $p; q$, *if b then p else q*, *while b do p*, $p \| q$ and $\exists x : p(x)$. In the proof, we use implicitly Theorem 3.7.

1) If P is an assignment statement, $x = e$, then we have

$$\begin{aligned} x = e &\equiv x = e \wedge (empty \vee more) \\ &\equiv x = e \wedge empty \vee x = e \wedge \bigcirc true \end{aligned}$$

As seen, an immediate assignment is a non-deterministic program. It needs other constructs to specify a definite interval. If an empty interval is specified, it is reduced to $x = e \wedge empty$; otherwise, it is reduced to $x = e \wedge \bigcirc true$.

2) If P is the terminal statement *empty*, the conclusion is trivially true.

3) If P is a statement in the form $\bigcirc q$, it is already in its normal form.

4) If P is the always statement $\Box q$, then

$$\begin{aligned} P &\equiv q \wedge \odot \Box q && \text{FE2} \\ &\equiv (\bigvee_{j=1}^{l} q_{ej} \wedge empty \vee \bigvee_{i=1}^{t} q_{ci} \wedge \bigcirc q_{fi}) \wedge (empty \vee \bigcirc \Box q) && \text{hypothesis, FW2} \\ &\equiv \bigvee_{j=1}^{l} q_{ej} \wedge empty \vee \bigvee_{i=1}^{t} (q_{ci} \wedge \bigcirc q_{fi} \wedge \bigcirc \Box q) && \text{FDU5} \\ &\equiv \bigvee_{j=1}^{l} q_{ej} \wedge empty \vee \bigvee_{i=1}^{t} q_{ci} \wedge \bigcirc (q_{fi} \wedge \Box q) && \text{FD3} \end{aligned}$$

5) If P is the conjunction statement $p \wedge q$, then, from the hypothesis, p and q have the normal form

$$p \equiv \bigvee_{k=1}^{l_1} p_{ek} \wedge empty \vee \bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc p_{fi}$$

$$q \equiv \bigvee_{h=1}^{l_2} q_{eh} \land empty \lor \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}$$

Thus, we can rewrite P as

$$\begin{aligned}
P &\equiv p \land q \\
&\equiv (\bigvee_{k=1}^{l_1} p_{ek} \land empty \lor \bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc p_{fi}) \land (\bigvee_{h=1}^{l_2} q_{eh} \land empty \lor \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}) && \text{hypothesis} \\
&\equiv \bigvee_{k=1}^{l} p_{ek} \land q_{ek} \land empty \lor (\bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc p_{fi}) \land (\bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc p_{fj}) && \text{FDU5, FS4} \\
&\equiv \bigvee_{k=1}^{l} p_{ek} \land q_{ek} \land empty \lor \bigvee_{1 \le i \le t_1, 1 \le j \le t_2} p_{ci} \land q_{cj} \land \bigcirc (p_{fi} \land q_{fj}) && \text{FD3}
\end{aligned}$$

6) If P is a sequential statement $p; q$, then, by hypothesis, we have

$$p \equiv \bigvee_{k=1}^{l_1} p_{ek} \land empty \lor \bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc p_{fi}$$

$$q \equiv \bigvee_{h=1}^{l_2} q_{eh} \land empty \lor \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}$$

Thus, we can rewrite P, as follows

$$\begin{aligned}
p; q &\equiv (\bigvee_{k=1}^{l_1} p_{ek} \land empty \lor \bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc p_{fi}); (\bigvee_{h=1}^{l_2} q_{eh} \land empty \lor \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}) && \text{hypothesis} \\
&\equiv (\bigvee_{k=1}^{l_1} p_{ek} \land empty; \bigvee_{h=1}^{l_2} q_{eh} \land empty) \\
&\quad \lor (\bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc p_{fi}; \bigvee_{h=1}^{l_2} q_{eh} \land empty) \\
&\quad \lor (\bigvee_{k=1}^{l_1} p_{ek} \land empty; \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}) \\
&\quad \lor (\bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc p_{fi}; \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}) && \text{FD9, 10} \\
&\equiv \bigvee_{k=1}^{l} p_{ek} \land q_{ek} \land empty \\
&\quad \lor \bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc (p_{fi}; q_{e1} \land empty) \\
&\quad \lor \bigvee_{j=1}^{t_2} p_{e1} \land q_{cj} \land \bigcirc q_{fj} \\
&\quad \lor \bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc (p_{fi}; \bigvee_{j=1}^{t_2} q_{cj} \land \bigcirc q_{fj}) && \text{EMP4, FCH1, FD9, 10} \\
&\equiv \bigvee_{k=1}^{l} p_{ek} \land q_{ek} \land empty \\
&\quad \lor \bigvee_{i=1}^{t_1} p_{ci} \land \bigcirc (p_{fi}; q_{e1} \land empty) \\
&\quad \lor \bigvee_{j=1}^{t_2} p_{e1} \land q_{cj} \land \bigcirc q_{fj} \\
&\quad \lor \bigvee_{1 \le i \le t_1, 1 \le j \le t_2} p_{ci} \land \bigcirc (p_{fi}; q_{cj} \land \bigcirc q_{fj}) && \text{FD9, 4}
\end{aligned}$$

7) If P is the parallel statement $p||q$, then, by hypothesis, we have

$$p \equiv \bigvee_{k=1}^{l_1} p_{ek} \wedge empty \vee \bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc p_{fi}$$

$$q \equiv \bigvee_{h=1}^{l_2} q_{eh} \wedge empty \vee \bigvee_{j=1}^{t_2} q_{cj} \wedge \bigcirc q_{fj}$$

We first prove the following conclusion:

$$\begin{array}{rcll}
(q; true) & \equiv & (\bigvee_{h=1}^{l_2} q_{eh} \wedge empty \vee \bigvee_{j=1}^{t_2} q_{cj} \wedge \bigcirc q_{fj}); true & \text{hypothesis} \\
& \equiv & (\bigvee_{h=1}^{l_2} q_{eh} \wedge empty; true) \vee (\bigvee_{j=1}^{t_2} q_{cj} \wedge \bigcirc q_{fj}); true & \text{FD10} \\
& \equiv & \bigvee_{h=1}^{l_2} q_{eh} \wedge (empty; true) \vee \bigvee_{j=1}^{t_2} q_{cj} \wedge \bigcirc (q_{fj}; true) & \text{FCH1, 2} \\
& \equiv & \bigvee_{h=1}^{l_2} q_{eh} \wedge true \vee \bigvee_{j=1}^{t_2} q_{cj} \wedge \bigcirc (q_{fj}; true) & empty; true \equiv true \\
& \equiv & \bigvee_{h=1}^{l_2} q_{eh} \wedge empty \vee q_{e1} \wedge \bigcirc true \vee \bigvee_{j=1}^{t_2} q_{cj} \wedge \bigcirc (q_{fj}; true) & \text{tautology} \\
& \equiv & \bigvee_{h=1}^{l_2} q_{eh} \wedge empty \vee \bigvee_{j=1}^{t_2+1} q_{cj} \wedge \bigcirc (q_{fj}; true) &
\end{array}$$

where $q_{ct_2+1} \equiv q_{e1}$ and $q_{ft_2+1} \equiv true$. Thus,

$$\begin{array}{rcll}
p \wedge (q; true) & \equiv & (\bigvee_{k=1}^{l_1} p_{ek} \wedge empty \vee \bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc p_{fi}) & \\
& & \wedge (\bigvee_{h=1}^{l_2} q_{eh} \wedge empty \vee \bigvee_{j=1}^{t_2+1} q_{cj} \wedge \bigcirc (q_{fj}; true)) & \text{hypothesis} \\
& \equiv & \bigvee_{k=1}^{l} p_{ek} \wedge q_{ek} \wedge empty \vee (\bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc p_{fi}) & \\
& & \wedge \bigvee_{j=1}^{t_2+1} q_{cj} \wedge \bigcirc (q_{fj}; true) & \text{FDU5} \\
& \equiv & \bigvee_{k=1}^{l} p_{ek} \wedge q_{ek} \wedge empty & \\
& & \vee \bigvee_{1 \leq i \leq t_1, 1 \leq j \leq t_2+1} p_{ci} \wedge q_{cj} \wedge \bigcirc (p_{fi} \wedge (q_{fj}; true)) &
\end{array}$$

In the same way, we can prove

$$q \wedge (p; true) \equiv \bigvee_{k=1}^{l} q_{ek} \wedge p_{ek} \wedge empty \vee (\bigvee_{1 \leq i \leq t_1+1, 1 \leq j \leq t_2} q_{ci} \wedge p_{cj} \wedge \bigcirc (q_{fi} \wedge (p_{fj}; true))$$

Hence,

$$\begin{aligned}
P &\equiv p\wedge(q;true)\vee q\wedge(p;true) && \text{definition}\\
&\equiv \bigvee_{k=1}^{l} p_{ek}\wedge q_{ek}\wedge empty \vee (\bigvee_{1\le i\le t_1, 1\le j\le t_2+1} p_{ci}\wedge q_{cj}\wedge \bigcirc(p_{fi}\wedge(q_{fj};true))\\
&\quad \vee \bigvee_{k=1}^{l} q_{ek}\wedge p_{ek}\wedge empty \vee (\bigvee_{1\le i\le t_1+1, 1\le j\le t_2} q_{ci}\wedge p_{cj}\wedge \bigcirc(q_{fi}\wedge(p_{fj};true))\\
&\equiv \bigvee_{k=1}^{l} p_{ek}\wedge q_{ek}\wedge empty \vee (\bigvee_{1\le i\le t_1, 1\le j\le t_2+1} p_{ci}\wedge q_{cj}\wedge \bigcirc(p_{fi}\wedge(q_{fj};true))\\
&\quad \vee(\bigvee_{1\le i\le t_1+1, 1\le j\le t_2} q_{ci}\wedge p_{cj}\wedge \bigcirc(q_{fi}\wedge(p_{fj};true))
\end{aligned}$$

8) If P is the conditional statement *if b then p else q*, then, by hypothesis, we have

$$p\equiv \bigvee_{k=1}^{l_1} p_{ek}\wedge empty\vee \bigvee_{i=1}^{t_1} p_{ci}\wedge \bigcirc p_{fi}$$

$$q\equiv \bigvee_{h=1}^{l_2} q_{eh}\wedge empty\vee \bigvee_{j=1}^{t_2} q_{cj}\wedge \bigcirc q_{fj}$$

Thus,

$$\begin{aligned}
&if\ b\ then\ p\ else\ q\\
\equiv\ & b\wedge p\vee\neg b\wedge q\\
\equiv\ & \begin{cases} p & \text{if } b\\ q & \text{otherwise}\end{cases}
\end{aligned}$$

Therefore, if b is true according to a context in which P is executed then the construct is reduced to

$$p\equiv \bigvee_{k=1}^{l_1} p_{ek}\wedge empty\vee \bigvee_{i=1}^{t_1} p_{ci}\wedge \bigcirc p_{fi}$$

Otherwise, it is reduced to

$$q\equiv \bigvee_{h=1}^{l_2} q_{eh}\wedge empty\vee \bigvee_{j=1}^{t_2} q_{cj}\wedge \bigcirc q_{fj}$$

9) If P is the while statement *while b do p*, then, by hypothesis, we have

$$p\equiv \bigvee_{k=1}^{l_1} p_{ek}\wedge empty\vee \bigvee_{i=1}^{t_1} p_{ci}\wedge \bigcirc p_{fi}$$

By Theorem 4.7, we have

$$while\ b\ do\ p\ \equiv\ \neg b\wedge empty\vee(b\wedge p\wedge more; while\ b\ do\ p)\vee p\wedge b\wedge \Box more$$

Therefore, if b is false at the current state according to a context in which P is executed, then the *while* statement is reduced to *empty*. Otherwise, it is reduced to the chop construct

$p \wedge more; while\ b\ do\ p \vee p \wedge \Box more$. This is immediately re-reduced, as follows

$p \wedge more; while\ b\ do\ p\ \vee p \wedge \Box more$	
$\equiv\ p \wedge (\bigcirc true; while\ b\ do\ p\ \vee more \wedge \odot \Box more)$	FE2
$\equiv\ p \wedge (\bigcirc(true; while\ b\ do\ p\ ) \vee \bigcirc\Box more)$	FW1, FCH1
$\equiv\ p \wedge \bigcirc((true; while\ b\ do\ p\ ) \vee \Box more)$	FD4
$\equiv\ (\bigvee_{k=1}^{l_1} p_{ek} \wedge empty \vee \bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc p_{fi}) \wedge \bigcirc((true; while\ b\ do\ p\ ) \vee \Box more)$	hypothesis
$\equiv\ (\bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc p_{fi}) \wedge \bigcirc((true; while\ b\ do\ p\ ) \vee \Box more)$	hypothesis
$\equiv\ \bigvee_{i=1}^{t_1} p_{ci} \wedge \bigcirc(p_{fi} \wedge ((true; while\ b\ do\ p\ ) \vee \Box more))$	FD3, FD9

10) If P is the statement $\exists x : p(x)$, then we first use the renaming method to reduce this to $p(y)$ as stated in Lemma 4.8. Then, we re-reduce $p(y)$. By induction hypothesis, $p(y)$ can be reduced to its normal form, so is the construct $\exists x : p(x)$. □

The following expansion laws regarding fin, $keep$ and $halt$ are useful for reduction of programs.

Theorem 4.10 The following formulas are valid.

$$\begin{array}{lll} FEF & fin(p) & \equiv\ p \wedge empty \vee \bigcirc fin(p) \\ FEK & keep(p) & \equiv\ empty \vee p \wedge \bigcirc keep(p) \\ FEH & halt(p) & \equiv\ p \wedge empty \vee \neg p \wedge \bigcirc halt(p) \end{array}$$

Proof

The proof of FEF

$fin(p) \equiv \Box(empty \rightarrow p)$	definition
$\equiv (empty \rightarrow p) \wedge \odot\Box(empty \rightarrow p)$	FE2
$\equiv (more \vee p) \wedge (empty \vee \bigcirc fin(p))$	theorem 3.1, definition
$\equiv p \wedge empty \vee p \wedge \bigcirc fin(p) \vee \bigcirc fin(p) \wedge more$	tautology
$\equiv p \wedge empty \vee p \wedge \bigcirc fin(p) \vee \bigcirc fin(p)$	FS4
$\equiv p \wedge empty \vee \bigcirc fin(p)$	theorem 3.1, theorem 3.7

The proof of FEK

$keep(p) \equiv \Box(\neg empty \rightarrow p)$	definition
$\equiv (\neg empty \rightarrow p) \wedge \odot\Box(\neg empty \rightarrow p)$	FE2
$\equiv (empty \vee p) \wedge (empty \vee \bigcirc keep(p))$	theorem 3.1, definition
$\equiv empty \vee p \wedge empty \vee p \wedge \bigcirc keep(p)$	tautology
$\equiv empty \vee p \wedge \bigcirc keep(p)$	theorem 3.1, theorem 3.7

The proof of FEH

$halt(p) \equiv \Box(empty \leftrightarrow p)$	definition
$\equiv (empty \leftrightarrow p) \wedge \odot\Box(empty \leftrightarrow p)$	FE2
$\equiv (\neg empty \wedge \neg p \vee empty \wedge p) \wedge (empty \vee \bigcirc halt(p))$	tautology
$\equiv p \wedge empty \vee (\neg p \wedge more) \wedge \bigcirc halt(p)$	theorem 3.1, definition
$\equiv p \wedge empty \vee \neg p \wedge \bigcirc halt(p)$	theorem 3.1, theorem 3.7

□

As seen, *halt*(p), in general, is not a deterministic program because it refers to the negation of p. However, we can take p to be a boolean condition b which may involve the negation operator. In practice, we frequently use *halt*(b) to specify an interval for a program.

To execute a program q is really to find an interval to satisfy the program q. Based on the Tableau method [88], the execution of a program q consists of a series of reductions over a sequence of states, i.e. an interval. At each internal state s_i, the program is reduced to the normal form defined in Definition 4.2 with the index i:

$$q_c^i \wedge \bigcirc q_f^i$$

while at the final state s_j, the program is reduced to:

$$q_e^j \wedge empty$$

Furthermore, the reduction at each state is composed of a series of reduction steps supported by logic laws. In this way, if a program is eventually reduced to *true*, then an interval which satisfies the program is obtained. With this method, the strong equivalence relation for each step in the reduction of a program over a subinterval is required since variables may refer to their previous values; whereas, at a chop point for a reduction, $p; q$ say, the weak equivalence relation is enough since the variables within p are not permitted to refer to q and the variables within q are not permitted to refer to p as specified in the definition of semantics (see I-chop, page 33).

As seen, some logic laws such as $empty; p \approx p$ and $q \wedge empty; p \approx p \wedge q$ (q is a state formula) hold without requiring that p be a lec-formula and are useful for the reduction at a chop point. Therefore, a program could involve some past operators and the reduction of this kind of program can be handled in the underlying logic. However, to simplify the matter, we confine ourself to the set of lec-formulas since all programs are lec-formulas as stated earlier.

Example 4.3 Let P denote the program $x = 0 \wedge while\ x < 3\ do\ x := x + 1$. The following is the process of the reduction of P.

$$\begin{array}{lll}
P & \equiv & x = 0 \wedge while\ x < 3\ do\ x := x + 1 \\
 & \equiv & x = 0 \wedge if\ x < 3\ then\ (x := x + 1 \wedge more; while\ x < 3\ do\ x := x + 1)\ else\ empty \\
 & \equiv & x = 0 \wedge (\ 0 < 3 \wedge (\bigcirc x = 0 + 1 \wedge skip; while\ x < 3\ do\ x := x + 1)\ \vee \neg(0 < 3) \wedge empty) \\
 & \equiv & x = 0 \wedge (\bigcirc x = 1 \wedge skip; while\ x < 3\ do\ x := x + 1) \\
 & \equiv & x = 0 \wedge \bigcirc(x = 1 \wedge empty; while\ x < 3\ do\ x := x + 1)
\end{array}$$

Thus,

$$P_c^0 \equiv x = 0$$

$$P_f^0 \equiv x = 1 \wedge empty; while\ x < 3\ do\ x := x + 1$$

$$\begin{array}{lll}
P_f^0 & \equiv & x = 1 \wedge empty; while\ x < 3\ do\ x := x + 1 \\
 & \equiv & x = 1 \wedge while\ x < 3\ do\ x := x + 1 \\
 & \equiv & x = 1 \wedge if\ x < 3\ then\ (x := x + 1 \wedge more; while\ x < 3\ do\ x := x + 1)\ else\ empty \\
 & \equiv & x = 1 \wedge (\ 1 < 3 \wedge (\bigcirc x = 1 + 1 \wedge skip; while\ x < 3\ do\ x := x + 1)\ \vee \neg(1 < 3) \wedge empty) \\
 & \equiv & x = 0 \wedge (\bigcirc x = 1 \wedge skip; while\ x < 3\ do\ x := x + 1) \\
 & \equiv & x = 1 \wedge \bigcirc(x = 2 \wedge empty; while\ x < 3\ do\ x := x + 1)
\end{array}$$

Thus,

$$P_c^1 \equiv x = 1$$

$$P_f^1 \equiv x = 2 \wedge empty; while\ x < 3\ do\ x := x + 1$$

Further,

$$\begin{aligned}
P_f^1 &\equiv x = 2 \wedge empty; while\ x < 3\ do\ x := x + 1 \\
&\equiv x = 2 \wedge while\ x < 3\ do\ x := x + 1 \\
&\equiv x = 2 \wedge if\ x < 3\ then\ (x := x + 1 \wedge more; while\ x < 3\ do\ x := x + 1)\ else\ empty \\
&\equiv x = 2 \wedge (\ 2 < 3 \wedge (\bigcirc x = 2 + 1 \wedge skip; while\ x < 3\ do\ x := x + 1)\ \vee \neg(2 < 3) \wedge empty) \\
&\equiv x = 0 \wedge (\bigcirc x = 1 \wedge skip; while\ x < 3\ do\ x := x + 1) \\
&\equiv x = 2 \wedge \bigcirc(x = 3 \wedge empty; while\ x < 3\ do\ x := x + 1)
\end{aligned}$$

We obtain

$$P_c^2 \equiv x = 2$$

$$P_f^2 \equiv x = 3 \wedge empty; while\ x < 3\ do\ x := x + 1$$

Finally,

$$\begin{aligned}
P_f^2 &\equiv x = 3 \wedge empty; while\ x < 3\ do\ x := x + 1 \\
&\equiv x = 3 \wedge while\ x < 3\ do\ x := x + 1 \\
&\equiv x = 3 \wedge if\ x < 3\ then\ (x := x + 1 \wedge more; while\ x < 3\ do\ x := x + 1)\ else\ empty \\
&\equiv x = 3 \wedge (\ 3 < 3 \wedge (x := x + 1 \wedge more; while\ x < 3\ do\ x := x + 1)\ \vee \neg(3 < 3) \wedge empty) \\
&\equiv x = 3 \wedge empty
\end{aligned}$$

We obtain

$$P_e \equiv x = 3 \wedge empty$$

□

In this chapter, an extended Tempura language has been presented and a normal form of a program has been proved under the assumption that the past operators are not used within statements but can be used within expressions. This provides us a way to execute programs.

Chapter 5

Projection in Temporal Logic Programming

Summary: A new projection operator is defined as a primitive in the Temporal logic. Its syntax and semantics are presented and illustrated with examples. Some logic laws concerning the projection operator are provided. In the framework of the temporal logic programming, the normal form of the projection construct is proved.

In a temporal logic programming language, such as Tempura [61], the next, always and chop are useful operators for sequential programs, while conjunction and parallel composition (||, see Chapter 4), are basic operators for concurrent programming. As discussed earlier, conjunction construct seems appropriate for dealing with fine-grained parallel operations that proceed in lock-step while the parallel composition operator is better suited to the coarse-grained concurrency, where each process proceeds at its own speed. Moreover, processes combined through the parallel composition operator share all the states and may interfere with one another.

In this chapter we introduce a projection operator, $(p_1, \ldots, p_m)\ prj\ q$, which can be thought of as a combination of the parallel and the projection operators presented in [61]. Intuitively, it means that q is executed in parallel with $p_1; \ldots; p_m$ over an interval obtained by taking the endpoints (rendezvous points) of the intervals over which $p_1, \ldots, p_m$ are executed. The projection construct permits the processes $p_1, \ldots, p_m, q$ to be autonomous, each process having the right to specify the interval over which it is executed. In particular, the sequence of processes $p_1, \ldots, p_m$ and process q may terminate at different time points. Although the communication between processes is still based on shared variables, the communication and synchronization only take place at the rendezvous points (global states), otherwise they are executed independently.

This chapter is organized as follows: Section 5.1 presents the new projection operator; the logic laws about the projection operator are proved in Section 5.2; in Section 5.3, the normal form of the projection construct, as a program statement, is proved, and an example of reduction of the projection construct is presented; Section 5.4 provides further examples for the application of the projection. Section 5.5 draws conclusions.

5.1 Syntax and Semantics

The projection construct is defined as

$$(p_1, \ldots, p_m)\ prj\ q$$

where $p_1, \ldots, p_m$ and q are formulas ($m \geq 1$). To ensure smooth synchronization between $p_1, \ldots, p_m$ and q, the previous operator is not allowed within q. However, it can be used in the p_l's. To define the semantics of the projection operator we need an auxiliary operator for intervals.

Let $\sigma = < s_0, s_1, \ldots s_{|\sigma|} >$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \preceq |\sigma|$. The *projection* of σ onto $r_1, \ldots, r_h$ is the interval

$$\sigma\downarrow(r_1, \ldots, r_h) = < s_{t_1}, s_{t_2}, \ldots, s_{t_l} >$$

where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. That is, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. For example,

$$< s_0, s_1, s_2, s_3, s_4 >\downarrow(0, 0, 2, 2, 2, 3) = < s_0, s_2, s_3 > .$$

The semantics of the projection operator is defined, as before, relative to an interpretation $\mathcal{I} = (\sigma, i, k, j)$. Formally,

$$(I - prj1) \quad \mathcal{I} \models (p_1, \ldots, p_m)\ prj\ q$$

iff $\sigma\downarrow(k) \models q$ and $\mathcal{I} \models p_1; \ldots; p_m$, or there are integers $r_1, r_2, \ldots, r_h$ ($1 \leq h \leq m$) such that $k \leq r_1 \leq r_2 \leq \ldots \leq r_h \preceq j$ and the following hold:

- $(\sigma, i, k, r_1) \models p_1$, and for $1 < l \leq h$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$.
- If $h < m$ then $\sigma\downarrow(k, r_1, \ldots, r_h) \models q$ and $(\sigma, r_h, r_h, j) \models p_{h+1}; \ldots; p_m$.
- If $h = m$ then $\sigma\downarrow(k, r_1, \ldots, r_h)\cdot\sigma_{(r_h+1..j)} \models q$.

Basically, the above definition of the projection is the same as in [22]. For ease of the proofs in the sequel, we define a notation, $[p_1, \ldots, p_m](r_1, \ldots, r_m)$, for formulas $p_1, \ldots, p_m$ and integers $r_1, \ldots, r_{m-1}$ and $r_m \in N_\omega$ ($r_1 \leq \ldots \leq r_{m-1} \preceq r_m$), to mean that the formulas $p_1, \ldots, p_m$ are executed sequentially over a subinterval and $r_1, \ldots, r_{m-1}$ are the partition points and r_m is the right end point of the subinterval. Formally, let $\mathcal{I} = (\sigma, i, k, j)$ be an interpretation, then

$$\begin{array}{ll} & (\sigma, i, k, j) \models [p_1, \ldots, p_m](r_1, \ldots, r_m) \\ \text{iff} & (\sigma, i, k, r_1) \models p_1 \text{ and for all } 1 < h \leq m, (\sigma, r_{h-1}, r_{h-1}, r_h) \models p_h \end{array}$$

We need also to generalize the notation of $\sigma\downarrow(r_1, \ldots, r_m)$ to allow r_i to be ω. For an interval $\sigma = < s_0, s_1, \ldots, s_{|\sigma|} >$ and $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \leq |\sigma|$ ($r_i \in N_\omega$), we define

$$\sigma\downarrow() = \epsilon$$

$$\sigma\downarrow(r_1, \ldots, r_h, \omega) = \sigma\downarrow(r_1, \ldots, r_h)$$

and, for integers $r_1, \ldots, r_h$, $\sigma\downarrow(r_1, \ldots, r_h)$ is defined as before.

Thus, the meaning of the projection construct can be expressed in terms of $[p_1, \ldots, p_m](r_1, \ldots, r_m)$ as follows

$$(I - prj2) \quad (\sigma, i, k, j) \models (p_1, \ldots, p_m)\ prj\ q$$

iff there exist integers $r_1, \ldots, r_{m-1}$ and $r_m \in N_\omega$ and $r_0 = k$ such that $(\sigma, i, k, j) \models [p_1, \ldots, p_m](r_1, \ldots, r_m)$ and

- $r_m = j$ and $\sigma \downarrow (r_0, \ldots, r_h) \models q$ for some $0 \leq h \leq m$ or

- $r_m < j$ and $\sigma \downarrow (r_0, ..., r_m).\sigma_{(r_m+1..j)} \models q$.

One can show that the semantics given by I-prj1 and I-prj2 are equivalent. We use I-prj2 only in a long proof.

In programming language terms, the interpretation of $(p_1, \ldots, p_m)$ *prj* q is that we need *two* sequences of clocks (states) running on different time scales: one is a local state sequence, over which $p_1, \ldots, p_m$ are executed, the other is a global state sequence over which q is executed. Process q is executed in a parallel manner with the sequence of processes $p_1, \ldots, p_m$. The execution proceeds as follows (see Figure 5.1): First, q and p_1 start at the first global state and p_1 is executed over a sequence of local states until its termination. Then (the remaining part of) q and p_2 are executed at the second global state. Subsequently, p_2 is continuously executed over a sequence of local states until its termination, and so on. Although q and p_1 start at the same time, $p_1, \ldots, p_m$ and q may terminate at different time points. If q terminates before some p_{h+1}, then, subsequently, $p_{h+1}, \ldots, p_m$ are executed sequentially. If $p_1, \ldots, p_m$ are finished before q, then the execution of q is continued until its termination.

```
t0        t2        t4        t6
|---------|---------|---------|
|<------------q-------------->|
t0   t1   t2   t3   t4   t5   t6   t7   t8
|----|----|----|----|----|----|----|----|
|<---p1-->|<---p2-->|<---p3-->|<---p4-->|

(a): q terminates before p4

t0        t2        t4        t6        t8   t9   t10
|---------|---------|---------|---------|----|----|
|<-------------------q--------------------------->|
t0   t1   t2   t3   t4   t5   t6   t7   t8
|----|----|----|----|----|----|----|----|
|<---p1-->|<---p2-->|<---p3-->|<---p4-->|

(b): p4 terminates before q

t0        t2        t4        t6        t8
|---------|---------|---------|---------|
|<-------------------q----------------->|
t0   t1   t2   t3   t4   t5   t6   t7   t8
|----|----|----|----|----|----|----|----|
|<---p1-->|<---p2-->|<---p3-->|<---p4-->|

(c): q and p4 terminate at the same point
```

Figure 5.1: Possible executions of (p1,p2,p3,p4) prj q

Note that the projection construct can be executed over an infinite interval. In this case, if p_m terminates before q, then q is continuously executed over the remaining infinite subinterval; whereas if q terminates before some p_l $(1 \leq l \leq m)$, then $p_l; ...; p_m$ is sequentially executed over the remaining infinite subinterval. This implies p_m is executed over an infinite subinterval.

Projection can be thought of as a special parallel computation which is executed on different time scales.

Example 5.1 Consider the following formulas:

$$p_1 \overset{\text{def}}{=} len(2) \wedge keep(i \circ\!\!= i+2)$$

$$p_2 \overset{\text{def}}{=} len(4) \wedge keep(i \circ\!\!= i+3)$$

$$p_3 \overset{\text{def}}{=} len(6) \wedge keep(i \circ\!\!= i+4)$$

$$q \overset{\text{def}}{=} len(4) \wedge (i = 2) \wedge (j = 0) \wedge keep(j \circ\!\!= j+i).$$

Then executing $(p_1, p_2, p_3)\ prj\ q$ yields the following result:

```
 t0      t2                t6                         t12 t13
 |-------|---------------|-----------------------|---|
 |------------------------q---------------------------->|
 t0  t1  t2  t3  t4  t5  t6  t7  t8  t9  t10 t11 t12
 |---|---|---|---|---|---|---|---|---|---|---|---|
 |<--p1->|<-----p2------>|<--------- p3--------->|
i=2  4   6   9   12  15  18  22  26  30  34  38  42
j=0      2               8                       26  68
```

Figure 5.2: Projection computation

□

The original projection operator defined in [61], $p\ proj\ q$, and the new projection operator defined above are not directly comparable. In the former, the formula p is executed repeatedly over a series of consecutive subintervals whose endpoints form the interval over which q is executed. This may result in repeating the same global state in the execution of q several times if some of the copies of p are executed over subintervals of zero length (in contrast, our definition rules this out). Moreover, in $p\ proj\ q$, the series of p's and the q always terminate at the same state. We feel that although $p\ proj\ q$ and $(p_1, \ldots, p_m)\ prj\ q$ do share some important properties, they still possess sufficiently distinct features to be treated independently as complementary constructs useful in the programming environment in which different time scales need to be considered.

5.2 Properties of Projection Operator

Projection operator enjoys a number of interesting properties. The theorems below are intended to formalize some of them. First, let us introduce some fundamental notions and auxiliary lemmas which are needed for proving the properties.

Definition 5.1

1 A formula p is called a terminal formula if $p \equiv p \wedge empty$.
2 A formula p is called a local formula if p is a state formula or a terminal formula.
3 A formula p is called *non-local* if for all σ, $\sigma \models p$ implies $|\sigma| \geq 1$.

□

A local formula can be satisfied by a singleton interval, but a non-local formula can not be satisfied by a singleton interval. Also, a terminal formula can be satisfied by a singleton interval. We formalize this fact in Lemma 5.1.

Lemma 5.1 A formula p is a terminal formula iff for all intervals σ, $\sigma \models p$ implies $|\sigma| = 0$.

Proof

Suppose p is a terminal formula. By definition 5.1, $p \equiv p \wedge empty$. Let σ be an interval. If $\sigma \models p$, then $\sigma \models p \wedge empty$. So $\sigma \models empty$, and $|\sigma| = 0$.

Conversely, for all σ, if $\sigma \models p$ implies $|\sigma| = 0$, then $\sigma \models empty$. Hence $\sigma \models p \wedge empty$.

□

We also claim that the following conclusions hold:

Lemma 5.2 Let σ be an interval and $r_1, ..., r_h \in N_\omega$.

$$\begin{array}{lll} 1) & \sigma \downarrow (r_1, ..., r_i, ..., r_i, r_{i+1}, ..., r_h) & = & \sigma \downarrow (r_1, ..., r_i, r_{i+1}, ..., r_h) \\ 2) & \sigma \downarrow (r_1, ..., r_t, r_{t+1}, ..., r_h) & = & \sigma \downarrow (r_1, ..., r_t) \circ \sigma \downarrow (r_t, r_{t+1}, ..., r_h) \end{array}$$

Proof Straightforward.

□

We now turn our attention to the logic laws about the projection formalized in Theorem 5.3 - Theorem 5.9. To prove them, we sometimes use I-prj1 and sometimes use I-prj2. Whenever I-prj2 is used, we claim it explicitly. In what follows, whenever the construct $(p_1, ..., p_m)\ prj\ q$ is encountered, the formula q is thought of as a non-past formula, of course, a lec-formula (see Lemma 2.20).

Theorem 5.3 Let $p_1, ..., p_m, q$ be formulas.

$PRJempty1$	$(empty\ prj\ q) \equiv q$
$PRJempty2$	$(q\ prj\ empty) \equiv q$
$PRJempty3$	$((p_1, \ldots, p_m)\ prj\ empty) \equiv (p_1; \ldots; p_m)$
$PRJempty4$	$(p_1, .., p_t, empty, p_{t+1}, ..., p_m)\ prj\ q \equiv (p_1, ... p_t, p_{t+1}, ..., p_m)\ prj\ q$
$PRJempty5$	$p_1 \wedge empty; (p_2, ..., p_m)\ prj\ q \supset (p_1, p_2, ..., p_m)\ prj\ q$

Proof

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of PRJempty1:

Suppose $(\sigma, 0, k, |\sigma|) \models empty\ prj\ q$. If $\sigma\downarrow(k) \models q$ and $(\sigma, 0, k, |\sigma|) \models empty$ then $k = |\sigma|$ and hence $(\sigma, 0, k, |\sigma|) \models q$. Otherwise, there is r, $k \leq r \preceq |\sigma|$, such that $(\sigma, 0, k, r) \models empty$ and $\sigma\downarrow(k, r)\cdot\sigma_{(r+1..|\sigma|)} \models q$. The former yields $r = k$. Hence $(\sigma, 0, k, |\sigma|) \models q$.

Conversely, if $(\sigma, 0, k, |\sigma|) \models q$ then, by taking $r_1 = k$, one can show that $(\sigma, 0, k, |\sigma|) \models empty\ prj\ q$.

The proof of PRJempty2:

Suppose $(\sigma, 0, k, |\sigma|) \models q\ prj\ empty$. If $\sigma\downarrow(k) \models empty$ and $(\sigma, 0, k, |\sigma|) \models q$ then we are done. Otherwise, there is r, $k \leq r \preceq |\sigma|$, such that $(\sigma, 0, k, r) \models q$ and $\sigma\downarrow(k, r)\cdot\sigma_{(r+1..|\sigma|)} \models empty$. The latter means that $r = |\sigma| = k$. Hence $(\sigma, 0, k, |\sigma|) \models q$.

Conversely, if $(\sigma, 0, k, |\sigma|) \models q$ then, since $\sigma\downarrow(k) \models empty$, $(\sigma, 0, k, |\sigma|) \models q\ prj\ empty$.

The proof of PRJempty3:

Suppose $(\sigma, 0, k, |\sigma|) \models (p_1, \ldots, p_m)\ prj\ empty$. If $\sigma\downarrow(k) \models empty$ and $(\sigma, 0, k, |\sigma|) \models p_1; \ldots; p_m$ then clearly $(\sigma, 0, k, |\sigma|) \models p_1; \ldots; p_m$. Otherwise, there are integers $r_1, \ldots, r_h$ $(1 \leq h \leq m)$ such that $k \leq r_1 \leq \ldots \leq r_h \preceq |\sigma|$ and the following hold:

- $(\sigma, 0, k, r_1) \models p_1$, and for $1 < l \leq h$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$
- If $h < m$ then $\sigma\downarrow(k, r_1, \ldots, r_h) \models empty$ and $(\sigma, r_h, r_h, |\sigma|) \models p_h; \ldots; p_m$.
- If $h = m$ then $\sigma\downarrow(k, r_1, \ldots, r_h)\cdot\sigma_{(r_h+1..|\sigma|)} \models empty$.

If $h < m$ then $r_1 = \cdots = r_h = k$ and hence $(\sigma, 0, k, |\sigma|) \models p_1; \ldots; p_m$. If $h = m$ then $|\sigma| = r_1 = \cdots = r_h = k$, yielding $(\sigma, 0, k, |\sigma|) \models p_1; \ldots; p_m$.

Conversely, if $(\sigma, 0, k, |\sigma|) \models p_1; \ldots; p_m$ then, by taking $h = 1$ and $r_1 = k$, one can show that $(\sigma, 0, k, |\sigma|) \models (p_1, \ldots, p_m)\ prj\ empty$.

The proof of PRJempty4:

We prove this using I-prj2.

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models (p_1, ..., p_t, empty, p_{t+1}, ..., p_m)\ prj\ q \\ \Longleftrightarrow & \text{there exist integers } r_1, ..., r_t, r_e, r_{t+1}, ..., r_{m-1} \text{ and } r_m \in N_\omega \text{ and } r_0 = k \text{ such that} \\ & (\sigma, 0, k, |\sigma|) \models [p_1, ..., p_t, empty, p_{t+1}, ..., p_m](r_1, ..., r_t, r_e, r_{t+1}, ..., r_m) \\ & \text{and} \\ & r_m = |\sigma| \text{ and } \sigma\downarrow(r_0, ..., r_h) \models q \text{ for some } 0 \leq h \leq m \text{ or } h = e \text{ or} \\ & r_m < |\sigma| \text{ and } \sigma\downarrow(r_0, ..., r_t, r_e, r_{t+1}, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models q \end{array}$$

Since

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models [p_1, ..., p_t, empty, p_{t+1}, ..., p_m](r_1, ..., r_t, r_e, r_{t+1}, ..., r_m) \\ \Longleftrightarrow & (\sigma, 0, k, r_1) \models p_1 \text{ and} \\ & (\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l \text{ for all } l, 1 < l \leq t \text{ and} \\ & (\sigma, r_t, r_t, r_e) \models empty \text{ and} \\ & (\sigma, r_e, r_e, r_{t+1}) \models p_{t+1} \text{ and} \\ & (\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l \text{ for all } l, t+1 < l \leq m \\ \Longleftrightarrow & (\sigma, 0, k, r_1) \models p_1 \text{ and} \\ & (\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l \text{ for all } l, 1 < l \leq m\ (r_e = r_t) \\ \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models [p_1, ..., p_t, p_{t+1}, ..., p_m](r_1, ..., r_t, r_{t+1}, ..., r_m) \end{array}$$

Moreover

$$\sigma\downarrow(r_0, ..., r_t, r_e) = \sigma\downarrow(r_0, ..., r_t) \quad (r_t = r_e)$$

and

$$\sigma\downarrow(r_0, ..., r_t, r_e, r_{t+1}, ..., r_h) = \sigma\downarrow(r_0, ..., r_t, r_{t+1}, ..., r_m) \quad (r_t = r_e)$$

Therefore

$$(\sigma, 0, k, |\sigma|) \models (p_1, ..., p_t, empty, p_{t+1}, ..., p_m)\ prj\ q \equiv (p_1, ..., p_t, p_{t+1}, ..., p_m)\ prj\ q$$

The proof of PRJempty5:

We prove this using I-prj2.

$(\sigma, 0, k, |\sigma|) \models p_1 \wedge empty; (p_2, ..., p_m)\ prj\ q$

$\Longleftrightarrow$ $(\sigma, 0, k, r_1) \models p_1 \wedge empty$ and $(\sigma, r_1, r_1, |\sigma|) \models (p_2, ..., p_m)\ prj\ q$ for some r_1, $0 \leq r_1 \preceq |\sigma|$

$\Longleftrightarrow$ $(\sigma, 0, k, r_1) \models p_1$ and $r_1 = k$ and
there exist integers $r_2, ... r_{m-1}$ and $r_m \in N_\omega$ such that
$(\sigma, r_1, r_1, |\sigma|) \models [p_2, ..., p_m](r_2, ..., r_m)$and
$r_m = |\sigma|$ and $\sigma\downarrow(r_1, ..., r_h) \models q$ for some $1 \leq h \leq m$ or
$r_m < |\sigma|$ and $\sigma\downarrow(r_1, ..., r_m) \cdot \sigma_{(r_m+1..|\sigma|)} \models q$

$\Longrightarrow$ there exist integers $r_1, ... r_{m-1}$ and $r_m \in N_\omega$ and $r_0 = k$ such that
$(\sigma, 0, k, |\sigma|) \models [p_1, ..., p_m](r_1, ..., r_m)$
and
$r_m = |\sigma|$ and $\sigma\downarrow(r_0, r_1, ..., r_h) \models q$ for some $0 \leq h \leq m$ or
$r_m < |\sigma|$ and $\sigma\downarrow(r_0, ..., r_m) \cdot \sigma_{(r_m+1..|\sigma|)} \models q$

$\Longleftrightarrow$ $(\sigma, 0, k, |\sigma|) \models (p_1, ..., p_m)\ prj\ q$

□

Theorem 5.4

$PRJskip1$	$(skip\ prj\ q) \equiv q,$	if q is non-local.
$PRJskip2$	$(q\ prj\ skip) \supset q,$	if q is non-local.
$PRJskip3$	$((p,q)\ prj\ skip) \supset (p;q),$	if p or q is non-local.

Proof

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of PRJskip1:

Suppose $(\sigma, 0, k, |\sigma|) \models skip\ prj\ q$. We first observe that $\sigma\downarrow(k) \models q$ and $(\sigma, 0, k, |\sigma|) \models skip$ is impossible since q is non-local. Hence there is r, $k \leq r \preceq |\sigma|$, such that $(\sigma, 0, k, r) \models skip$ and $\sigma\downarrow(k, r) \cdot \sigma_{(r+1..|\sigma|)} \models q$. The former implies $r = k + 1$ and hence $(\sigma, 0, k, |\sigma|) \models q$.

Suppose $(\sigma, 0, k, |\sigma|) \models q$. Then, since q is non-local, $|\sigma| \geq k + 1$. Hence $(\sigma, 0, k, |\sigma|) \models$ $skip\ prj\ q$ can be shown by taking $r_1 = k + 1$.

The proof of PRJskip2:

Suppose $(\sigma, 0, k, |\sigma|) \models q\ prj\ skip$. We first observe that $\sigma\downarrow(k) \models skip$ and $(\sigma, 0, k, |\sigma|) \models q$ is impossible. Hence there is r, $k \leq r \preceq |\sigma|$, such that $(\sigma, 0, k, r) \models q$ and $\sigma\downarrow(k, r) \cdot \sigma_{(r+1..|\sigma|)} \models skip$. The former implies $r \geq k + 1$ (since q is non-local). This and the latter mean that $r = |\sigma|$. Hence $(\sigma, 0, k, |\sigma|) \models q$.

The proof of PRJskip3:

Suppose $(\sigma, 0, k, |\sigma|) \models (p, q)\ prj\ skip$. We first observe that $\sigma\downarrow(k) \models skip$ and $\sigma \models p; q$ is impossible. Thus one of the following must hold:

- There is r, $k \leq r \preceq |\sigma|$, such that $(\sigma, 0, k, r) \models p$, $\sigma\downarrow(k, r) \models skip$ and $(\sigma, r, r, |\sigma|) \models q$. Hence $(\sigma, 0, k, |\sigma|) \models p; q$.

- **There are integers r_1, r_2 such that $k \le r_1 \le r_2 \le |\sigma|$, $(\sigma, 0, k, r_1) \models p$, $(\sigma, r_1, r_1, r_2) \models q$ and $\sigma\downarrow(k, r_1, r_2)\cdot\sigma_{(r_2+1..|\sigma|)} \models$ *skip*. Since at least one of p and q is non-local, we must have $r_1 \ge k+1$ or $r_2 \ge r_1+1$.** Thus, from $\sigma\downarrow(k, r_1, r_2)\cdot\sigma_{(r_2+1..|\sigma|)} \models$ ***skip*** it follows that $|\sigma| = r_2$. **Hence $(\sigma, 0, k, |\sigma|) \models p; q$.**

□

Theorem 5.5 Let $p, q, p_1, \ldots, p_m$ be formulas.

$$\begin{array}{ll} PRJor1 & (p_1, \ldots, (p_i \vee p_i'), \ldots, p_m)\ prj\ q \\ & \equiv (p_1, \ldots, p_i, \ldots, p_m)\ prj\ q \vee (p_1, \ldots, p_i', \ldots, p_m)\ prj\ q \\ PRJor2 & (p_1, \ldots, p_m)\ prj\ (p \vee q) \\ & \equiv (p_1, \ldots p_m)\ prj\ p \vee (p_1, \ldots, p_m)\ prj\ q. \end{array}$$

Proof

The proof of PRJor1 and PRJor2 follow directly from the definition of the semantics of the projection operator and FD9

$$q_1; \ldots; (q_j \vee q_j'); \ldots; q_s \equiv (q_1; \ldots; q_j; \ldots; q_s) \vee (q_1; \ldots; q_j'; \ldots; q_s).$$

□

Theorem 5.6 Let p, q be formulas.

$$PRJpar \quad p \parallel q \equiv p \wedge ((q, true)\ prj\ empty) \vee q \wedge ((p, true)\ prj\ empty)$$

Proof

The proof of PRJpar follows from PRJempty3.

□

Theorem 5.7 Let $p_1, \ldots, p_m$ be lec-formulas.

$$\begin{array}{llcl} PRJnext1 & p\ prj\ \bigcirc q & \equiv & (p \wedge more; q) \vee (p \wedge empty; \bigcirc q) \\ PRJnext2 & p_1 \wedge more; (p_2, \ldots, p_m)\ prj\ q & \supset & (p_1, p_2, \ldots, p_m)\ prj\ \bigcirc q \\ PRJnext3 & (p_1, \ldots, p_m)\ prj\ \bigcirc q & \equiv & p_1 \wedge more; (p_2, \ldots, p_m)\ prj\ q \\ & & & \vee p_1 \wedge empty; (p_2, \ldots, p_m)\ prj\ \bigcirc q \\ PRJnext4 & (\bigcirc p\ prj\ \bigcirc q) & \equiv & \bigcirc(p; q) \\ PRJnext5 & (\bigcirc p_1, \ldots, p_m)\ prj\ \bigcirc q & \equiv & \bigcirc(p_1; (p_2, \ldots, p_m)\ prj\ q) \end{array}$$

Proof

We prove this theorem using I-prj2. Let σ be an interval and k an integer, $0 \le k \preceq |\sigma|$ and $m \ge 2$.

The proof of PRJnext1

$$\begin{array}{ll} & (\sigma, 0, k, |\sigma|) \models p\ prj\ \bigcirc q \\ \Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models [p](r_1) \text{ (i.e. } (\sigma, 0, k, r_1) \models p) \\ & \text{and} \\ & r_1 = |\sigma| \text{ and } \sigma\downarrow(k, r_1) \models \bigcirc q \text{ or} \\ & r_1 < |\sigma| \text{ and } \sigma\downarrow(k, r_1)\cdot\sigma_{(r_1+1..|\sigma|)} \models \bigcirc q \end{array}$$

In the case of $r_1 = |\sigma|$, if $|\sigma| = \omega$ or $r_1 = k$ then $\sigma\downarrow(k, r_1) = \sigma\downarrow(k) \not\models \bigcirc q$. Hence $k < r_1 = |\sigma| < \omega$. $\sigma\downarrow(k, r_1) \models \bigcirc q$ implies $(\sigma, 0, |\sigma|, |\sigma|) \models q$. Moreover, $(\sigma, 0, k, |\sigma|) \models p \wedge more$, hence $(\sigma, 0, k, |\sigma|) \models p \wedge more; q$.

In the case of $r_1 < |\sigma|$, if $r_1 = k$ then $\sigma\downarrow(k, r_1)\cdot\sigma_{(r_1+1..|\sigma|)} = \sigma^{(k)} \models \bigcirc q$. This amounts to $(\sigma, k, k, |\sigma|) \models \bigcirc q$. Since $(\sigma, 0, k, r_1) = (\sigma, 0, k, k) \models p \wedge empty$, $(\sigma, 0, k, |\sigma|) \models p \wedge empty; \bigcirc q$. If $r_1 > k$, $\sigma\downarrow(k, r_1).\sigma_{(r_1+1..|\sigma|)} = \sigma^{(k)} \models \bigcirc q$ iff $\sigma^{(r_1)} \models q$ iff $(\sigma, r_1, r_1, |\sigma|) \models q$. Since $(\sigma, 0, k, r_1) \models p \wedge more$, $(\sigma, 0, k, |\sigma|) \models p \wedge more; q$.

Conversely, suppose $(\sigma, 0, k, |\sigma|) \models (p \wedge empty; \bigcirc q) \vee (p \wedge more; q)$. If $(\sigma, 0, k, |\sigma|) \models p \wedge empty; \bigcirc q$ then there exists r, $k \leq r \preceq |\sigma|$ such that $(\sigma, 0, k, r) \models p \wedge empty$ and $(\sigma, r, r, |\sigma|) \models \bigcirc q$. It is obvious that $r < |\sigma|$ otherwise $(\sigma, r, r, |\sigma|) \not\models \bigcirc q$. Thus, $(\sigma, 0, k, |\sigma|) \models [p](r)$ and $r = k$ and $(\sigma, r, r, |\sigma|) \models \sigma\downarrow(k, r)\cdot\sigma_{(r+1..|\sigma|)} \models \bigcirc q$. Hence $(\sigma, 0, k, |\sigma|) \models p\ prj\ \bigcirc q$.

If $(\sigma, 0, k, |\sigma|) \models p \wedge more; q$, then there is r, $k \leq r \leq |\sigma|$ such that $(\sigma, 0, k, r) \models p \wedge more$ and $(\sigma, r, r, |\sigma|) \models q$. Thus, $(\sigma, 0, k, |\sigma|) \models [p](r)$ and $k \leq r = |\sigma|$ and $\sigma\downarrow(k, r) \models \bigcirc q$ (since $< s_r > = < s_{|\sigma|} > \models q$). Or $r < |\sigma|$ and $\sigma\downarrow(k, r)\cdot\sigma_{(r+1..|\sigma|)} = < s_k, s_r > \cdot\sigma^{(r+1)} = < s_k, s_r > \circ\sigma^{(r)} \models \bigcirc q$ (since $\sigma^{(r)} \models q$).

Therefore, $(\sigma, 0, k, |\sigma|) \models p\ prj\ \bigcirc q$.

The proof of PRJnext2

If $(\sigma, 0, k, |\sigma|) \models p_1 \wedge more; (p_2, ..., p_m)\ prj\ q$, then there exists an integer $r_1, k \leq r_1 \preceq |\sigma|$ such that $(\sigma, 0, k, r_1) \models p_1 \wedge more$ and $(\sigma, r_1, r_1, |\sigma|) \models (p_2, ..., p_m)\ prj\ q$. By the former, we know that $(\sigma, 0, k, r_1) \models p_1$ and $k < r_1$. By the latter, we obtain

$(\sigma, r_1, r_1, |\sigma|) \models [p_2, ..., p_m](r_2, ..., r_m)$ for $r_1 \leq r_2 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma|$
and
$r_m = |\sigma|$ and $\sigma\downarrow(r_1, ..., r_h) \models q$ for some $1 \leq h \leq m$ or
$r_m < |\sigma|$ and $\sigma\downarrow(r_1, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models q$

Therefore,

$(\sigma, 0, k, |\sigma|) \models [p_1, ..., p_m](r_1, ..., r_m)$ for $k < r_1 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma|$
and
$r_m = |\sigma|$ and $\sigma\downarrow(k, r_1, ..., r_h) = \sigma\downarrow(k, r_1) \circ \sigma\downarrow(r_1, ..., r_h) \models \bigcirc q$ for some $1 \leq h \leq m$ or
$r_m < |\sigma|$ and $\sigma\downarrow(k, r_1, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} = \sigma\downarrow(k, r_1) \circ \sigma\downarrow(r_1, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models \bigcirc q$

Therefore, $(\sigma, 0, k, |\sigma|) \models (p_1, ..., p_m)\ prj\ \bigcirc q$.

The proof of PRJnext3

Suppose $(\sigma, 0, k, |\sigma|) \models (p_1, ..., p_m)\ prj\ \bigcirc q$. We have

$(\sigma, 0, k, |\sigma|) \models [p_1, ..., p_m](r_1, ..., r_m)$ for $k = r_0 \leq r_1 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma|$
and
$r_m = |\sigma|$ and $\sigma\downarrow(r_0, r_1, ..., r_h) \models \bigcirc q$ for some $0 \leq h \leq m$ or
$r_m < |\sigma|$ and $\sigma\downarrow(r_0, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models \bigcirc q$

Thus, two cases need considering:

1) $r_1 = k$.

In this case, $\sigma \downarrow (k, r_1, ..., r_h) = \sigma\downarrow(r_1, ..., r_h)$ (Lemma 5.2). Hence, the following hold:

$(\sigma, 0, k, r_1) \models p_1$ and $(\sigma, r_1, r_1, |\sigma|) \models [p_2, ..., p_m](r_2, ..., r_m)$ for $r_1 \leq r_2 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma|$
and
$r_m = |\sigma|$ and $\sigma\downarrow(r_0, r_1, ..., r_h) = \sigma\downarrow(r_1, ..., r_h) \models \bigcirc q$ for some $0 \leq h \leq m$ or
$r_m < |\sigma|$ and $\sigma\downarrow(r_0, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} = \sigma\downarrow(r_1, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models \bigcirc q$

Thus, $(\sigma, 0, k, r_1) \models p \wedge empty$ and $(\sigma, r_1, r_1, |\sigma|) \models (p_2, ..., p_m)\ prj\ \bigcirc q$ leading to $(\sigma, 0, k, |\sigma|) \models p \wedge empty; (p_2, ..., p_m)\ prj\ \bigcirc q$.

2) $k < r_1$.

In this case, $\sigma \downarrow (k, r_1, ..., r_h) =< s_k, s_{r_1} > \circ\sigma\downarrow(r_1, ..., r_h)$ (Lemma 5.2). Hence, the following hold:

$(\sigma, 0, k, r_1) \models p_1 \wedge more$ and $(\sigma, r_1, r_1, |\sigma|) \models [p_2, ..., p_m](r_2, ..., r_m)$
for $k = r_0 < r_1 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma|$
and
$r_m = |\sigma|$ and $\sigma\downarrow(r_0, r_1, ..., r_h) =< s_k, s_{r_1} > \circ\sigma\downarrow(r_1, ..., r_h) \models \bigcirc q$ leading to $(\sigma\downarrow(r_1, ..., r_m) \models q$
or
$r_m < |\sigma|$ and $\sigma\downarrow(r_0, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} =< s_k, s_{r_1} > \circ\sigma\downarrow(r_1, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models \bigcirc q$
leading to $\sigma\downarrow(r_1, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models q$

Thus, we obtain $(\sigma, 0, k, r_1) \models p_1 \wedge more$ and $(\sigma, r_1, r_1, |\sigma|) \models (p_2, ..., p_m)\ prj\ q$. Hence, $(\sigma, 0, k, |\sigma|) \models p_1 \wedge more; (p_2, ..., p_m)\ prj\ q$.

Conversely, suppose $(\sigma, 0, k, |\sigma|) \models (p_1 \wedge more; (p_2; ...; p_m)\ prj\ q) \vee (p_1 \wedge empty; (p_2, ..., p_m)\ prj\ \bigcirc q)$. By PRJempty5 and PRJnext2, $(\sigma, 0, k, |\sigma|) \models (p_1, ..., p_m)\ prj\ \bigcirc q$.

The proof of PRJnext4:

$$\begin{array}{rcll} \bigcirc p\ prj\ \bigcirc q & \equiv & \bigcirc p \wedge more; q & \text{PRJnext3} \\ & \equiv & \bigcirc p; q & \text{FS4} \\ & \equiv & \bigcirc(p; q) & \text{FCH1} \end{array}$$

The proof of PRJnext5:

$$\begin{array}{rcll} (\bigcirc p_1, ... p_m)\ prj\ \bigcirc q & \equiv & \bigcirc p_1 \wedge more; (p_2, ..., p_m)\ prj\ q & \text{PRJnext3} \\ & \equiv & \bigcirc p_1; (p_2, ..., p_m)\ prj\ q & \text{FS4} \\ & \equiv & \bigcirc(p; (p_2, ..., p_m)\ prj\ q) & \text{FCH1} \end{array}$$

□

From Theorem 5.7, a useful conclusion can be derived. We formalize it in Corollary 5.8.

Corollary 5.8 Let $p_1, ..., p_m$ be lec-formulas and q a non-past formula.

$$\begin{array}{lrl} PRJnext6 & & (p_1, ... p_m)\ prj\ \bigcirc q \\ & \equiv & (p_1 \wedge more; (p_2, ..., p_m)\ prj\ q) \\ & & \vee \bigvee_{t=1}^{m-2} ((p_1 \wedge ... \wedge p_t) \wedge empty; p_{t+1} \wedge more; (p_{t+2}, ..., p_m)\ prj\ q) \\ & & \vee((p_1 \wedge ... \wedge p_{m-1}) \wedge empty; p_m \wedge more; q) \\ & & \vee((p_1 \wedge ... \wedge p_m) \wedge empty; \bigcirc q) \end{array}$$

Proof

$$
\begin{array}{lll}
 & (p_1,...p_m)\ prj\ \bigcirc q & \\
\equiv & (p_1 \wedge more; (p_2,...,p_m)\ prj\ q) & \\
 & \vee(p_1 \wedge empty; (p_2,...,p_m)\ prj\ \bigcirc q) & \text{PRJnext3} \\
\equiv & (p_1 \wedge more; (p_2,...,p_m)\ prj\ q) & \\
 & \vee(p_1 \wedge empty; p_2 \wedge more; (p_3,...,p_m)\ prj\ q) & \\
 & \vee(p_1 \wedge empty; p_2 \wedge empty; (p_3,...,p_m)\ prj\ \bigcirc q) & \text{PRJnext3} \\
\equiv & (p_1 \wedge more; (p_2,...,p_m)\ prj\ q) & \\
 & \vee(p_1 \wedge empty; p_2 \wedge more; (p_3,...,p_m)\ prj\ q) & \\
 & \vee((p_1 \wedge p_2) \wedge empty; p_3 \wedge more; (p_4,...,p_m)\ prj\ q) & \\
 & \vee((p_1 \wedge p_2) \wedge empty; p_3 \wedge empty; (p_4,...,p_m)\ prj\ \bigcirc q) & \text{EMP4, PRJnext3} \\
\equiv & ... & \\
\equiv & p_1 \wedge more; (p_2,...,p_m)\ prj\ q & \\
 & \vee p_1 \wedge empty; p_2 \wedge more; (p_3,...,p_m)\ prj\ q & \\
 & \vee(p_1 \wedge p_2) \wedge empty; p_3 \wedge more; (p_4,...,p_m)\ prj\ q & \\
 & \vee ... & \\
 & \vee(p_1 \wedge ... \wedge p_{m-1}) \wedge empty; p_m \wedge more; q & \\
 & \vee(p_1 \wedge ... \wedge p_{m-1}) \wedge empty; p_m \wedge empty; \bigcirc q & \text{EMP4, PRJnext1, PRJnext3} \\
\equiv & (p_1 \wedge more; (p_2,...,p_m)\ prj\ q) & \\
 & \vee((p_1 \wedge ... \wedge p_{m-1}) \wedge empty; p_m \wedge more; q) & \\
 & \bigvee_{t=1}^{m-2} ((p_1 \wedge ... \wedge p_t) \wedge empty; p_{t+1} \wedge more; (p_{t+2},...,p_m)\ prj\ q) & \\
 & \vee((p_1 \wedge ... \wedge p_m) \wedge empty; \bigcirc q) &
\end{array}
$$

□

PRJnext6 can be written in a more concise form, as follows:

$$
\begin{array}{lll}
PRJnext6 & & (p_1,...,p_m)\ prj\ \bigcirc q \\
 & \equiv & \bigvee_{t=0}^{m-1} ((p_0 \wedge ... \wedge p_t) \wedge empty; p_{t+1} \wedge more; (p_{t+2},...,p_{m+1})\ prj\ q) \\
 & & \vee((p_0 \wedge ... \wedge p_m); \bigcirc q)
\end{array}
$$

where $p_0 \equiv p_{m+1} \equiv empty$.

Since the disjunction in the above formula is mutually exclusive, PRJnext6 can also be written in another form shown below:

$$
\begin{array}{lll}
PRJnext6 & & (p_1,...,p_m)\ prj\ \bigcirc q \\
 & \equiv & \exists t : 0 \le t \le m-1 \wedge ((p_0 \wedge ... \wedge p_t) \wedge empty; p_{t+1} \wedge more; (p_{t+2},...,p_{m+1})\ prj\ q) \\
 & & \vee((p_0 \wedge ... \wedge p_m); \bigcirc q)
\end{array}
$$

These concise forms of PRJnext6 are frequently used in the reduction of programs.

Theorem 5.9 Let $p_1,...,p_m,q$ be formulas.

$$
\begin{array}{ll}
PRJand1 & p\ prj\ q \equiv p \wedge q \\
 & \text{if } p \text{ or } q \text{ is a state formula.} \\
PRJand2 & (w \wedge p_1,...,p_m)\ prj\ q \equiv w \wedge (p_1,...,p_m)\ prj\ q \\
 & \text{if } w \text{ is a state formula.} \\
PRJand3 & (p_1,...,p_m)\ prj\ (w \wedge q) \equiv w \wedge (p_1,...,p_m)\ prj\ q \\
 & \text{if } w \text{ is a state formula.}
\end{array}
$$

Proof

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of PRJand1:

Suppose $(\sigma, 0, k, |\sigma|) \models p\ prj\ q$. If $\sigma{\downarrow}(k) \models q$ and $(\sigma, 0, k, |\sigma|) \models p$ then $(\sigma, 0, k, |\sigma|) \models p \wedge q$. Otherwise, there is r, $k \leq r \preceq |\sigma|$, such that $(\sigma, 0, k, r) \models p$ and $\sigma{\downarrow}(k, r)\cdot\sigma_{(r+1..|\sigma|)} = \sigma_{(k..|\sigma|)} \models q$. If p is a state formula, then $(\sigma, 0, k, r) \models p$ iff $(\sigma, 0, k, |\sigma|) \models p$. Hence $(\sigma, 0, k, |\sigma|) \models p \wedge q$.

Conversely, if $(\sigma, 0, k, |\sigma|) \models p \wedge q$ then $(\sigma, 0, k, |\sigma|) \models p$ and $(\sigma, 0, k, |\sigma|) \models q$. Furthermore, if p is a state formula, by taking $r_1 = k$, one can show that $(\sigma, 0, k, r_1) \models p$ and $\sigma{\downarrow}(0, r_1)\cdot\sigma_{(r_1+1..|\sigma|)} \models q$. Hence, $(\sigma, 0, k, |\sigma|) \models p\ prj\ q$. If q is a state formula, one can show that $\sigma{\downarrow}(k) \models q$. Hence $(\sigma, 0, k, |\sigma|) \models p\ prj\ q$.

The proof of PRJand3:

We prove this using I-prj2. Since w is a state formula, $\sigma{\downarrow}(k, r_1, ..., r_l) \models w$ iff $(\sigma, 0, k, |\sigma|) \models w$. Thus

$$\begin{array}{ll}
 & (\sigma, 0, k, |\sigma|) \models (p_1, ..., p_m)\ prj\ (w \wedge q) \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models [p_1, ..., p_m](r_1, ..., r_m) \text{ for } k = r_0 \leq r_1 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma| \\
 & \text{and} \\
 & r_m = |\sigma| \text{ and } \sigma{\downarrow}(r_0, r_1, ..., r_h) \models w \wedge q \text{ for some } 0 \leq h \leq m \text{ or} \\
 & r_m < |\sigma| \text{ and } \sigma{\downarrow}(r_0, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models w \wedge q \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models [p_1, ..., p_m](r_1, ..., r_m) \text{ for } k = r_0 \leq r_1 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma| \\
 & \text{and} \\
 & r_m = |\sigma| \text{ and } (\sigma, 0, k, |\sigma|) \models w \text{ and } \sigma{\downarrow}(r_0, r_1, ..., r_h) \models q \text{ for some } 0 \leq h \leq m \text{ or} \\
 & r_m < |\sigma| \text{ and } (\sigma, 0, k, |\sigma|) \models w \text{ and } \sigma{\downarrow}(r_0, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models q \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models w \text{ and} \\
 & (\sigma, 0, k, |\sigma|) \models [p_1, ..., p_m](r_1, ..., r_m) \text{ for } k = r_0 \leq r_1 \leq ... \leq r_{m-1} \preceq r_m \leq |\sigma| \\
 & \text{and} \\
 & r_m = |\sigma| \text{ and } \sigma{\downarrow}(r_0, r_1, ..., r_h) \models q \text{ for some } 0 \leq h \leq m \text{ or} \\
 & r_m < |\sigma| \text{ and } \sigma{\downarrow}(r_0, ..., r_m)\cdot\sigma_{(r_m+1..|\sigma|)} \models q \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models w \text{ and} \\
 & (\sigma, 0, k, |\sigma|) \models (p_1, ..., p_m)\ prj\ q \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models w \wedge ((p_1, ..., p_m)\ prj\ q)
\end{array}$$

The proof of PRJand2 is similar to PRJand3.

□

5.3 Normal Form of Projection Construct

The projection is a new construct for EITL. It can be also a statement for programming language as long as $p_1, ..., p_m, q$ are all programs (of course, lec-formulas). In this section, we prove that a program involving the projection construct can also be reduced to the normal form given in Chapter 4.

Theorem 5.10 If $p_1, ..., p_m, q$ are programs, and q is a non-past formula, then there is a program p as in (4.1) (Definition 4.2) such that

$$p \equiv (p_1, ..., p_m)\ prj\ q$$

Proof

Suppose $q \equiv \bigvee_{r=1}^{d} q_{er} \wedge empty \vee \bigvee_{i=1}^{h} q_{ci} \wedge \bigcirc q_{fi}$ and, for all $1 \leq l \leq m$, $p_l \equiv \bigvee_{r=1}^{d_r} p_{ler} \wedge empty \vee \bigvee_{j=1}^{k_l} p_{lcj} \wedge \bigcirc p_{lfj}$.

$$\begin{aligned}
(p_1,...,p_m)\ prj\ q &\equiv (p_1,...,p_m)\ prj\ (\bigvee_{r=1}^{t} q_{er} \wedge empty \vee \bigvee_{i=1}^{h} q_{ci} \wedge \bigcirc q_{fi}) && \text{hypothesis}\\
&\equiv (p_1,...,p_m)\ prj\ (q_{e1} \wedge empty)\\
&\quad \vee \bigvee_{i=1}^{h} (p_1,...,p_m)\ prj\ (q_{ci} \wedge \bigcirc q_{fi}) && \text{PRJor2}
\end{aligned}$$

Further,

$$\begin{aligned}
(p_1,...,p_m)\ prj\ (q_{e1} \wedge empty) &\equiv q_{e1} \wedge (p_1,...,p_m)\ prj\ empty && \text{PRJand3}\\
&\equiv q_{e1} \wedge (p_1;...;p_m) && \text{PRJempty3}
\end{aligned}$$

By Theorem 4.9, $(p_1;...;p_m)$ can be reduced to a normal form and so can $q_{e1} \wedge (p_1;...;p_m)$. Hence, there is a program p such that the following holds

$$q_{e1} \wedge (p_1;...;p_m) \quad \equiv \quad \bigvee_{r=1}^{d} p_{er} \wedge empty \vee \bigvee_{j=1}^{n} p_{cj} \wedge \bigcirc p_{fj} \qquad (1)$$

On the other hand,

$$\begin{aligned}
& (p_1,...,p_m)\ prj\ (q_{ci} \wedge \bigcirc q_{fi})\\
\equiv\ & q_{ci} \wedge ((p_1,...,p_m)\ prj\ \bigcirc q_{fi}) && \text{PRJand3}\\
\equiv\ & q_{ci} \wedge (p_1 \wedge p_2 \wedge ... \wedge p_m \wedge empty; \bigcirc q_{fi})\\
& \vee q_{ci} \wedge \bigvee_{t=0}^{m-1} (p_0 \wedge ... \wedge p_t \wedge empty; p_{t+1} \wedge more; (p_{t+2},...,p_{m+1})\ prj\ q_{fi}) && \text{PRJnext6}\\
\equiv\ & q_{ci} \wedge (p_{1e1} \wedge ... \wedge p_{me1} \wedge empty; \bigcirc q_{fi})\\
& \vee q_{ci} \wedge \bigvee_{t=0}^{m-1} (p_{0e1} \wedge p_{1e1} \wedge p_{2e1} \wedge ... \wedge p_{te1} \wedge empty;\\
& \bigvee_{j=1}^{k_{t+1}} p_{t+1cj} \wedge \bigcirc p_{t+1fj} \wedge more; (p_{t+2},...,p_{m+1})\ prj\ q_{fi}) && p_{0e1} \equiv empty\\
& && \text{hypothesis}\\
\equiv\ & q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{me1} \wedge (empty; \bigcirc q_f)\\
& \vee q_{ci} \wedge \bigvee_{t=0}^{m-1} (p_{0e1} \wedge p_{1e1} \wedge ... \wedge p_{te1} \wedge empty; \bigvee_{j=1}^{k_{t+1}} p_{t+1cj} \wedge \bigcirc p_{t+1fj}; (p_{t+2},...,p_{m+1})\ prj\ q_{fi})\\
& && \text{FCH2, FS4}\\
\equiv\ & q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{me1} \wedge \bigcirc q_{fi}\\
& \vee \bigvee_{1 \leq j \leq k_1} (q_{ci} \wedge p_{1cj} \wedge \bigcirc p_{1fj}; (p_2,...,p_m)\ prj\ q_{fi})\\
& \vee \bigvee_{1 \leq t \leq m-1, 1 \leq j \leq k_{t+1}} (q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{te1} \wedge p_{t+1cj} \wedge \bigcirc p_{t+1fj}; (p_{t+2},...,p_{m+1})\ prj\ q_{fi})\\
& && \text{FD9, FEP5}\\
\equiv\ & q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{me1} \wedge \bigcirc q_{fi}\\
& \vee \bigvee_{1 \leq j \leq k_1} q_{ci} \wedge p_{1cj} \wedge \bigcirc(p_{1fj}; (p_2,...,p_m)\ prj\ q_{fi})\\
& \vee \bigvee_{1 \leq t \leq m-1, 1 \leq j \leq k_{t+1}} (q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{te1} \wedge p_{t+1cj} \wedge \bigcirc(p_{t+1fj}; (p_{t+2},...,p_{m+1})\ prj\ q_{fi})\\
& && \text{FCH1}
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\bigvee_{i=1}^{h} (p_1, ..., p_m)\ prj\ (q_{ci} \wedge \bigcirc q_{fi}) \\
\equiv\ &\bigvee_{i=1}^{h} q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{me1} \wedge \bigcirc q_{fi} \\
\vee\ &\bigvee_{1 \le j \le k_1, 1 \le i \le h} q_{ci} \wedge p_{1cj} \wedge \bigcirc(p_{1fj}; (p_2, ..., p_m)\ prj\ q_{fi}) \\
\vee\ &\bigvee_{1 \le t \le m-1, 1 \le j \le k_{t+1}, 1 \le i \le h} (q_{ci} \wedge p_{1e1} \wedge ... \wedge p_{te1} \wedge p_{t+1cj} \wedge \bigcirc(p_{t+1fj}; (p_{t+2}, ..., p_{m+1})\ prj\ q_{fi}) \qquad (2)
\end{aligned}$$

From (1), (2), it follows that $(p_1, ..., p_m)\ prj\ q$ is reduced to its normal form. □

Note that, if $(p_1, ..., p_m)\ prj\ q$ is a deterministic program, then its normal form is of a simpler form. That is, in an internal state, it is reduced to:

$$(p_1, ..., p_m)\ prj\ q \equiv p_c \wedge \bigcirc p_f$$

and at the final state, it is reduced to:

$$(p_1, ..., p_m)\ prj\ q \equiv p_e \wedge empty$$

where p_e, p_c and p_f are defined as in (4.1) (Definition 4.2).

Example 5.2 Programs p_1, p_2, p_3 and q are defined as follows:

$$p_1 \stackrel{\text{def}}{=} len(2) \wedge keep(i\ o{=}i + 2)$$

$$p_2 \stackrel{\text{def}}{=} len(4) \wedge keep(i\ o{=}i + 3)$$

$$p_3 \stackrel{\text{def}}{=} len(6) \wedge keep(i\ o{=}i + 4)$$

$$q \stackrel{\text{def}}{=} len(4) \wedge (i = 2) \wedge (j = 0) \wedge keep(j\ o{=}j + i).$$

We reduce $(p_1, p_2, p_3)\ prj\ q$. Note that, in the reduction, we use Theorem 3.7 for substitution without declaration.

Since

$$\begin{aligned}
q &\equiv len(4) \wedge i = 2 \wedge j = 0 \wedge \Box(more \rightarrow (\bigcirc j = j + i)) && \text{definition} \\
&\equiv \bigcirc len(3) \wedge i = 2 \wedge j = 0 \wedge more \rightarrow (\bigcirc j = j + i) \wedge \odot \Box(more \rightarrow (\bigcirc j = j + i)) && \text{FE2} \\
&\equiv \bigcirc len(3) \wedge i = 2 \wedge j = 0 \wedge \bigcirc j = 2 + 0 \wedge \bigcirc \Box(more \rightarrow (\bigcirc j = j + i)) && \text{FS4, FW1} \\
&\equiv i = 2 \wedge j = 0 \wedge \bigcirc(j = 2 \wedge len(3) \wedge \Box(more \rightarrow (\bigcirc j = j + i))) && \text{FD3}
\end{aligned}$$

and

$$\begin{aligned}
p_1 &\equiv len(2) \wedge \Box(more \rightarrow (\bigcirc i = i + 2)) && \text{definition} \\
&\equiv \bigcirc len(1) \wedge (more \rightarrow \bigcirc i = i + 2) \wedge \odot \Box(more \rightarrow (\bigcirc i = i + 2)) && \text{FE2} \\
&\equiv \bigcirc len(1) \wedge \bigcirc i = 2 + 2 \wedge \bigcirc \Box(more \rightarrow (\bigcirc i = i + 2)) && \text{FS4, FW1} \\
&\equiv \bigcirc(i = 4 \wedge len(1) \wedge \Box(more \rightarrow (\bigcirc i = i + 2))) && \text{FD3}
\end{aligned}$$

Thus, $(p_1,p_2,p_3)\ prj\ q$ can be reduced as follows:

$$\begin{array}{lll}
& (p_1,p_2,p_3)\ prj\ q & \\
\equiv & (\bigcirc(i=4\wedge len(1)\wedge\Box(more\rightarrow(\bigcirc i=i+2))),p_2,p_3)\ prj & \\
& (i=2\wedge j=0\wedge\bigcirc(j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))) & \text{definition}\\
\equiv & i=2\wedge j=0\wedge(\bigcirc(i=4\wedge len(1)\wedge\Box(more\rightarrow(\bigcirc i=i+2))),p_2,p_3)\ prj & \\
& (\bigcirc(j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))) & \text{PRJand3}\\
\equiv & i=2\wedge j=0\wedge\bigcirc(i=4\wedge len(1)\wedge\Box(more\rightarrow(\bigcirc i=i+2));(p_2,p_3)\ prj & \\
& (j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))) & \text{PRJnext5}
\end{array}$$

Hence,

$$\begin{array}{lll}
P_c^0 & \equiv & i=2\wedge j=0\\
P_f^0 & \equiv & i=4\wedge len(1)\wedge\Box(more\rightarrow(\bigcirc i=i+2));(p_2,p_3)\ prj\\
& & (j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))
\end{array}$$

Now we turn our attention to reducing the chop construct. Since

$$\begin{array}{ll}
& i=4\wedge len(1)\wedge\Box(more\rightarrow(\bigcirc i=i+2))\\
\equiv & i=4\wedge\bigcirc empty\wedge more\rightarrow(\bigcirc i=i+2)\wedge\odot\Box(more\rightarrow(\bigcirc i=i+2))\\
\equiv & i=4\wedge\bigcirc empty\wedge\bigcirc i=4+2\wedge\bigcirc\Box(more\rightarrow(\bigcirc i=i+2))\\
\equiv & i=4\wedge\bigcirc(empty\wedge i=6\wedge\Box(more\rightarrow(\bigcirc i=i+2)))
\end{array}$$

By Theorem 5.10, it follows that

$$\begin{array}{lll}
P_c^1 & \equiv & i=4\\
P_f^1 & \equiv & empty\wedge i=6\wedge\Box(more\rightarrow(\bigcirc i=i+2));\\
& & (p_2,p_3)\ prj\ (j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))\\
& \equiv & empty\wedge i=6;\\
& & (p_2,p_3)\ prj\ (j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))\\
& \equiv & i=6\wedge(p_2,p_3)\ prj\ (j=2\wedge len(3)\wedge\Box(more\rightarrow\bigcirc j=j+i))\\
& \equiv & i=6\wedge(len(4)\wedge\Box(more\rightarrow\bigcirc i=i+3),p_3)\ prj\\
& & (j=2\wedge\bigcirc len(2)\wedge more\rightarrow\bigcirc j=j+i\wedge\odot\Box(more\rightarrow\bigcirc j=j+i))\\
& \equiv & i=6\wedge j=2\wedge(\bigcirc len(3)\wedge\bigcirc i=6+3\wedge\bigcirc\Box(more\rightarrow\bigcirc i=i+3),p_3)\ prj\\
& & (\bigcirc len(2)\wedge\bigcirc j=2+6\wedge\bigcirc\Box(more\rightarrow\bigcirc j=j+i))\\
& \equiv & i=6\wedge j=2\wedge(\bigcirc(len(3)\wedge i=9\wedge\Box(more\rightarrow\bigcirc i=i+3)),p_3)\ prj\\
& & \bigcirc(len(2)\wedge j=8\wedge\Box(more\rightarrow\bigcirc j=j+i))\\
& \equiv & i=6\wedge j=2\wedge\bigcirc(len(3)\wedge i=9\wedge\Box(more\rightarrow\bigcirc i=i+3);p_3\ prj\\
& & (len(2)\wedge j=8\wedge\Box(more\rightarrow\bigcirc j=j+i))
\end{array}$$

We achieve

$$\begin{array}{lll}
P_c^2 & \equiv & i=6\wedge j=2\\
P_f^2 & \equiv & len(3)\wedge i=9\wedge\Box(more\rightarrow\bigcirc i=i+3);(p_3\ prj\\
& & (len(2)\wedge j=8\wedge\Box(more\rightarrow\bigcirc j=j+i)))
\end{array}$$

In a similar way, we can eventually obtain the following:

$$\begin{array}{lll}
P_c^3 & \equiv & i=9\\
P_f^3 & \equiv & len(2)\wedge i=12\wedge\Box(more\rightarrow\bigcirc i=i+3);p_3\ prj\\
& & (len(2)\wedge j=8\wedge\Box(more\rightarrow\bigcirc j=j+i))
\end{array}$$

$$\begin{aligned}
P_c^4 &\equiv i = 12 \\
P_f^4 &\equiv \mathit{len}(1) \wedge i = 15 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 3); p_3\ \mathit{prj} \\
&\quad (\mathit{len}(2) \wedge j = 8 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^5 &\equiv i = 15 \\
P_f^5 &\equiv \mathit{empty} \wedge i = 18 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 3); p_3\ \mathit{prj} \\
&\quad (\mathit{len}(2) \wedge j = 8 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i)) \\
&\equiv i = 18 \wedge (\mathit{len}(6) \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4)\ \mathit{prj} \\
&\quad (\mathit{len}(2) \wedge j = 8 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))) \\
&\equiv i = 18 \wedge j = 8 \wedge (\bigcirc \mathit{len}(5) \wedge i = 22 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4)\ \mathit{prj} \\
&\quad (\bigcirc \mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))) \\
&\equiv i = 18 \wedge j = 8 \wedge \bigcirc(\mathit{len}(5) \wedge i = 22 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i)))
\end{aligned}$$

$$\begin{aligned}
P_c^6 &\equiv i = 18 \wedge j = 8 \\
P_f^6 &\equiv \mathit{len}(5) \wedge i = 22 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^7 &\equiv i = 22 \\
P_f^7 &\equiv \mathit{len}(4) \wedge i = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^8 &\equiv i = 26 \\
P_f^8 &\equiv \mathit{len}(3) \wedge i = 30 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^9 &\equiv i = 30 \\
P_f^9 &\equiv \mathit{len}(2) \wedge i = 34 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^{10} &\equiv i = 34 \\
P_f^{10} &\equiv \mathit{len}(1) \wedge i = 38 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^{11} &\equiv i = 38 \\
P_f^{11} &\equiv \mathit{empty} \wedge i = 42 \wedge \Box(\mathit{more} \rightarrow \bigcirc i = i + 4); \\
&\quad (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i)) \\
&\equiv i = 42 \wedge (\mathit{len}(1) \wedge j = 26 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i)) \\
&\equiv i = 42 \wedge j = 26 \wedge (\bigcirc \mathit{empty} \wedge \bigcirc j = 42 + 26 \wedge \bigcirc\Box(\mathit{more} \rightarrow \bigcirc j = j + i)) \\
&\equiv i = 42 \wedge j = 26 \wedge \bigcirc(\mathit{empty} \wedge j = 68 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i))
\end{aligned}$$

$$\begin{aligned}
P_c^{12} &\equiv i = 42 \wedge j = 26 \\
P_f^{12} &\equiv \mathit{empty} \wedge j = 68 \wedge \Box(\mathit{more} \rightarrow \bigcirc j = j + i) \\
&\equiv j = 68 \wedge \mathit{empty} \wedge (\mathit{more} \rightarrow \bigcirc j = j + i) \wedge \odot \Box(\mathit{more} \rightarrow \bigcirc j = j + i) \\
&\equiv j = 68 \wedge \mathit{empty}
\end{aligned}$$

$$P_e \equiv j = 68 \wedge \mathit{empty}$$

$\Box$

5.4 Examples

We now present two simple applications of the projection construct. The first is a pulse generator for variable x which can assume two values: 0 (low) and 1 (high). We first define two types of processes: The first one is *hold*(i) $(i \geq 1)$ which is executed over an interval of length i and ensures that the value of x remains constant in all but the final state:

$$hold(i) \overset{\text{def}}{=} len(i) \wedge \Box(\bigcirc more \rightarrow (\bigcirc x = x)).$$

The other is *switch*(j) which ensures that the value of x is first set to 0 and then changed at every subsequent state:

$$switch(j) \overset{\text{def}}{=} (x = 0) \wedge len(j) \wedge \Box(more \rightarrow (\bigcirc x = 1 - x)).$$

Having defined *hold*(i) and *switch*(j) we can define pulse generators with varying number and length of low and high intervals for x,

$$pulse(i_1, \ldots, i_k) \overset{\text{def}}{=} (hold(i_1), \ldots, hold(i_k))\ prj\ switch(k).$$

The second example is that of special parallel computation. Consider the formula $((len(i_1), len(i_2), \ldots, len(i_k))\ prj\ q) \wedge p$. This allows processes p and q to be executed in a special parallel manner in which p is executed over a series of subintervals, and q is executed at their endpoints:

```
t0      t2                t6                t10
|-------|---------------|---------------|
|-----------------------q-------------->|
t0  t1  t2  t3  t4  t5  t6  t7  t8  t9 t10 t11 t12
|---|---|---|---|---|---|---|---|---|---|---|---|
|<--------------------- p--------------------->|
```

Figure 5.3: Special parallel computation

The projection operator $(p_1, \ldots, p_m)\ prj\ q$ presented in this thesis can be used to specify computations on different time scales. Another possible application area is that of the real time systems. In this case, we could treat $p_1, \ldots, p_m$ as formulas over a series of dense or real intervals and q as a formula over a projected discrete interval.

Chapter 6

Framing

Summary: Problems and principles about framing are discussed and a solution for framing based on a default logic is presented. With this solution, a new assignment operator ($\Leftarrow$), an assignment flag ($af(x)$), and a framing operator ($frame(x)$) are defined in EITL. Finally, some basic framing techniques in temporal logic programming are illustrated with examples.

In a conventional programming language such as C or Pascal, if a variable has not been assigned a new value within a program, the current value of the variable remains the same as its old value, and all variables have this inertial property. This framing technique is simple and often taken for granted. However, in a temporal logic programming language such as Tempura, the situation is different, since a program is executed over a sequence of states (time stamps) and the values of variables are not inherited automatically. Thus, some interesting questions arise: 1. Is it necessary for a variable to inherit its value automatically? 2. How is the value of a variable inherited while a program is executed? 3. Do all variables behave similarly? This chapter intends to answer these questions and to formalize a framing technique in which framed and non-framed variables can be mixed.

This chapter is organized as follows: Section 6.1 discusses why we need framing techniques. Section 6.2 looks at some fundamental aspects of framing issue. In Section 6.3, a solution based on a default logic for framing is introduced. In Section 6.4, some basic techniques for temporal logic programming are discussed. Finally, conclusions are drawn in Section 6.5.

6.1 Why Framing ?

The introduction of a framing technique to temporal logic programming is motivated by both practical and theoretical considerations: improving the efficiency of a program and synchronizing communication for parallel processes. Let us begin with an example in Tempura:

$$x = 1 \wedge (y := 2; y := x + y) \tag{1}$$

The program only tells us that x equals 1 at state s_0, and y equals 2 at state s_1. One may expect that y equals 3 (the sum of the values of x and y) at state s_2. Unfortunately, the program does not satisfy this requirement. The reason is that x is unspecified at s_1 and s_2. So y is unspecified at s_2 (y is also unspecified at s_0). There are several ways to correct it. An ad hoc fix to the problem is to make stability explicit; the above example can be rewritten as

$$x = 1 \wedge \Box(more \rightarrow (\bigcirc x = x)) \wedge (y := 2; y := x + y) \qquad (2)$$

As seen, however, x has to be assigned its current value at the next state repeatedly from one state to another so that its value could be inherited. These additional assignments are tedious and may also decrease the efficiency of the program. For a small program, the repeated assignments may be tolerable; however, in some cases, repeated assignments may be unacceptable. For instance, to maintain a large array while changing a few its elements at different times over an interval, using repeated assignments would be a disaster. Furthermore, the verification and the transformation of programs may also suffer from these excessive assignments. To eliminate them in an implementation, it seems that the best choice is to introduce a framing technique.

As discussed in Chapter 1, in a sequential program, an await statement is useless, since if c is currently false it will remain so forever. In a process within a concurrent program, however, this statement makes sense, since another process, acting in parallel, may cause c to become true.

How can *await*(c) be defined in temporal logic programming like Tempura? It would be difficult without framing. Within Tempura, the statement *halt*(c) (see Chapter 4) may play a role similar to that of the *await*(c) statement; however, *halt*(c) requires that c become true only at the end of an interval. Thus, c must be false until the time at which c is true. However, *halt*(c) does not prevent the variables from being changed. Thus, a problem arises whether we adopt repeated or unrepeated assignments, when we attempt to synchronize parallel components by *halt*(c). For instance, within a finite specified interval, the program

$$x = 0 \wedge halt(x = 1) \wedge len(2)$$

is satisfied by an interval, $< x : 0, x : \_, x : 1 >$, (here we are only concerned with variable x; $\_$ denotes any value in the domain). It forces $x = 1$ at the final state of the interval. On the other hand, if we use the repeated assignment approach to inherit the value of x, the program

$$x = 0 \wedge \Box(more \rightarrow \bigcirc x = x) \wedge halt(x = 1) \wedge len(2)$$

is obviously false. For an unspecified interval, the program

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

is also satisfiable on an interval such as $< x : 0, ..., x : 1 >$. It terminates at some indefinite state where $x = 1$ because x is only defined at the initial state. On the other hand, if we use the repeated assignment approach to carry forward the value of x, the program

$$x = 0 \wedge \Box(more \rightarrow \bigcirc x = x) \wedge halt(x = 1)$$

satisfies the busy waiting (i.e. waiting for $x = 1$ to become true), but it waits forever. No process, acting in parallel, can change x to 1, since such an assignment conflicts with $x = 0$.

The problem is caused by the fact that, unlike conventional programming languages, the values of variables are not inherited automatically from one state to another and the assignments are not destructive in temporal logic programming languages.

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 waiting 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.

Framing is concerned with how the value of a variable from one state can be carried to the next one. Temporal logic offers no solution in this respect; no value from a previous state is assumed to be carried along. Framing techniques have been used in the programming with conventional imperative languages for many years without consideration. However, we have to consider this option carefully in temporal logic programming.

Considerable attention has been given to framing in recent years [68, 40, 53, 83, 39]; however, no consensus has emerged as to the best underlying semantics. Moreover, some of papers deal with framing in a simplified manner, e.g. assuming the values of variables are automatically inherited. This thesis is concerned with investigating the behaviour of concurrent programs under a particular mode of framing in which framed and non-framed variables are mixed.

To deal with framing, a new assignment operator ($\Leftarrow$) and an assignment flag (af) will be defined within EITL. Armed with the assignment flag, a framing operator $frame(x)$ will be formalized. These new constructs will be interpreted within minimal model semantics.

This will allow us to specify the framing status of variables throughout an interval in a flexible manner, and to verify the properties of a reactive system in a manageable way. However, introducing the framing operator destroys the monotonicity and leads to a default logic [76, 56, 57]. Therefore, negation by default has to be used to manipulate the framing operator.

When a framing technique is introduced to temporal logic programming, the semantics of a program may be changed. Therefore, one issue we have to face is how to interpret a framed program; that is, how to capture its intended meaning. This will be discussed in Chapter 7.

With the framing operator, the synchronous communication construct, $await(c)$, can easily be defined. Therefore, real concurrent programs can be managed within our system. We discuss this issue in Chapter 8.

An interpreter [20], written in SICSTUS Prolog, has been developed using the framing technique presented in this thesis. It will be presented in Chapter 9.

6.2 Problems and Principles

As mentioned earlier, framing is concerned with the persistence of the values of variables from one state to another. There are at least two realistic ways to go about framing. One way is directly associated with the definition of the assignment operator as [40] does in an algebraic programming language. There the assignment is defined as follows:

$$x := e \stackrel{\text{def}}{=} x' = e \wedge y_i' = y_i \ (1 \leq i \leq m)$$

where x is a variable, and x' represents the new value of x. $y_1, ..., y_m$ which are different from x are all the other variables within a program. Intuitively, this means that whenever a variable x is assigned a value, the other variables remain stable. However, this method can only manage framing in a limited case in which all variables are framed, and the conjunction of assignments is forbidden since $x := e_1 \wedge y := e_2$ is obviously false if e_1 (or e_2) does not equal the current value of x (or y). In Tempura, however, the definition of parallel composition is based on conjunction (see Chapter 4); this implies that parallel assignments would be forbidden if we adopted the same strategy for framing. The other way in which the values of variables could persist is by means of an explicit operator. This method may enable us to establish a framed environment in a flexible manner in which framed and non-framed variables can be mixed and the framing operator can be used in a sequential, conjunctive, or parallel manner. Intuitively, the meaning of the framing operator, denoted by $frame$, can be stated as follows:

> *frame(x) means that variable x always keeps its old value over an interval if no assignment to x is encountered.* (*)

The crux of the above technique is how to perceive the assignments of values to variables. To identify an occurrence of an assignment to a variable, say x, we make use of a flag called the assignment flag, denoted by a predicate $af(x)$; it is true whenever an assignment of a value to x is encountered, and false otherwise (Note that $af(x)$ is not allowed to use freely in a program but only connected with an assignment (see Condition 1) and a framing operator (see Definitions 6.3, 6.4)). $af(x)$ is easy to understand but difficult to formalize in a logic framework. The problem is that a program provides only positive information, that is, some explicit assignments from which we know those variables assigned explicitly within the program. However, what we need is negative information, i.e. those variables which do not encounter assignments at the current state. It is possible to search for the positive information syntactically and to obtain the negative information by complement. But it is very hard to define the negative information in a logic framework.

Let P be a program and $R = M(P)$. $M(P)$ denotes the set of all P-models in which framing operators are used. In fact, here $M(P)$ equals $M_m(P)$ which is based on the minimal model given in the sequel. We use P-models below because the conditions given below involve formulas which not all are programs but relevant to a program P. The framing technique using an assignment flag should satisfy some necessary conditions, as follows:

Condition 1: $\models_R \Box(af(x) \leftrightarrow \exists b : x \overset{1}{=} b \wedge b \neq nil)$

where $\overset{1}{=}$ is an assignment operator associated with $af(x)$. As a predicate, the assignment flag ($af(x)$) must be defined in a way in which it is associated with some assignment operator and can be used to assert whether or not such an assignment has taken place to x in the execution of a program. Whenever such an assignment is encountered, $af(x)$ should be true. Conversely, when $af(x)$ is true, such an assignment should have been perceived in the execution of the program P. Therefore, whichever way an assignment and assignment flag may be defined, Condition 1 should hold. The $b \neq nil$ means that nil is not allowed to be assigned to a variable by the assignment operator $\overset{1}{=}$.

Condition 2: $\models_R \Box(af(x) \leftrightarrow x \in X)$

where X is the set of assigned variables associated with the assignment operator $\overset{1}{=}$. X is determined syntactically (see Chapter 7). Condition 2 requires that the assignment of a value to x semantically should be equivalent to the assignment of a value to x syntactically. This amounts to requiring that the set of variables assigned semantically within a program should equal the set of variables intended to be assigned by the program at every state.

Condition 3: $\models_R remain(\neg af(x) \wedge \ominus x \neq nil \rightarrow (x = \ominus x))$ or
$\models_R keep(\bigcirc \neg af(x) \wedge x \neq nil \rightarrow (\bigcirc x = x))$

Condition 3 is merely an informal description of the intuitive meaning in (*). It indicates the conditions under which the values of variables are inherited. The first formula is denoted by

$lbf(x)$, and the second by $lff(x)$ (the formal definitions are given later). It implies causality, i.e. when an explicit assignment has not been encountered, an inherited assignment takes place instead; it also implies simultaneity, that is, the disappearance of an assignment using $\overset{1}{=}$ and the appearance of an inherited assignment using $=$ take place simultaneously. Note that *nil* is not allowed to be inherited. So the condition requires $\bigcirc x \neq nil$ with $lbf(x)$, and $x \neq nil$ with $lff(x)$. It is easy to show the following fact:

Fact 6.1 $\overset{1}{=}$ and $=$ are different.

The above can be justified as follows. Let P be a program and x a framed variable in P. By Condition 3, for any interval $\sigma, \sigma \in M(P)$, and any integer $i, 0 < i \preceq |\sigma|$, $(\sigma, 0, i, |\sigma|) \models \neg af(x) \wedge \bigcirc x \neq nil \rightarrow (x = \bigcirc x)$. Suppose $(\sigma, 0, i, |\sigma|) \models \neg af(x)$. That is, no assignment which is associated with $af(x)$ is encountered to x at the current state s_i. Thus, $(\sigma, 0, i, |\sigma|) \models x = \bigcirc x \neq nil$. If $\overset{1}{=}$ were $=$, then $(\sigma, 0, i, |\sigma|) \models x \overset{1}{=} \bigcirc x \wedge \bigcirc x \neq nil$. By Theorem 3.32, we have $(\sigma, 0, i, |\sigma|) \models \exists b : x \overset{1}{=} b \wedge x \neq nil$. This implies, by Condition 1, $(\sigma, 0, i, |\sigma|) \models af(x)$. This yields a contradiction.

$\square$

Fact 6.1 tells us that the assignment operator associated with af and the assignment operator ($=$) inheriting the value of a variable must be different, otherwise a conflict arises. Hence, the existing assignment operators in ITL are not enough to manage framing. In fact, it is impossible to choose $\overset{1}{=}$ from the existing assignment operators: $=$, $\circ\!\!=$, $:=$ and $\leftarrow$ in ITL. By the simultaneity of Condition 3, $\overset{1}{=}$ cannot be $\circ\!\!=$, $:=$ and $\leftarrow$; by Fact 6.1, $\overset{1}{=}$ cannot be $=$. Therefore, $\overset{1}{=}$ has to be defined as a new assignment operator (it is definable in ITL). We will define $\overset{1}{=}$ to be a new assignment operator ($\Leftarrow$) and use the equality ($=$) as the inheritance assignment operator in this thesis.

An important fact we claim is that adding framing operators to ITL makes the underlying semantics a radical shift from monotonicity to non-monotonicity.

Let us recall that the first order logic has the following properties [34]:

(1) reflexivity: $\{w_1, ..., w_n, w\} \vdash w$.
(2) monotonicity: if $\{w_1, ..., w_n\} \vdash w$ then $\{w_1, ..., w_n, u\} \vdash w$.
(3) transitivity: if $\{w_1, ..., w_n\} \vdash u$ *and* $\{w_1, ..., w_n, u\} \vdash w$, then $\{w_1, ..., w_n,\} \vdash w$.

In the expressions above, $w_1, ..., w_n, u, w$ represent formulas of logical language. From above (2), 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. We conclude it in Fact 6.2.

Fact 6.2 (non-monotonicity)

A logic involving framing operators is non-monotonic. That is, given formulas, $w_1, ..., w_n, u, w$, it may happen that $w_1 \wedge ... \wedge w_n \rightarrow w$ holds, but $w_1 \wedge ... \wedge w_n \wedge u \rightarrow w$ does not.

$\square$

For example, we should have, according to the derived properties of framing,

$$x = 1 \wedge len(1) \wedge frame(x) \supset \bigcirc x = 1$$

$$x = 1 \wedge len(1) \wedge frame(x) \wedge \bigcirc(x \Leftarrow 2) \supset \bigcirc x = 2$$

From the above discussion about the framing technique using assignment flag (af) and framing operator, we know the following facts: 1. the assignment flag af must be defined based on an assignment operator; 2. the framing technique must use two different assignment operators: one is the assignment operator associated with the definition of af, and the other is the assignment operator used to inherit the value of a variable when the variable is framed and an explicit assignment to it is absent; 3. the syntactic definition and the semantic interpretation of af must be consistent; 4. a logic involving framing operator is non-monotonic. Therefore, we have to provide a way to perceive the absence of the explicit assignment associated with the assignment flag to a variable so that an inheriting assignment to the variable can take place if the variable is framed.

6.3 Solutions

In this section, we first define some new assignments which are required by framing as discussed earlier; then we define framing operators; and finally, we present a minimal model-based approach for framing.

6.3.1 New Assignments and Framing Operators

Let $S_p = \{x_1, ..., x_n\}(S_p \subset V)$ be a set of state variables within a program p and $\Phi_p = \{p_{x_1}, ..., p_{x_n}\}$ be the set of propositions associated with state variables. Note that variables bound by quantifiers can always be given distinct names by renaming them as necessary. We also assume that a program p does not involve propositions other than Φ_p.

Definition 6.1 (new assignments)

(1) $x_i \Leftarrow e \overset{\text{def}}{=} x_i = e \wedge p_{x_i}\ (1 \leq i \leq n)$

(2) $x_i \leftarrow^+ e \overset{\text{def}}{=} \exists a : \ (a = e \wedge \Box(empty \rightarrow x_i \Leftarrow a))$

(3) $x_i \circ\!\!=^+ e \overset{\text{def}}{=} \exists a : (a = e \wedge \bigcirc(x_i \Leftarrow a))$

(4) $x_i :=^+ e \overset{\text{def}}{=} x_i \circ\!\!=^+ e \wedge skip$

(5) $(x_1, ...x_n)\ op\ (e_1, ..., e_n) \overset{\text{def}}{=} (x_1\ op\ e_1) \wedge ... \wedge (x_n\ op\ e_n)$

where a is a static variable, e an expression, p_{x_i} an atomic proposition associated with state variable x_i $(1 \leq i \leq n)$. p_{x_i} cannot be used for other purposes. In (5), $op ::= \leftarrow^+ | \circ\!\!=^+ | :=^+ | \Leftarrow$.

$\Box$

In the above definition, (1) defines a positive immediate assignment operator; (2)-(5) specify derived assignment operators. The meaning of these assignment operators is similar to those presented in Chapter 4, but they set some proposition(s) to be true besides assigning some value(s) to variable(s) at the same time. These operators are called (in the order of their definitions) positive temporal, next, unit, and multiple assignment operators, respectively. It is now time to define the assignment flag.

Definition 6.2 (assignment flag)

$$af(x_i) \overset{\text{def}}{=} p_{x_i}$$

where proposition p_{x_i} associated with variable x_i is the same as in Definition 6.1, and cannot be used for any other purpose. As expected, whenever $x_i \Leftarrow b$ is encountered, p_{x_i} is set to true, hence $af(x_i)$ is true. Whereas, if no assignment to x_i takes place, p_{x_i} is unspecified. In this case, we will use a minimal model to force it to be false. $\Box$

Armed with the assignment flag, we can define state framing and interval framing operators. Intuitively, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval. We formalize this idea in Definition 6.3 and Definition 6.4. The former invokes the previous operator to look back to the previous state, (1) specifies x_k to be framed at the current sate, and (2) defines x_k to be framed over an interval; while the latter employs the next operator to look forward to the next state, (1) specifies the state framing and (2) defines the interval framing. In fact, the two definitions are equivalent (see Theorem 7.11).

Definition 6.3 (looking back framing)
Let b be a static variable, and $x_1, ..., x_n$ dynamic (state) variables.

(1) $lbf(x_k) \overset{\text{def}}{=} \neg af(x_k) \rightarrow \exists b : (\ominus x_k = b \wedge x_k = b)$
(2) $frame(x_k) \overset{\text{def}}{=} \Box(more \rightarrow \bigcirc lbf(x_k))$
(3) $frame(x_1, ..., x_n) \overset{\text{def}}{=} frame(x_1) \wedge ... \wedge frame(x_n)$

$\Box$

Definition 6.4 (looking forward framing)
Let b be a static variable, and $x_1, ..., x_n$ dynamic (state) variables.

(1) $lff(x_k) \overset{\text{def}}{=} \bigcirc \neg af(x_k) \rightarrow \exists b : (x_k = b \wedge \bigcirc x_k = b)$
(2) $frame'(x_k) \overset{\text{def}}{=} \Box(more \rightarrow lff(x_k))$
(3) $frame'(x_1, ..., x_n) \overset{\text{def}}{=} frame'(x_1) \wedge ... \wedge frame'(x_n)$

$\Box$

Note that the above definitions are on domain D rather than D'. So, *nil* is not needed in contrast with Condition 3.

A dynamic variable x is said to be framed in a program p if $frame(x)$ or $lbf(x)$ or $lff(x)$ is contained in p. A program p is said to be framed if p contains at least one framed variable. In general, a framed program is non-deterministic under the canonical model. Consequently, a framed program can inductively be defined, as follows

- For any variable $x \in V$ and any well-formed expression e, $x = e$, $x \Leftarrow e$, and *empty* are framed programs.
- $lff(x), lbf(x)$, and $frame(x)$ are framed programs.

- If $p, q, p_1, ..., p_m$ are framed programs, then so are the followings:

  $\bigcirc p$, $\Box p$, $p \wedge q$, $p; q$, *if b then p else q*, *while b do p*, $p || q$, $(p_1, ..., p_m)$ *prj* q, and $\exists x : p$.

Fact 6.3

$$\begin{array}{llcl} EQFR & x_i = e_i & \equiv & p_{x_i} \wedge x_i = e_i \vee \neg p_{x_i} \wedge x_i = e_i \\ LBF & lbf(x_i) & \equiv & p_{x_i} \vee \neg p_{x_i} \wedge x_i = \ominus x_i \end{array}$$

□

Proof

EQFR is obviously true. We prove only LBF.

$$\begin{array}{rcll} lbf(x_i) & \equiv & \neg afx_i \rightarrow \exists b : \ominus x_i = b \wedge x_i = b & \\ & \equiv & \neg p_{x_i} \rightarrow \exists b : \ominus x_i = b \wedge x_i = b & \\ & \equiv & \neg p_{x_i} \rightarrow \ominus x_i = a \wedge x_i = a & \text{lemma 4.8} \\ & \equiv & \neg p_{x_i} \rightarrow x_i = \ominus x_i \ (\neq nil) & \\ & \equiv & p_{x_i} \vee \neg p_{x_i} \wedge x_i = \ominus x_i & \end{array}$$

□

By EQFR and LBF, when we reduce a framed program p, whenever $x_i = e_i$ occurs in p, it is replaced by $p_{x_i} \wedge x_i = e_i \vee \neg p_{x_i} \wedge x_i = e_i$; whereas whenever $lbf(x_i)$ occurs in p, it is replaced by $p_{x_i} \vee \neg p_{x_i} \wedge x_i = \ominus x_i$. Then we can reduce p under the canonical model as usual.

Example 6.1

$$\begin{array}{rl} & frame(x) \wedge x \Leftarrow 1 \wedge \bigcirc(x = 2) \wedge len(1) \\ \equiv & \Box(more \rightarrow \bigcirc lbf(x)) \wedge x \Leftarrow 1 \wedge \bigcirc(x = 2) \wedge \bigcirc(empty) \\ \equiv & (more \rightarrow \bigcirc lbf(x)) \wedge \bigcirc\Box(more \rightarrow \bigcirc lbf(x)) \wedge x \Leftarrow 1 \wedge \bigcirc(x = 2) \wedge more \wedge \bigcirc(empty) \\ \equiv & \bigcirc lbf(x) \wedge \bigcirc\Box(more \rightarrow \bigcirc lbf(x)) \wedge x \Leftarrow 1 \wedge \bigcirc(x = 2) \wedge \bigcirc(empty) \\ \equiv & x \Leftarrow 1 \wedge \bigcirc(lbf(x) \wedge \Box(more \rightarrow \bigcirc lbf(x)) \wedge x = 2 \wedge empty) \\ \equiv & x = 1 \wedge p_x \wedge \bigcirc(lbf(x) \wedge x = 2 \wedge empty) \end{array}$$

Thus,

$$p_c^0 \equiv x = 1 \wedge p_x$$

$$\begin{array}{rcl} p_f^0 & \equiv & lbf(x) \wedge x = 2 \wedge empty \\ & \equiv & (p_x \vee \neg p_x \wedge x = \ominus x) \wedge (p_x \wedge x = 2 \vee \neg p_x \wedge x = 2) \wedge empty \\ & \equiv & p_x \wedge x = 2 \wedge empty \vee \neg p_x \wedge x = 1 \wedge x = 2 \wedge empty \\ & \equiv & p_x \wedge x = 2 \wedge empty \end{array}$$

□

This example shows us that although no explicit assignments using $\Leftarrow$ at a state, a potential positive assignment can occur in the state since $x = e$ can be treated as $x = e \wedge p_x \vee x = e \wedge \neg p_x$.

6.3.2 Minimal Models

A framed program p also has its normal form which is similar to the normal form for a non-framed program but the propositions in Φ_p have to be involved.

Definition 6.5 A program q is in normal form if

$$q \overset{\text{def}}{=} \bigvee_{i=1}^{k} q_{ei} \land empty \lor \bigvee_{j=1}^{h} q_{cj} \land \bigcirc q_{fj} \qquad (6.1)$$

where $k, h \geq 0$ $(k + h \geq 1)$ and

- for all $1 \leq j \leq h$, $\bigcirc q_{fj}$ are lec-formulas and q_{fj} are all internal programs.
- q_{cj} $(j \leq h)$ and q_{ei} $(i \leq k)$ are *true* or all state formulas of the form:

  $$(x_1 = e_1) \land ... \land (x_l = e_l) \land \dot{p}_{x_1} \land ... \land \dot{p}_{x_m}$$

  where $e_i \in D$ $(1 \leq i \leq l)$ and $\dot{p}_x$ denotes p_x or $\neg p_x$ and $l \geq 0$ and $m \geq 0$ and $l + m \geq 1$.

$\square$

A similar proof as for Theorem 4.9 can be given to obtain the following conclusion.

Theorem 6.4 If p is a framed program, then there is a program q as defined in (6.1) such that

$$p \equiv q$$

$\square$

Proof Similar to the proof of Theorem 4.9.

$\square$

Example 6.2

$$\begin{aligned}
& frame(x) \land x = 1 \land len(1) \\
\equiv\ & \square(more \rightarrow \bigcirc lbf(x)) \land x = 1 \land \bigcirc(empty) \\
\equiv\ & (more \rightarrow \bigcirc lbf(x)) \land \odot\square(more \rightarrow \bigcirc lbf(x)) \land x = 1 \land more \land \bigcirc(empty) \\
\equiv\ & \bigcirc lbf(x) \land \bigcirc\square(more \rightarrow \bigcirc lbf(x)) \land x = 1 \land \bigcirc(empty) \\
\equiv\ & (p_x \land x = 1 \lor \neg p_x \land x = 1) \land \bigcirc(lbf(x) \land empty)
\end{aligned}$$

Thus,

$$p_c^0 \equiv p_x \land x = 1 \lor \neg p_x \land x = 1$$

$$\begin{aligned}
p_f^0 \ \equiv\ & lbf(x) \land empty \\
\equiv\ & (p_x \lor \neg p_x \land x = 1) \land empty \\
\equiv\ & p_x \land empty \lor \neg p_x \land x = 1 \land empty
\end{aligned}$$

Hence, four models given below can satisfy the program.

$$\sigma_1 =< (\{p_x\}, \{x : 1\}), (\{p_x\}, \phi) >$$

$$\sigma_2 =< (\{p_x\}, \{x : 1\}), (\phi, \{x : 1\}) >$$

$$\sigma_3 =< (\phi, \{x : 1\}), (\{p_x\}, \phi) >$$

$$\sigma_4 =< (\phi, \{x : 1\}), (\phi, \{x : 1\}) >$$

$\square$

As seen, a proposition p_x can appear alone without an equality $x = e$ at a state. Also, a framed program can have a number of canonical models. Thus, a problem we have to face is how to choose a model to satisfy the intended meaning of a program. We interpret framed programs using minimal models.

Definition 6.6 Let p be a framed program, and $\Sigma_p = \{\sigma | \sigma \models_c p\}$. Let $\sigma_1, \sigma_2 \in \Sigma_p$. We define

- $\sigma_{1p} \sqsubseteq \sigma_{2p}$ iff $I^i_{1p} \subseteq I^i_{2p}$ and $|\sigma_1| = |\sigma_2|$ for all $i, 0 \leq i \leq |\sigma_1|$
- $\sigma_1 \sqsubseteq \sigma_2$ iff $\sigma_{1p} \sqsubseteq \sigma_{2p}$
- $\sigma_1 \doteq \sigma_2$ iff $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2 \sqsubseteq \sigma_1$
- $\sigma_1 \sqsubset \sigma_2$ iff $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2 \not\sqsubseteq \sigma_1$

□

Example 6.3

$$< (\{p_x\}, \phi) > \sqsupseteq < (\phi, \{x : 1\}) >$$
$$< (\{p_x\}, \phi) > \sqsupset < (\phi, \{x : 1\}) >$$
$$< (\phi, \{x : 1\}) > \doteq < (\phi, \{x : 2\}) >$$
$$< (\phi, \{x : 1\}) > = < (\phi, \{x : 1\}) >$$

□

Definition 6.7 Let p be a framed program. Then

- $[\sigma]_{\doteq} = \{\sigma' | \sigma' \doteq \sigma, \sigma \models_c p\}$.
- $\bar{\Sigma}_p = \{[\sigma]_{\doteq} | \sigma \models_c p\}$
- $\dot{\Sigma}_p = f(\bar{\Sigma}_p)$, where $f : \bar{\Sigma} \rightarrow \Sigma_p$ is any function such that $f([\sigma]_{\doteq}) \in [\sigma]_{\doteq}$

  Note that (1) $[\sigma]_{\doteq}$ is an equivalence class, i.e. the set of models which are equivalent to σ under $\doteq$; (2) $\dot{\Sigma}_p$ consists of $|\bar{\Sigma}_p|$ elements. Each element belongs to a different equivalence class in $\bar{\Sigma}_p$. $\dot{\Sigma}_p$ is needed in the proof of Theorem 7.1.

□

Let p be a program, and $\sigma = < s_0, ... s_{|\sigma|} >$ be an interval over S_p and Φ_p, where $s_i = (I^i_v, I^i_p)$, I^i_v is defined as in Chapter 4 and I^i_p is canonical interpretation defined as in Section 4.2 of Chapter 4.

Definition 6.8 (the minimal satisfaction relation)
Let p be a program, and (σ, i, k, j) be an interpretation. Then the minimal satisfaction relation $\models_m$ is defined as

$(\sigma, i, k, j) \models_m p$ iff $(\sigma, i, k, j) \models_c p$ and there is no σ' such that $\sigma' \sqsubset \sigma$ and $(\sigma', i, k, j) \models_c p$.

□

A program p is satisfied by a model σ under relation $\models_m$, denoted by $\sigma \models_m p$, if $(\sigma, 0, 0, |\sigma|) \models_m p$. A model σ is a minimal model of program p if $\sigma \models_m p$.

The relations $\equiv_m$ and $\approx_m$ can be defined similarly to the relations $\equiv$ and $\approx$. $p \equiv_m q$ iff for all σ, all $0 \leq k \preceq |\sigma|$, $(\sigma, 0, k, |\sigma|) \models_m p \Leftrightarrow (\sigma, 0, k, |\sigma|) \models_m q$. $p \approx_m q$ iff for all σ, $\sigma \models_m p \Leftrightarrow \sigma \models_m q$. The relations $\equiv_m$ and $\approx_m$ are also equivalence relations over the set of programs. That is, they are reflexive, symmetric and transitive.

The strong implication relation under the minimal model, $\supset_m$, can also be defined similarly to $\supset$. That is, $p \supset_m q$ iff for all σ, and for all $0 \leq k \preceq |\sigma|)$, $(\sigma, 0, k, |\sigma|) \models_m p \Rightarrow (\sigma, 0, k, |\sigma|) \models_m q$.

Note that the definition of the minimal model of a program p is also independent of its syntax in the sense that the definition does not refer to the structure of the program, and can be applied to temporal formulas.

Example 6.4 The program p in Example 6.2 has only one minimal model $\sigma_4 = (<\phi, \{x : 1\}), (\phi, \{x : 1\})>$. The formula P_1 in Example 4.2 has only two minimal models, namely, $<\phi, \{B\}>$ and $<\{A\}, \phi>$. □

The intended meaning of a program p is captured by its minimal model. For instance, if p is $x_1 \Leftarrow 1 \wedge frame(x_1) \wedge len(1)$ then under the minimal model, $x_1 = 1$ defined at both state s_0 and s_1, this is the intended meaning of p. However, within only the canonical model, p_{x_1} is unspecified at state s_1, so it could be true at s_1. This causes x_1 to be unspecified at state s_1. Therefore, x_1 could be any value from its domain. Using the framing technique, the program (1) in the introduction can be amended as follows

$$frame(x) \wedge (x = 1 \wedge y := 2; y := x + y) \tag{3}$$

As seen, this program has the same meaning as, but is more concise than, the program (2) in the introduction. Furthermore, the implementation of program (3) is simpler than program (2) (see Chapter 9). Therefore, program (3) is more efficient than program (2).

6.4 Basic Framing Techniques

In this section, we show how the framing techniques can be used in temporal logic programming; and some basic techniques in various types of programs including sequential, conjunctive, parallel, or mixed computations are illustrated with examples. Each example indicates a useful property.

We consider first a simple computation: given a positive integer n, to compute the sum of all integers in the sequence $1, ..., n$; and the sum of the sums of integers in every prefix of the sequence $1, ..., n$, i.e. $1 + (1 + 2) + ... + (1 + ... + n)$. In the following, let n be a static variable, x, y, s dynamic variables. x denotes an integer, y denotes $1 + ... + n$, and s denotes $1 + (1 + 2) + ... + (1 + ... + n)$. The programs are designed in different manners for the purpose of showing the different applications of framing techniques.

1. Sequential computation

   The following program computes y and s in a sequential way:

   $$frame(x, y, s) \wedge (x, y, s) = (0, 0, 0) \wedge for\ n\ times\ do\ (x :=^+ x + 1; y :=^+ y + x; s :=^+ s + y).$$

The computation of the program is depicted in Fig 6.1 with $n = 4$. The correctness of the program is obvious. In each iteration, x is incremented by 1, y by x, and s by y. After n time iterations, the computation sequences are $1, .., n$ for x, $0, 1, 1 + 2, ..., 1 + ... + n$ for y, and $0, 0, 1, 1 + (1 + 2), ..., 1 + (1 + 2) + ... + (1 + 2 + ... + n)$ for s. Hence, the values of y and s at the final state are correct results.

```
     s0   s1   s2   s3   s4   s5   s6   s7   s8   s9  s10  s11  s12
     |----|----|----|----|----|----|----|----|----|----|----|----|
    x=0   1    1    1    2    2    2    3    3    3    4    4    4
    y=0   0    1    1    1    3    3    3    6    6    6    10   10
    s=0   0    0    1    1    1    4    4    4    10   10   10   20
```

Fig. 6.1 A Sequential Computation

2. Conjunctive Computation

The following program also computes y and s but in a conjunctive way.

```
$frame(x, y, s) \wedge (x, y, s) = (0, 0, 0)$
$\wedge for\ n\ times\ do\ x :=^{+} x + 1$
$\wedge for\ (n + 1)\ times\ do\ y :=^{+} y + x$
$\wedge for\ (n + 2)\ times\ do\ s :=^{+} s + y$
```

The computation is pictorialized in Fig.6.2 with $n = 4$.

```
     s0    s1    s2    s3    s4    s5    s6
     |-----|-----|-----|-----|-----|-----|
    x=0    1     2     3     4     4     4
    y=0    0     1     3     6     10    10
    s=0    0     0     1     4     10    20
```

Fig. 6.2 A Conjunctive Computation

3. Parallel Computation

The following program also computes y and s but in a parallel manner.

```
$frame(x, y, s) \wedge (x, y, s) = (0, 0, 0)$
$||\ for\ n\ times\ do\ x :=^{+} x + 1$
$||\ (skip; for\ n\ times\ do\ y :=^{+} y + x)$
$||\ (len(2); for\ n\ times\ do\ s :=^{+} s + y)$
```

The computation proceeds similarly to the conjunctive computation (see Fig. 6.3).

```
  s0    s1    s2    s3    s4    s5    s6
  |-----|-----|-----|-----|-----|-----|
 x=0    1     2     3     4     4     4
 y=0    0     1     3     6     10    10
 s=0    0     0     1     4     10    20
```

Fig. 6.3 A Parallel Computation

4. A Mixed Case of Sequential and Conjunctive Computations

The following program also computes y and s but in a mixed manner.

$frame(x,y,s)\wedge(x,y,s) = (0,0,0)\wedge(x :=^{+} x+1; (x,y) :=^{+} (x+1, y+x);$
$for\ n-2\ times\ do\ (x,y,s) :=^{+} (x+1, y+x, s+y); (y,s,) :=^{+} (y+x, s+y); s :=^{+} s+y)$

The computation proceeds in a similar way as Fig.6.2 shows. However, at state s_1, s_2, the computations proceed sequentially, at state s_3, s_4, conjunctively by *for* iteration, and at state s_5, s_6, again sequentially.

5. A Mixed Case of Conjunctive and Parallel Computations

The following program also computes y and s but in a mixed way.

$frame(x,y,s)\wedge(x,y,s) = (0,0,0)\ \wedge for\ n\ times\ do\ x :=^{+} x+1$
$\wedge((skip; for\ n\ times\ do\ y :=^{+} y+x)||(len(2); for\ n\ times\ do\ s :=^{+} s+y))$

The computation proceeds basically in the same way as in Fig. 6.3.

6. A Mixed Case of Sequential and Parallel Computations

The following program (algorithm borrowed from [66]) computes binomial coefficient $\binom{n}{k} = \frac{n.(n-1).....(n-k+1)}{1.2.....k}$:

```
$k = 2\wedge n = 3\wedge y_1 = n\wedge y_2 = 1\wedge x = 1\wedge frame(x, y_1, y_2)\wedge$
(
$while(y_1 > (n-k))\ do$
    $(x :=^{+} x * y_1; y_1 :=^{+} y_1 - 1)$
||
$while(y_2 \leq k) do$
    $(halt(y_1 + y_2 \leq n);$
     $x :=^{+} x/y_2;$
     $y_2 :=^{+} y_2 + 1$
    )
)
```

In the program, k, n are static variables, $(0\leq k\leq n)$, and x, y_1, y_2 are framed variables. The output of the program is stored in the variable x. The computation is illustrated in Fig 6.4. The program is justified as follows: the first *while* statement computes the product $n.(n-1).....(n-k+1)$. These factors are successively computed in variable y_1.

The second *while* statement responsible for the denominator, successively divides x by the factors, $1, 2, ..., k$, using the integer-division operator $/$. That x is divisible by y_2 is guaranteed by the general property by which a product of m consecutive integers is evenly divisible by $m!$. Thus, x should be divided by y_2, which completes the stage of dividing x by y_2, only when at least y_2 factors have already been multiplied into x by the first *while* statement. Since it multiplies x by $n, (n-1)$, etc. and y_1 holds the value of the next factor to be multiplied, the number of factors that have been multiplied into x as soon as $y_2 \leq n - y_1$, or equivalently, $y_1 + y_2 \leq n$. This is the condition for the *halt* statement.

```
          s0    s1    s2    s3    s4    s5
          |-----|-----|-----|-----|-----|
        y1=3    3     2     2     1     1
        y2=1    1     1     2     2     2
         x=1    3     3     6     3     3
```

Fig. 6.4 Sequential and Parallel Computations

7. A Mixed Case of Framed and Non-framed Variables

   To Compute $s_1 = 1 + 3 + 5 + ... + (2n-1)$, and $s_2 = 2 + 4 + 6 + ... + 2n$, we have the following program:

   $frame(s_1, s_2) \wedge (x, s_1, s_2) = (1, 0, 0) \wedge len(2 * n - 1) \wedge \Box(x \circ\!\!= x + 1)$
   $\wedge \Box(if (x\ mod\ 2 = 1)\ then\ s_1 \circ\!\!=^{+} s_1 + x\ else\ s_2 \circ\!\!=^{+} s_2 + x)$

   where n is a static variable, x is an non-framed variable, and s_1, s_2 are framed variables. The computation is illustrated in Fig.6.5.

```
      s0    s1    s2    s3    s4    s5    s6    s7    s8
      |-----|-----|-----|-----|-----|-----|-----|-----|
     x=1    2     3     4     5     6     7     8     9
    s1=0    1     1     4     4     9     9     16    16
    s2=0    0     2     2     6     6     12    12    20
```

Fig. 6.5 A Computation with Framed and Non-framed Variables

8. Initialization

   We do not require that variables must be initialized. The following program computes the greatest common divisor (GCD)

   $a = 6 \wedge b = 4 \wedge y_1 = a \wedge y_2 = b \wedge frame(y_1, y_2)$
   $while\ y_1 \neq y_2\ do\ if\ y_1 > y_2\ then\ y_1 :=^{+} y_1 - y_2\ else\ y_2 :=^{+} y_2 - y_1;$
   $g := y_1$

   The program is based on Euclid's well known algorithm [1]. The computation is depicted in Fig 6.6. In the program, a, b are static variables, y_1, y_2 are framed dynamic variables, and g is non-framed dynamic variable not initialized at s_0, even is not defined at s_1, s_2, only assigned at the final state s_3. The question mark (?) in Fig 6.6 means that we do not know the value of g at the marked state (the same meaning for below).

```
   s0    s1    s2    s3
   |-----|-----|-----|
y1=6     2     2     2
y2=4     4     2     2
 g=?     ?     ?     2
```

Fig. 6.6 Initialization

9. The framing operator can be used in different parallel components and different subintervals

   The following program computes y and s again as in 1. The computation is pictorialized in Fig. 6.7 with $n = 4$.

   $$\begin{array}{l} halt(x = 4)\land x = 0\land\Box(more\rightarrow x := x + 1) \\ \|(skip; frame(y)\land y = 0\land\Box(more\rightarrow y :=^{+} y + x)\land len(4) \\ \|(len(2); frame(s)\land s = 0\land\Box(more\rightarrow s :=^{+} s + y)\land len(4) \end{array}$$

```
 s0    s1    s2    s3    s4    s5    s6
 |-----|-----|-----|-----|-----|-----|
x=0    1     2     3     4     4     4
y=?    0     1     3     6     10    10
s=?    ?     0     1     4     10    20
```

Fig. 6.7 Frame operators in Different Processes

   The computation is similar to 3 but $frame(y)$ and $frame(s)$ are used in different parallel processes, and start at different states. Note that y is unspecified at state s_0, and s is unspecified at states s_0 and s_1.

10. A variable can be framed on one subinterval and non-framed on another subinterval

    The following program is satisfiable.

    $$frame(x)\land x = 1\land len(5); \Box(x \circ\!\!=^{+} x + 1)\land len(5); frame(x)\land len(5).$$

11. The framing operator can handle the immediate assignment

    The following example is satisfiable within our notation.

    $$frame(x)\land(x = 1\land len(3); x\Leftarrow 5\land x :=^{+} x + 1)$$

The framing operator can be used in programs executed over either a local interval or a projected interval, or local intervals and projected intervals. The examples given below are intended to illustrate these respects.

12. The framing operator with a local interval in the projection construct

This example shows how the framing operator can be used in programs executed over a local interval.

$$(len(2) \wedge \Box(more \rightarrow i :=^{+} i+2), frame(i) \wedge len(4)) \; prj$$

$$(i = 2 \wedge j = 0 \wedge len(2) \wedge \Box(more \rightarrow j :=^{+} j + i))$$

```
  s0    s1    s2    s3    s4    s5    s6
  |-----|-----|-----|-----|-----|-----|
i=2     4     6     6     6     6     6
  |-----------|-----------------------|
j=0           2                       8
```

Fig. 6.8 Framing operator with a local interval

13. The framing operator being in conjunction with the projection construct

This example shows how the framing operator can be used in conjunction with the projection construct in a program.

$$((len(2) \wedge \Box(more \rightarrow i :=^{+} i+2), len(4)) \; prj$$

$$(i = 2 \wedge j = 0 \wedge len(2) \wedge \Box(more \rightarrow j :=^{+} j + i))) \wedge frame(i)$$

```
  s0    s1    s2    s3    s4    s5    s6
  |-----|-----|-----|-----|-----|-----|
i=2     4     6     6     6     6     6
  |-----------|-----------------------|
j=0           2                       8
```

Fig. 6.9 Framing operator with a local interval

Basically, this program proceeds in a similar way to the program shown in Fig. 6.8. That is, the framing operator affects every state over the local interval.

14. The framing operator with a projected interval in the projection construct

This example shows how the framing operator can be used in programs executed over a projected interval.

$$(len(2) \wedge \Box(more \rightarrow i :=^{+} i+2), len(4) \wedge \Box(more \rightarrow i :=^{+} i+1)) \; prj$$

$$(i = 2 \wedge j = 0 \wedge frame(j) \wedge len(3) \wedge j :=^{+} j + i)$$

```
 s0    s1    s2    s3    s4    s5    s6   s7
 |-----|-----|-----|-----|-----|-----|
i=2    4     6     7     8     9     10
 |-----------|-----------------------|-----|
j=0          2                       2     2
```

Fig. 6.10 Framing operator with a projected interval

15. The framing operator with projected and local intervals in the projection construct

This example shows how the framing operator can be used in programs executed over projected and local intervals.

$$(len(2) \wedge \Box(more \rightarrow i :=^{+} i+2), len(4) \wedge frame(i))\ prj$$
$$(i = 2 \wedge j = 0 \wedge frame(j) \wedge len(3) \wedge j :=^{+} j + i)$$

```
 s0    s1    s2    s3    s4    s5    s6   s7
 |-----|-----|-----|-----|-----|-----|
i=2    4     6     6     6     6     6
 |-----------|-----------------------|-----|
j=0          2                       2     2
```

Fig. 6.11 Framing operator with both local and projected intervals

However, in the case in which framing operators are used both in local and projected intervals for one variable, one has to be careful because an error may be hidden. For instance, the following program is false because it causes $i = 6 \wedge i = 2$ at state s_6. Note that i equals 6 rather than 2 at state s_2 because an assignment is encountered at that state with the local state. It is a programmer's responsibility to make a program correct when the program involves framing operators with the projection construct. To avoid the problem, in practice, we suggest using the framing operator for a variable only once either with a local interval or with a projected interval.

$$(len(2) \wedge \Box(more \rightarrow i :=^{+} i+2), len(4) \wedge frame(i))\ prj$$
$$(i = 2 \wedge j = 0 \wedge frame(i) \wedge len(3) \wedge j :=^{+} j + i)$$

```
 s0    s1    s2    s3    s4    s5    s6   s7
 |-----|-----|-----|-----|-----|-----|
i=2    4     6     6     6     6     ?
 |-----------|-----------------------|-----|
j=0          2                       ?     ?
```

Fig. 6.12 A wrong use of the framing operator

This chapter developed the framing technique by using the minimal model. However, we have not investigated the logic laws relevant to the minimal model. These will be discussed in the next chapter.

Chapter 7

Minimal Model Semantics of Framed Programs

Summary: In order to capture semantics of framed temporal logic programs, a minimal model theory is developed. Algebraic properties of framing operators are formalized. An example is given to illustrate how to use the theory for reducing a framed program.

Semantics of a program in imperative languages can be captured in an operational or denotational or axiomatic manner. In temporal logic programming, these semantics of a program can also be investigated. Since a temporal logic programming language, e.g. Tempura, is a subset of the corresponding logic, and the logic has its model theory and its axiomatic system, the semantics of a program can be captured naturally by the model theory and axiomatic theory respectively. Of course, when executed, a program can also be interpreted in a more operational way. Actually, the model theory, in some sense, plays a similar role in temporal logic programming languages as the denotational semantics theory in imperative programming languages.

To capture the temporal semantics of non-framed programs in Tempura, the canonical model has been introduced to interpret programs. Within this model, the semantics of a non-framed program is well captured. However, since introducing a framing operator destroys monotonicity, a canonical model may no longer capture the intended meaning of a program. A program, therefore, can have different meanings under different models. To interpret a framed program faithfully, minimal models will be employed in this thesis.

This chapter is devoted to the development of the minimal model in detail. Some conclusions regarding the minimal model are presented in Section 7.1. Furthermore, algebraic properties of the framing operators are formalized in Section 7.2. Finally, in Section 7.3, an example is given to illustrate how to use the theory for reducing a framed program. Conclusions are drawn in Section 7.4.

7.1 Minimal Model Semantics

In this section, we prove, first, the existence of a minimal model for a given program. Then, we discuss the substitution law under the minimal model. Subsequently, we consider the normal form of a framed program in the extended Tempura.

Theorem 7.1 Let p be a satisfiable framed program (which may be non-terminating, and/or non-deterministic). If, (1) p has at least one finite model or (2) p has finitely many models, then p has at least one minimal model on propositions.

Proof

Let $\dot{\Sigma}_p$ be defined as in Definition 6.7. Since p is satisfiable under the canonical interpretation on propositions, $\dot{\Sigma}_p \neq \phi$. Furthermore, if program p has at least one finite model σ, then we claim that $\Gamma = \{\sigma'_p | \ \sigma' \in \dot{\Sigma}_p$ and $|\sigma'| = |\sigma|\}$ is finite. This is immediately obvious since the proposition set Φ_p is finite. Thus, let $\sigma^0_p \in \Gamma$ be an arbitrary canonical interpretation sequence on propositions. If σ^0_p is not a minimal interpretation sequence on propositions, then there exists a $\sigma^1_p \in \Gamma$ such that $\sigma^0_p \sqsupseteq \sigma^1_p \wedge \sigma^0_p \neq \sigma^1_p$. Analogously, if σ^1_p is not a minimal interpretation sequence on propositions, then there exists a $\sigma^2_p \in \Gamma$ such that $\sigma^1_p \sqsupseteq \sigma^2_p \wedge \sigma^1_p \neq \sigma^2_p$, ..., and so on. In this way, we obtain a sequence:

$$\sigma^0_p \sqsupset \sigma^1_p \sqsupset \sigma^2_p \sqsupset \ldots \tag{7.1}$$

Since Γ is finite, so is sequence (7.1). Therefore, there exists a σ^m_p $(m \geq 0)$, i.e. the last one in sequence (7.1), such that σ^m_p is a minimal interpretation sequence.

If p has finitely many models then $\dot{\Sigma}_p$ is finite. Let $\Gamma = \dot{\Sigma}_p$. Thus, a similar argument to the above can be given to produce the sequence (7.1). □

Theorem 7.1 asserts the existence of a minimal model for a given (possibly non-deterministic or non-terminable) program as long as the program has finitely many models or at least one finite model. For a program which has infinitely many infinite models and has no finite model, the sequence (7.1) above can be infinite and each σ^i_p in it is also an infinite sequence; but, in this case, sequence (7.1) has a limit because the sequence is monotonically decreasing with the lower bound $< \emptyset, \emptyset, \ldots >$. Unfortunately, we cannot know whether the limit is a canonical interpretation of program p.

Introducing the framing operator into programs destroys the well known monotonic law. This implies that some logic laws related to the monotonic law such as substitution law are also no longer valid within framed programs.

Theorem 7.2 Let $p, p', p_1, ..., p_m,$ and q be framed programs. Then

1 If $p \equiv_m p'$ and $q \equiv_m q'$ then $p \wedge q \equiv_m p' \wedge q'$ may not hold.
2 If $p \equiv_m p'$ and $q \equiv_m q'$ then $p; q \equiv_m p'; q'$ may not hold.
3 If $p \equiv_m p'$ and $q \equiv_m q'$ then $p \| q \equiv_m p' \| q'$ may not hold.
4 If $p \equiv_m p'$ then $p^+ \equiv_m p'^+$ may not hold.
5 If $p_i \equiv_m p'_i$ $(1 \leq i \leq m)$ and $q \equiv_m q'$ then $(p_1, ..., p_i, ... p_m)\ prj\ q \equiv_m (p'_1, ..., p'_i, ..., p'_m)\ prj\ q'$ may not hold.

Proof

We prove only 1 and 2.

The proof of 1:

Let p be $y = 1 \wedge \bigcirc(\neg af(y) \rightarrow y = \ominus y)$, p' be $y = 1 \wedge \bigcirc(y = \ominus y)$, q be $y = 1 \wedge \bigcirc(\neg af(y) \rightarrow y = \ominus y) \wedge \bigcirc (y \Leftarrow 9)$, and q' be $y = 1 \wedge \bigcirc(y \Leftarrow 9)$. We have

$$p \equiv_m p' \text{ and } q \equiv_m q'$$

However, $p \wedge q \equiv_m q$ but $p' \wedge q' \equiv_m false$.

The proof of 2:

Let p be $y = 1 \wedge \bigcirc(\neg af(y) \rightarrow y = \ominus y) \wedge len(1)$, p' be $y = 1 \wedge \bigcirc(y = \ominus y) \wedge len(1)$, q be $(\neg af(y) \rightarrow y = \ominus y) \wedge y \Leftarrow 9 \wedge empty$, and q' be $y \Leftarrow 9 \wedge empty$. Thus,

$$p \equiv_m p' \text{ and } q \equiv_m q'$$

However, there is a σ such that $(\sigma, 0, 0, |\sigma|) \models_m p; q$ but for any σ, $(\sigma, 0, 0, |\sigma|) \not\models_m p'; q'$ since $y = 9$ conflicts with $y = 1$ at the final state of the interval.

□

Even if the conditions in Theorem 7.2 are strengthened, the conclusions can still be invalid. Theorem 7.3 is a stronger version of Theorem 7.2. Its proof is easier than Theorem 7.2, and is therefore omitted here.

Theorem 7.3 Let $p, p', p_1, ..., p_m$, and q be framed programs. Then

1 If $p \equiv_m p'$ then $p \wedge q \equiv_m p' \wedge q$ may not hold.
2 If $p \equiv_m p'$ then $q \wedge p \equiv_m q \wedge p'$ may not hold.
3 If $p \equiv_m p'$ then $p; q \equiv_m p'; q$ may not hold.
4 If $p \equiv_m p'$ then $q; p \equiv_m q; p'$ may not hold.
5 If $p \equiv_m p'$ then $p \| q \equiv_m p' \| q$ may not hold.
6 If $p \equiv_m p'$ then $q \| p \equiv_m q \| p'$ may not hold.
7 If $p_i \equiv_m p'_i$ then $(p_1, ..., p_i, ...p_m)\ prj\ q \equiv_m (p_1, ..., p'_i, ..., p_m)\ prj\ q$ may not hold.
8 If $q \equiv_m q'$ then $(p_1, ..., p_i, ...p_m)\ prj\ q \equiv_m (p_1, ..., p_i, ..., p_m)\ prj\ q'$ may not hold.

□

It is clear that Theorem 7.2 and Theorem 7.3 destroy the substitution law presented in Theorem 3.7 within framed programs under the minimal model. Therefore, although we have proved a collection of logic laws under the basic and canonical models in the preceding chapters, these laws need to be re-examined for framed programs under the minimal model. As a matter of fact, all logic laws with the equivalence are valid under the minimal model. We discuss this in the following.

As seen in Theorem 7.2 and Theorem 7.3, the problems arise in the case when a minimal model takes effect on a sub-program, that is, a proposition p_x associated with the dynamic variable x is evaluated to false under the minimal model in the context within the sub-program, and another sub-program in conjunction with this sub-program contains a positive immediate assignment which conflicts with $\neg p_x$. In fact, this violates the principle of the minimal model because the minimal model requires that whole program, i.e. all the conjuncts at the current state be taken into account before a 'default effect' functions. However, when reducing composite programs such as $p \wedge q$, $p; q$, $p \| q$, and $(p_1, ..., p_m)\ prj\ q$, we have to reduce sub-programs one by one and the components contained in a sub-program one part by one part. Therefore, to ensure a reduction process without 'default conflicts', whenever a 'default effect' is invoked by the minimal model, we have to make sure that it is impossible to come up with a positive immediate assignment in the later reduction which conflicts with the current 'default effect'.

In the following, two simple but useful theorems regarding the substitution laws are formalised. For our purpose, these laws and the laws provided in the preceding chapters are sufficient to reduce a framed program under the minimal model. Before proving them, we first prove Theorem 7.4 and Lemma 7.5.

Theorem 7.4 Let p and q be framed programs.

$$MIN - EQ \quad \text{If } p \equiv q \text{ then } p \equiv_m q$$

Proof

Let σ be a model and k an integer, $0 \leq k \preceq |\sigma|$. Suppose $(\sigma, 0, k, |\sigma|) \models_m p$. Then, $(\sigma, 0, k, |\sigma|) \models p$. Since $p \equiv q$, we have $(\sigma, 0, k, |\sigma|) \models q$. If $(\sigma, 0, k, |\sigma|) \not\models_m q$, then there is a σ' such that $(\sigma', 0, k, |\sigma|) \models q$ and $\sigma' \sqsubset \sigma$. By $p \equiv q$, we have $(\sigma', 0, k, |\sigma|) \models p$ and $\sigma' \sqsubset \sigma$. This contradicts $(\sigma, 0, k, |\sigma|) \models_m p$. Hence $(\sigma, 0, k, |\sigma|) \models_m q$.

Conversely, if $(\sigma, 0, k, |\sigma|) \models_m q$ then, in the same way, we can prove $(\sigma, 0, k, |\sigma|) \models_m p$.

□

Theorem 7.4 is useful since it shows that the strong equivalence relation under the basic (canonical) model can be inherited under the minimal model. For example, $(p \vee q) \wedge r \equiv_m p \wedge r \vee q \wedge r$ since $(p \vee q) \wedge r \equiv p \wedge r \vee q \wedge r$.

Lemma 7.5 Let σ be a model, r an integer, $0 \leq r \preceq |\sigma|$, and p, q framed programs.

$$MIN - OR \quad \text{If } (\sigma, 0, r, |\sigma|) \models_m p \vee q \text{ then } (\sigma, 0, r, |\sigma|) \models_m p \text{ or } (\sigma, 0, r, |\sigma|) \models_m q.$$

Proof

Suppose $(\sigma, 0, r, |\sigma|) \models_m p \vee q$. We have $(\sigma, 0, r, |\sigma|) \models p$ or $(\sigma, 0, r, |\sigma|) \models q$. Without loss of generality, let $(\sigma, 0, r, |\sigma|) \models p$. If $(\sigma, 0, r, |\sigma|) \not\models_m p$, then there exists a σ' such that $(\sigma', 0, r, |\sigma'|) \models p$ and $\sigma' \sqsubset \sigma$ leading to $(\sigma', 0, r, |\sigma'|) \models p \vee q$ and $\sigma' \sqsubset \sigma$. This contradicts $(\sigma, 0, r, |\sigma|) \models_m p \vee q$.

□

We now prove two useful theorems concerning substitution under the minimal model.

Theorem 7.6 Let $q \equiv \bigvee_{i=1}^{k} q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj}$ be normal form of a framed program q. If p_x is not contained in q_{ei} $(1 \leq i \leq k)$ and q_{cj} $(1 \leq j \leq h)$, then

$$q \equiv_m \neg p_x \wedge q$$

Proof

$$\begin{aligned}
q &\equiv p_x \wedge q \vee \neg p_x \wedge q \\
&\equiv \bigvee_{i=1}^{k} p_x \wedge q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} p_x \wedge q_{cj} \wedge \bigcirc q_{fj} \\
&\quad \vee \bigvee_{i=1}^{k} \neg p_x \wedge q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} \neg p_x \wedge q_{cj} \wedge \bigcirc q_{fj} \\
\neg p_x \wedge q &\equiv \bigvee_{i=1}^{k} \neg p_x \wedge q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} \neg p_x \wedge q_{cj} \wedge \bigcirc q_{fj}
\end{aligned}$$

Let σ be a model and r an integer, $0 \leq r \preceq |\sigma|$. Suppose $(\sigma, 0, r, |\sigma|) \models_m \neg p_x \wedge q$. Then, $(\sigma, 0, r, |\sigma|) \models \neg p_x$ and $(\sigma, 0, r, |\sigma|) \models q$. If $(\sigma, 0, r, |\sigma|) \not\models_m q$, then there is σ' such that

$(\sigma',0,r,|\sigma'|) \models q$ and $\sigma' \sqsubset \sigma$. From $(\sigma,0,r,|\sigma|) \models \neg p_x$ and $\sigma' \sqsubset \sigma$, it follows that $(\sigma',0,r,|\sigma'|) \models \neg p_x$. Hence, $(\sigma',0,r,|\sigma'|) \models \neg p_x \wedge q$. So, $(\sigma,0,r,|\sigma|) \models_m \neg p_x \wedge q$ could not hold. Thus, we obtain a contradiction.

Conversely, suppose $(\sigma,0,r,|\sigma|) \models_m q$. We claim $(\sigma,0,r,|\sigma|) \models_m \neg p_x \wedge q$.

If $(\sigma,0,r,|\sigma|) \models_m p_x \wedge q$, then by Lemma 7.5, $(\sigma,0,r,|\sigma|) \models_m \bigvee_{i=1}^{k} p_x \wedge q_{ei} \wedge empty$ or $(\sigma,0,r,|\sigma|) \models_m \bigvee_{j=1}^{h} p_x \wedge q_{cj} \wedge \bigcirc q_{fj}$. If $(\sigma,0,r,|\sigma|) \models_m \bigvee_{i=1}^{k} p_x \wedge q_{ei} \wedge empty$, then we can construct a σ' from σ by a single change (note that $r = |\sigma|$ in this case), namely,

$$I_p'^{|\sigma|} = I_p^{|\sigma|} - \{p_x\}$$

Since p_x is not contained in q_{ei} $(1 \le i \le k)$, $(\sigma',0,r,|\sigma'|) \models \bigvee_{i=1}^{k} \neg p_x \wedge q_{ei} \wedge empty$. So, $\sigma' \sqsubset \sigma$ and $(\sigma',0,r,|\sigma'|) \models q$. This contradicts $(\sigma,0,r,|\sigma|) \models_m q$.

If $(\sigma,0,r,|\sigma|) \models_m \bigvee_{j=1}^{h} p_x \wedge q_{cj} \wedge \bigcirc q_{fj}$ then we can construct a σ' from σ by a single change, namely,

$$I_p'^{r} = I_p^{r} - \{p_x\}.$$

Since p_x is not contained in q_{cj} $(1 \le j \le h)$, and for all $1 \le j \le h$, $\bigcirc q_{fj}$ are lec-formulas and q_{fj} are internal programs, q_{fj} do not refer to p_x at the current state. Thus, $(\sigma',0,r,|\sigma'|) \models \bigvee_{j=1}^{h} \neg p_x \wedge q_{cj} \wedge \bigcirc q_{fj}$. So, $\sigma' \sqsubset \sigma$ and $(\sigma',0,r,|\sigma'|) \models q$. This contradicts $(\sigma,0,r,|\sigma|) \models_m q$.

Therefore, if $(\sigma,0,r,|\sigma|) \models_m q$, then, by Lemma 7.5, it must be $(\sigma,0,r,|\sigma|) \models_m \neg p_x \wedge q$.

$\square$

Corollary 7.7 Let $q \equiv \bigvee_{i=1}^{k} q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj}$ be the normal form of a framed program q. If p_x and $x = e'$, where $e' \not= e$ $(e', e \in D)$, are not contained in q_{ei} $(1 \le i \le k)$ and q_{cj} $(1 \le j \le h)$, then

$$x = e \wedge q \equiv_m \neg p_x \wedge x = e \wedge q$$

Proof

Since $x = e'$ $(e' \not= e)$ is not contained in q_{ei} $(1 \le i \le k)$ and q_{cj} $(1 \le j \le h)$, $x = e \wedge q \not\equiv (\not\equiv_m) false$ if $q \not\equiv (\not\equiv_m) false$. Moreover, for all $1 \le j \le h$, q_{fj} are internal programs and $\bigcirc q_{fj}$ are lec-formulas. $x = e$ being in conjunction with q does not affect the evaluation of $\bigcirc q_{fj}$.

Since p_x is not contained in q_{ei} $(1 \le i \le k)$ and q_{cj} $(1 \le j \le h)$, p_x is not contained in $q_{ei} \wedge x = e$ $(1 \le i \le k)$ and $q_{cj} \wedge x = e$ $(1 \le j \le h)$. Thus, taking $x = e \wedge q$ to be the q in the Theorem 7.6, we obtain

$$x = e \wedge q \equiv_m \neg p_x \wedge x = e \wedge q$$

$\square$

Theorem 7.8 Let $q \equiv \bigvee_{i=1}^{k} q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} q_{cj} \bigcirc q_{fj}$ be the normal form of a framed program q. If p_x and $x = e'$, where $e' \not= e$ $(e', e \in D)$, are not contained in q_{ei} $(1 \le i \le k)$ and q_{cj} $(1 \le j \le h)$, then

$$(p_x \vee \neg p_x \wedge x = e) \wedge q \equiv_m \neg p_x \wedge x = e \wedge q$$

Proof

$$\begin{aligned}
& (p_x \vee \neg p_x \wedge x = e) \wedge q \\
\equiv\ & p_x \wedge q \vee \neg p_x \wedge x = e \wedge q \\
\equiv\ & \bigvee_{i=1}^{k} p_x \wedge q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} p_x \wedge q_{cj} \wedge \bigcirc q_{fj} \\
& \vee \bigvee_{i=1}^{k} \neg p_x \wedge x = e \wedge q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} \neg p_x \wedge x = e \wedge q_{cj} \wedge \bigcirc q_{fj} \\
\neg p_x \wedge x = e \wedge q \equiv\ & \bigvee_{i=1}^{k} \neg p_x \wedge x = e \wedge q_{ei} \wedge empty \vee \bigvee_{j=1}^{h} \neg p_x \wedge x = e \wedge q_{cj} \wedge \bigcirc q_{fj}
\end{aligned}$$

Let σ be a model and r an integer, $0 \leq r \preceq |\sigma|$. Suppose $(\sigma, 0, r, |\sigma|) \models_m \neg p_x \wedge x = e \wedge q$. Then, $(\sigma, 0, r, |\sigma|) \models \neg p_x$, $(\sigma, 0, r, |\sigma|) \models x = e$ and $(\sigma, 0, r, |\sigma|) \models q$. We claim $(\sigma, 0, r, |\sigma|) \models_m (p_x \vee \neg p_x \wedge x = e) \wedge q$.

If $(\sigma, 0, r, |\sigma|) \not\models_m (p_x \vee \neg p_x \wedge x = e) \wedge q$, then there is σ' such that $(\sigma', 0, r, |\sigma'|) \models (p_x \wedge q \vee \neg p_x \wedge x = e \wedge q)$ and $\sigma' \sqsubset \sigma$. Thus, $(\sigma', 0, r, |\sigma'|) \models p_x \wedge q$ or $(\sigma', 0, r, |\sigma|) \models \neg p_x \wedge x = e \wedge q$. Since $(\sigma, 0, r, |\sigma|) \models_m \neg p_x \wedge x = e \wedge q$, it follows that $(\sigma', 0, r, |\sigma'|) \not\models \neg p_x \wedge x = e \wedge q$ and $\sigma' \sqsubset \sigma$. Hence, $(\sigma', 0, r, |\sigma'| \models p_x \wedge q$ leading to $(\sigma', 0, r, |\sigma'|) \models p_x$. However, $\sigma' \sqsubset \sigma$ and $(\sigma, 0, r, |\sigma|) \models \neg p_x$. So, $(\sigma', 0, r, |\sigma'|) \models \neg p_x$. This is a contradiction. Hence, $(\sigma, 0, r, |\sigma|) \models_m (p_x \vee \neg p_x \wedge x = e) \wedge q$.

Conversely, suppose $(\sigma, 0, r, |\sigma|) \models_m (p_x \vee \neg p_x \wedge x = e) \wedge q$. By MIN-EQ, $(\sigma, 0, r, |\sigma|) \models_m p_x \wedge q \vee \neg p_x \wedge x = e \wedge q$. Then, we claim $(\sigma, 0, r, |\sigma|) \models_m \neg p_x \wedge x = e \wedge q$.

Suppose $(\sigma, 0, r, |\sigma|) \models_m p_x \wedge q$. By MIN-EQ and MIN-OR, $(\sigma, 0, r, |\sigma|) \models_m \bigvee_{i=1}^{k} p_x \wedge q_{ei} \wedge empty$ or $(\sigma, 0, r, |\sigma|) \models_m \bigvee_{i=1}^{k} p_x \wedge q_{cj} \wedge \bigcirc q_{fj}$.

If $(\sigma, 0, r, |\sigma|) \models_m \bigvee_{i=1}^{k} p_x \wedge q_{ei} \wedge empty$, then we can construct a σ' from σ by a single change ($r = |\sigma|$), namely,

$$I_p'^{|\sigma|} = I_p^{|\sigma|} - \{p_x\} \text{ and } I_v'^{|\sigma|}[x] = e$$

Since p_x and $x = e' \neq e$ are not contained in q_{ei} $(1 \leq i \leq k)$, $(\sigma', 0, r, |\sigma'|) \models \bigvee_{i=1}^{k} \neg p_x \wedge x = e \wedge q_{ei} \wedge empty$ leading to $(\sigma', 0, r, |\sigma'|) \models \neg p_x \wedge x = e \wedge q$. So, $\sigma' \sqsubset \sigma$ and $(\sigma', 0, r, |\sigma'|) \models (p_x \vee \neg p_x \wedge x = e) \wedge q$. This contradicts $(\sigma, 0, r, |\sigma|) \models_m (p_x \vee \neg p_x \wedge x = e) \wedge q$.

On the other hand, if $(\sigma, 0, r, |\sigma|) \models_m \bigvee_{j=1}^{h} p_x \wedge q_{cj} \wedge \bigcirc q_{fj}$ then we can construct a σ' from σ by a single change, namely,

$$I_p'^{r} = I_p^{r} - \{p_x\} \text{ and } I_{j=1}'^{r}[x] = e$$

Since p_x and $x = e' \neq e$ are not contained in q_{cj} $(1 \leq j \leq h)$, and for all $1 \leq j \leq h$, $\bigcirc q_{fj}$ are lec-formulas and q_{fj} are internal programs, q_{fj} do not refer to p_x at the current state, and $I_v'^{r}[x] = e$ does not affect the evaluation of q_{fj}. Thus, $(\sigma', 0, r, |\sigma'|) \models \bigvee_{j=1}^{h} \neg p_x \wedge x = e \wedge q_{cj} \wedge \bigcirc q_{fj}$ leading to $(\sigma', 0, r, |\sigma'|) \models \neg p_x \wedge x = e \wedge q$. So, $\sigma' \sqsubset \sigma$ and $(\sigma', 0, r, |\sigma'|) \models (p_x \vee \neg p_x \wedge x = e) \wedge q$. This contradicts $(\sigma, 0, r, |\sigma|) \models_m (p_x \vee \neg p_x \wedge x = e) \wedge q$.

Therefore, by Lemma 7.5, $(\sigma, 0, r, |\sigma|) \models_m \neg p_x \wedge x = e \wedge q$.

□

We have proved several useful logic laws concerned with the substitution law within framed programs under the minimal model. Actually, there are some very simple heuristics to follow in the reduction of a framed program under the minimal model using the Tableau method. These are as follows:

1. use the relation $\equiv$ as far as possible during a reduction;
2. do not use the minimal model to obtain a $\neg p_x$ for a dynamic variable x until the last stage of a reduction, and make sure that x has really not been assigned a value by $\Leftarrow$ within all the conjuncts at the current state.

A framed program p also has its normal form under the minimal model, which is similar to the normal form defined in (6.1).

Theorem 7.9 If p is a framed program, then there is a program q as defined in Definition 6.5 with $l = m$ such that

$$p \equiv_m q$$

□

Proof

By Theorem 6.4 and Theorem 7.4, we need to prove only $l = m$ in the Definition 6.5. That is, in q_{ei} $(i \leq k)$ and q_{cj} $(j \leq h)$, the number of the equalities and the number of propositions are equal. We justify this fact as follows:

Since q_{ei} and q_{cj} are state formulas, three types of assignments, $x_i \Leftarrow e_i$, $x_i = e_i$ and $lbf(x_i)$ need to be considered. $x_i \Leftarrow e_i$ is simply reduced to $x_i = e_i \wedge p_{x_i}$ under the minimal model. $x_i = e_i \equiv x_i = e_i \wedge p_{x_i} \vee x_i = e_i \wedge \neg p_{x_i}$ is reduced to $x_i = e_i \wedge \neg p_{x_i}$ under the minimal model (Theorem 7.6). $lbf(x_i) \equiv (p_{x_i} \vee \neg p_{x_i} \wedge x_i = \bigodot x_i)$ is reduced to $\neg p_{x_i} \wedge x_i = e_i$ under the minimal model (Theorem 7.8 and with the assumption $\bigodot x_i = e_i$). Thus, we can see $x_i = e_i$ and p_{x_i} (or $\neg p_{x_i}$) always occur together in q_{ei} and q_{cj}.

□

If a program is deterministic under the minimal model, its normal form has a simpler form, as follows:

$$p \equiv_m p_e \wedge empty \quad \text{or} \quad p_c \wedge \bigcirc p_f$$

where p_e, p_c and p_f are defined as in (6.1) of Definition 6.5.

By Theorem 7.9, if a framed deterministic program p terminates, it can be reduced to a sequence of state formulas, $p_c^0, ..., p_c^n$, where p_c^i corresponds to state s_i. Thus, we can define a calculus var to select assigned variables as follows:

Algorithm AX (selecting the set of assigned variables)

- $var(true) = \phi$,
- $var(x = e) = \phi$,
- $var(\neg p_{x_i}) = \phi$,
- $var(p_{x_i}) = \{x_i\}$,
- $var(w_1 \wedge w_2) = var(w_1) \cup var(w_2)$

where e is a state term, and w_1 and w_2 are state formulas. Then $X^i = var(p_c^i)$ is called a set of positively assigned variables of program p at state s_i.

Let $\sigma_x =< X^0, ..., >$ be a sequence of the sets of the positively assigned variables of program p as defined in the above. Let $\sigma_p^a =< P^0, ..., >$ be the sequence of the sets of the assigned propositions with respect to the sequence σ_x, i.e. $P^i \overset{\text{def}}{=} \{p_{x_i} | x_i \in X^i\}$ for every $i, i \geq 0$. These notations will be used henceforth.

Finally, we claim that Definitions 6.1, 6.2, 6.3, 6.4 and 6.8 satisfy Conditions 1, 2 and 3 under the minimal model on propositions. Condition 1 is simply satisfied by Definition 6.2; and Condition 2 and Condition 3 are satisfied by these definitions, as stated in Theorem 7.10.

Theorem 7.10 Let Y be a variable denoting the set of positively assigned variables within program p at each state over a model of p, and $R = M_m(p)$. Then

(1) $\models_R \Box(af(x_i) \leftrightarrow x_i \in Y)$
(2) $\models_R frame(x_i) \leftrightarrow \Box(x_i \notin Y \wedge \neg first \rightarrow \exists b : \bigcirc x_i = b \wedge x_i = b)$

Proof

First, we claim that, if $\sigma \in M_m(p)$ then $\sigma \models_m p$. This fact can be proved in a way similar to the proof of Theorem 4.1. Let $\sigma =< (I_v^0, I_p^0), ..., (I_v^j, I_p^j), ... > \in M_m(p)$, and k an integer $0 \leq k \preceq |\sigma|$. Since $\sigma \in R = M_m(p)$, $\sigma \models_m p$.

The proof of (1)

Suppose $(\sigma, 0, k, |\sigma|) \models_R x_i \in Y$, i.e. $x_i \in X^k$ at state s_k. Since $\sigma \models_m p$, according to the algorithm AX given in Section 7.1, there exists e such that $x_i \Leftarrow e$ at state s_k. That is, $(\sigma, 0, k, |\sigma|) \models_R x_i \Leftarrow e$. This leads to $(\sigma, 0, k, |\sigma|) \models_R af(x_i)$.

Conversely, suppose $(\sigma, 0, k, |\sigma|) \models_R af(x_i)$. That is, by Definition 6.2, $(\sigma, 0, k, |\sigma|) \models_R p_{x_i}$. Hence, $p_{x_i} \in I_p^k$ at state s_k according to the interpretation. Thus, there is an assignment $x_i \Leftarrow e$ $(e \in D)$ at state s_k (Theorem 7.9). Hence, $p_{x_i} \in P^k$ at state s_k , and $x_i \in Y$ at state s_k leading to $(\sigma, 0, k, |\sigma|) \models_R x_i \in Y$.

The proof of (2)

	$(\sigma, 0, k, \lvert\sigma\rvert) \models_R frame(x_i)$	
$\Longleftrightarrow$	$(\sigma, 0, k, \lvert\sigma\rvert) \models_R \Box(\neg first \rightarrow lbf(x))$	corollary 7.18
$\Longleftrightarrow$	$(\sigma, 0, k, \lvert\sigma\rvert) \models_R \Box(\neg first \rightarrow (\neg af(x_i) \rightarrow \exists b : \bigcirc x_i = b \wedge x_i = b))$	definition 6.3
$\Longleftrightarrow$	$(\sigma, 0, k, \lvert\sigma\rvert) \models_R \Box(\neg af(x_i) \wedge \neg first \rightarrow \exists b : \bigcirc x_i = b \wedge x_i = b)$	theorem 3.7
$\Longleftrightarrow$	$(\sigma, 0, k, \lvert\sigma\rvert) \models_R \Box(x_i \notin Y \wedge \neg first \rightarrow \exists b : \bigcirc x_i = b \wedge x_i = b)$	theorem 7.10 (1)

□

So far different models have been introduced to capture semantics of logic formulas and programs under different assumptions. In the logic (EITL), the basic model and the satisfaction relation ($\models$) are used to interpret terms and formulas. Within the extended Tempura, the canonical model and the satisfaction relation ($\models_c$) are employed to interpret expressions and programs. To capture the intended meaning of a framed program in the extended Tempura, the minimal model and the satisfaction relation ($\models_m$) are introduced. This model enables us to catch those variables which are framed, but not assigned new values by the positive immediate assignments so that their values can be inherited. The minimal model takes effect by means of perceiving the defaults of positive immediate assignments.

In addition to the above models, the P-model relevant to a program P is needed to express some conditions relative to the program P since the conditions can be expressed by means of formulas of EITL rather than in the extended Tempura. A P-model can also be connected with a minimal model in addition to with a canonical model.

Theorems 7.1 - 7.10 show us that the temporal semantics of a framed program can be well captured by the minimal model. For easy manipulation, we need further to investigate the properties of framing operators.

7.2 Algebraic Properties of Framing Operators

The framing operators enjoy some interesting properties such as equivalent, distributive, absorptive, and idempotent laws etc. They are formalized in Theorems (Corollaries) 7.11 - 7.18. Note that these theorems are established on the basic model and are independent of concrete programs.

Theorem 7.11 $frame(x) \equiv frame'(x)$

Proof

$$\begin{array}{lll}
 & frame(x) & \\
\equiv & \Box(more \rightarrow \bigcirc lbf(x)) & \text{definition 6.3} \\
\equiv & \Box(more \rightarrow \bigcirc(\neg af(x) \rightarrow (\exists b : \ominus x = b \wedge x = b))) & \text{definition 6.3} \\
\equiv & \Box(more \rightarrow (\odot \neg af(x) \rightarrow \bigcirc (\exists b : \ominus x = b \wedge x = b))) & \text{FD5} \\
\equiv & \Box(more \rightarrow (\odot \neg af(x) \rightarrow \exists b : \bigcirc(\ominus x = b \wedge x = b))) & \text{theorem 3.8 1} \\
\equiv & \Box(more \rightarrow (\odot \neg af(x) \rightarrow \exists b : \bigcirc(\ominus x = b) \wedge \bigcirc (x = b))) & \text{FD3} \\
\equiv & \Box(more \rightarrow (\odot \neg af(x) \rightarrow \exists b : (\bigcirc \ominus x = \bigcirc b) \wedge (\bigcirc x = \bigcirc b))) & \text{theorem 3.13} \\
\equiv & \Box(more \rightarrow (\odot \neg af(x) \rightarrow \exists b : x = b \wedge \bigcirc x = b)) & \text{lemma 3.17, theorem 3.13} \\
\equiv & \Box(more \wedge \odot \neg af(x) \rightarrow \exists b : x = b \wedge \bigcirc x = b) & \\
\equiv & \Box(more \wedge \bigcirc \neg af(x) \rightarrow \exists b : x = b \wedge \bigcirc x = b) & \text{FW1} \\
\equiv & \Box(more \rightarrow (\bigcirc \neg af(x) \rightarrow \exists b : x = b \wedge \bigcirc x = b)) & \\
\equiv & frame'(x) & \text{definition 6.4}
\end{array}$$

□

Theorem 7.11 asserts that two operators, *frame* and *frame'*, are equivalent. Hence, in what follows, we do not distinguish between them.

Theorem 7.12 The following formulas hold

$$\begin{array}{lll}
(1) & frame(x) \wedge frame(x) & \equiv & frame(x) \\
(2) & frame(x) \wedge more & \equiv & \bigcirc(lbf(x) \wedge frame(x)) \\
(3) & frame(x) \wedge more & \equiv & lff(x) \wedge \bigcirc frame(x) \\
(4) & frame(x) \wedge empty & \equiv & empty
\end{array}$$

Proof

We prove only (2).

$$\begin{array}{lll}
 & frame(x) \land more & \\
\equiv & \Box(more \rightarrow \bigcirc lbf(x)) \land more & \text{definition 6.3} \\
\equiv & (more \rightarrow \bigcirc lbf(x)) \land \odot\Box(more \rightarrow \bigcirc lbf(x)) \land more & \text{FE2} \\
\equiv & \bigcirc lbf(x) \land more \land \bigcirc\Box(more \rightarrow \bigcirc lbf(x)) & \text{FW1} \\
\equiv & \bigcirc lbf(x) \land more \land \bigcirc frame(x) & \text{definition 6.3} \\
\equiv & \bigcirc(lbf(x) \land frame(x)) \land more & \text{FD3} \\
\equiv & \bigcirc(lbf(x) \land frame(x)) & \text{FS4}
\end{array}$$

□

In Theorem 7.12, (1) describes the idempotent law; (2) and (3) describe the properties of the framing operator at non-terminating states over an interval; whereas, (4) describes the property of the framing operator at a terminating state.

Theorem 7.13 The following formulas hold

$$\begin{array}{lll}
(1) & frame(x) \land (frame(x) \land p \lor q) & \equiv \; frame(x) \land (p \lor q) \\
(2) & frame(x) \land (p \lor frame(x) \land q) & \equiv \; frame(x) \land (p \lor q) \\
(3) & frame(x) \land (frame(x) \land p \lor frame(x) \land q) & \equiv \; frame(x) \land (p \lor q) \\
(4) & (frame(x) \land p \lor frame(x) \land q) & \equiv \; frame(x) \land (p \lor q)
\end{array}$$

This set of laws is concerned with disjunction. The first three are absorptive laws, the fourth is the distributive law.

Proof

We prove only (4); and the others can be easily proved by (4) and Theorem 7.12 (1). Let σ be an interval, and k an integer, $0 \leq k \preceq |\sigma|$. First, by abbreviation of $p \lor q$, the following fact is obvious.

$$(\sigma, 0, k, |\sigma|) \models p \lor q \iff (\sigma, 0, k, |\sigma|) \models p \text{ or } (\sigma, 0, k, |\sigma|) \models q \qquad (7.1)$$

Thus,

$$\begin{array}{lll}
 & (\sigma, 0, k, |\sigma|) \models frame(x) \land (p \lor q) & \\
\iff & (\sigma, 0, k, |\sigma|) \models frame(x) \text{ and } (\sigma, 0, k, |\sigma|) \models p \lor q & \text{I-and} \\
\iff & (\sigma, 0, k, |\sigma|) \models frame(x) \text{ and } ((\sigma, 0, k, |\sigma|) \models p \text{ or } (\sigma, 0, k, |\sigma|) \models q) & (7.1) \\
\iff & (\sigma, 0, k, |\sigma|) \models frame(x) \text{ and } (\sigma, 0, k, |\sigma|) \models p & \\
 & \text{ or } (\sigma, 0, k, |\sigma|) \models frame(x) \text{ and } (\sigma, 0, k, |\sigma|) \models q & \\
\iff & (\sigma, 0, k, |\sigma|) \models frame(x) \land p \text{ or } (\sigma, 0, k, |\sigma) \models frame(x) \land q & \text{I-and} \\
\iff & (\sigma, 0, k, |\sigma|) \models frame(x) \land p \lor frame(x) \land q & (7.1)
\end{array}$$

□

Theorem 7.15 is concerned with chop(;) operator. (1), (2) and (3) state the absorptive law; (4) states distributive law; and (5) states idempotent law. To prove them, we need first to prove Lemma 7.14.

Lemma 7.14 Let σ be an interval, and k, r integers, $0 \leq k \leq r \preceq |\sigma|$. Then

$$(\sigma, 0, k, |\sigma|) \models frame(x) \Longrightarrow (\sigma, 0, k, r) \models frame(x)$$

Proof

Let $(\sigma,0,k,|\sigma|) \models frame(x)$. Suppose $(\sigma,0,k,r) \not\models frame(x)$ for some $r, k\le r\preceq|\sigma|$. Then we have

$$
\begin{array}{lll}
 & (\sigma,0,k,r)\models\neg frame(x) & \text{I-not} \\
\Longleftrightarrow & (\sigma,0,k,r)\models\Diamond\neg(more\rightarrow lff(x)) & \text{definition 6.3} \\
\Longleftrightarrow & (\sigma,0,k,r)\models\Diamond\neg(empty\vee\bigcirc\neg af(x)\rightarrow\exists b(x=b\wedge\bigcirc x=b))) & \text{definition 6.3} \\
\Longleftrightarrow & (\sigma,0,k,r)\models\Diamond(more\wedge\bigcirc\neg af(x)\wedge\neg(\exists b(x=b\wedge\bigcirc x=b))) & \\
\Longleftrightarrow & (\sigma,0,l,r)\models more\wedge\bigcirc\neg af(x)\wedge\neg(\exists b(x=b\wedge\bigcirc x=b)) \text{ for some } l, k\le l\le r & \text{abb-som} \\
\Longleftrightarrow & (\sigma,0,l,r)\models more \text{ and} & \\
 & (\sigma,0,l,r)\models\bigcirc\neg af(x) \text{ and} & \\
 & (\sigma,0,l,r)\models\neg(\exists bx=b\wedge\bigcirc x=b) \text{ for some } l,r,\, k\le l\le r\preceq|\sigma| & \text{I-and}
\end{array}
$$

Note that the following facts are obvious.

$$
\begin{array}{ll}
(\sigma,0,l,r)\models more\Longrightarrow(\sigma,0,l,|\sigma|)\models more & (7.2) \\
(\sigma,0,l,r)\models\bigcirc\neg af(x)\Longrightarrow(\sigma,0,l,|\sigma|)\models\bigcirc\neg af(x) & (7.3) \\
(\sigma,0,l,r)\models\neg(\exists bx=b\wedge\bigcirc x=b)\Longrightarrow(\sigma,0,l,r)\models s_l[x]\neq s_{l+1}[x] & (7.4)
\end{array}
$$

On the other hand, we have

$$
\begin{array}{llll}
 & (\sigma,0,k,|\sigma|)\models frame(x) & & \text{premise} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models\Box(more\rightarrow lff(x)) & & \text{definition 6.3} \\
\Longleftrightarrow & (\sigma,0,h,|\sigma|)\models more\rightarrow lff(x) \text{ for every } h, k\le h\preceq|\sigma| & & \text{abb-alw} \\
\Longrightarrow & (\sigma,0,l,|\sigma|)\models more\rightarrow lff(x) \text{ let } h=l & (7.5) & \\
\Longrightarrow & (\sigma,0,l,|\sigma|)\models lff(x) & & (7.2), (7.5) \\
\Longleftrightarrow & (\sigma,0,l,|\sigma|)\models\bigcirc af(x)\rightarrow\exists b(x=b\wedge\bigcirc x=b) & (7.6) & \text{definition 6.3} \\
\Longrightarrow & (\sigma,0,l,|\sigma|)\models\exists b(x=b\wedge\bigcirc x=b) & & (7.3), (7.6) \\
\Longrightarrow & (\sigma,0,l,|\sigma|)\models s_l[x]=s_{l+1}[x] & (7.7) & \\
\Longrightarrow & (\sigma,0,l,|\sigma|)\models false & & (7.4), (7.7)
\end{array}
$$

Thus, we achieve,

$$(\sigma,0,k,r)\models frame(x)$$

$\Box$

In general, $(\sigma,0,k,|\sigma|)\models\Box p$ does not imply $(\sigma,0,k,r)\models\Box p$ $(k<r\preceq|\sigma|)$. For instance, given an infinite interval σ, $(\sigma,0,k,|\sigma|)\models\Box more$ but $(\sigma,0,k,r)\not\models\Box more$ if $r<|\sigma|$. Even for a finite interval σ, $(\sigma,0,k,|\sigma|)\models\Box(empty\rightarrow x=10)$ can hold, but $(\sigma,0,k,r)\models\Box(empty\rightarrow x=10)$ may not hold at all because x can be 10 at the final state but may never be 10 at any internal state over the interval.

Theorem 7.15 The following formulas hold

$$
\begin{array}{lll}
(1) & frame(x)\wedge(frame(x)\wedge p;q) & \equiv\ frame(x)\wedge(p;q) \\
(2) & frame(x)\wedge(p;frame(x)\wedge q) & \equiv\ frame(x)\wedge(p;q) \\
(3) & frame(x)\wedge(frame(x)\wedge p;frame(x)\wedge q) & \equiv\ frame(x)\wedge(p;q) \\
(4) & (frame(x)\wedge p;frame(x)\wedge q) & \equiv\ frame(x)\wedge(p;q) \\
(5) & frame(x);frame(x) & \equiv\ frame(x)
\end{array}
$$

Proof

We prove only (1); and the others can be proved in the similar way. Let σ be a model, and k an integer, $0 \le k \preceq |\sigma|$.

$$\begin{array}{lll}
 & (\sigma,0,k,|\sigma|)\models frame(x)\land(frame(x)\land p;q) & \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,|\sigma|)\models(frame(x)\land p;q) & \text{I-and} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,r)\models frame(x)\land p \text{ and} & \\
 & (\sigma,r,r,|\sigma|)\models q \text{ for some } r, k\le r\le|\sigma| & \text{I-chop} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,r)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,r)\models p \text{ and} & \\
 & (\sigma,r,r,|\sigma|)\models q \text{ for some } k\le r\preceq|\sigma| & \text{I-and} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,r)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,|\sigma|)\models p;q & \text{I-chop} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x) \text{ and} & \\
 & (\sigma,0,k,|\sigma|)\models p;q & \text{lemma 7.14} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x)\land(p;q) & \text{I-and}
\end{array}$$

□

Theorem 7.16 states some similar laws as Theorem 7.10 with respect to the parallel operator.

Theorem 7.16 The following formulas hold

$$\begin{array}{lll}
(1) & frame(x)\land(frame(x)\land p\|q) & \equiv frame(x)\land(p\|q) \\
(2) & frame(x)\land(p\|frame(x)\land q) & \equiv frame(x)\land(p\|q) \\
(3) & frame(x)\land(frame(x)\land p\|frame(x)\land q) & \equiv frame(x)\land(p\|q) \\
(4) & (frame(x)\land p\|frame(x)\land q) & \equiv frame(x)\land(p\|q) \\
(5) & frame(x)\|frame(x) & \equiv frame(x)
\end{array}$$

Proof

We prove only (1); and the others can be proved analogously. Let σ be an interval, and k an integer, $0 \le k \preceq |\sigma|$.

$$\begin{array}{lll}
 & (\sigma,0,k,|\sigma|)\models frame(x)\land(frame(x)\land p\|q) & \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x)\land((frame(x)\land p;true)\land q)\lor frame(x)\land p\land(q;true)) & \text{definition} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x)\land q\land(frame(x)\land p;true)\lor frame(x)\land p\land(q;true)) & \text{th 7.13(4), 7.12(1)} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x)\land q\land(p;true)\lor frame(x)\land p\land(q;true)) & \text{th7.15 (1)} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x)\land(q\land(p;true)\lor p\land(q;true)) & \text{th7.13 (4)} \\
\Longleftrightarrow & (\sigma,0,k,|\sigma|)\models frame(x)\land(p\|q) & \text{definition}
\end{array}$$

□

Theorem 7.17 is concerned with lbf and lff. Its proof is straightforward.

Theorem 7.17 Let σ be an interval, $|\sigma| > 0$, then

(1) $\sigma\models frame(x) \Longleftrightarrow (\sigma,0,i,|\sigma|)\models lbf(x)$ for all $0 < i\preceq|\sigma|$
(2) $\sigma\models frame(x) \Longleftrightarrow (\sigma,0,i,|\sigma|)\models lff(x)$ for all $0\le i < |\sigma|$

By Theorem 7.17, the *frame* operator can be expressed by the previous operator as follows.

Corollary 7.18

$$\begin{aligned}
&1\quad frame(x) \equiv \Box(\neg first \rightarrow lbf(x))\\
&2\quad frame(x) \equiv \Box(\neg first \rightarrow \ominus lff(x))
\end{aligned}$$

□

Theorems (or Corollaries) 7.11 - 7.18 play an important role in temporal logic programming within our system. They enable us to reduce a program in a convenient way. Many reduction rules of the interpreter developed by us recently are based on these theorems.

7.3 Example

In this section, an example is given to show how to apply the algebraic properties together with the minimal model to interpret a framed program. The example also illustrates how to reduce a program in a way in which Theorem 7.9 operates. By such means, we hope that the reader will agree that the temporal semantics of a framed program is well captured. The program considered is as follows

$$p \equiv frame(x_1) \wedge (x_1 = 1 \wedge x_2 = 2 \wedge len(2); x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2)$$

The following is a complete reduction process of program p.

$$\begin{aligned}
p \;\equiv\;& frame(x_1) \wedge (x_1 = 1 \wedge x_2 = 2 \wedge len(2); x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2)\\
\equiv\;& frame(x_1) \wedge x_1 = 1 \wedge x_2 = 2 \wedge len(2); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge\\
& x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2 && \text{theorem 7.15}\\
\equiv\;& x_1 = 1 \wedge x_2 = 2 \wedge frame(x_1) \wedge \bigcirc len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2)\\
& \wedge x_1 :=^{+} x_1 + \bigcirc x_2 && \text{Abb-len}\\
\equiv\;& x_1 = 1 \wedge x_2 = 2 \wedge frame(x_1) \wedge more \wedge \bigcirc len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2)\\
& \wedge x_1 :=^{+} x_1 + \bigcirc x_2 && \text{FS4}\\
\equiv\;& x_1 = 1 \wedge x_2 = 2 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1)) \wedge \bigcirc len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 2\\
& \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2 && \text{theorem 7.12 2}\\
\equiv\;& x_1 = 1 \wedge x_2 = 2 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1) \wedge len(1)); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3\\
& \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2 && \text{FD3}\\
\equiv\;& x_1 = 1 \wedge x_2 = 2 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3\\
& \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2) && \text{FCH1}\\
\equiv\;& (p_{x_1} \wedge x_1 = 1 \vee \neg p_{x_1} \wedge x_1 = 1) \wedge (p_{x_2} \wedge x_2 = 2 \vee \neg p_{x_2} \wedge x = 2) \wedge\\
& \bigcirc(lbf(x_1) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2)\\
\equiv_m\;& \neg p_{x_1} \wedge x_1 = 1 \wedge \neg p_{x_2} \wedge x = 2 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3\\
& \wedge frame(x_2) \wedge x_1 :=^{+} x_1 + \bigcirc x_2) && \text{corollary 7.7}
\end{aligned}$$

The last formula shows that p is in a well-reduced (i.e. normal) form $p_c^0 \wedge \bigcirc p_f^0$ at state s_0.

Then p_f^0 is continuously re-reduced as follows:

$$\begin{aligned}
p_c^0 &\equiv x_1 = 1 \wedge x_2 = 2 \wedge \neg p_{x_1} \wedge \neg p_{x_2} \\
p_f^0 &\equiv lbf(x_1) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 \\
&\equiv (\neg af(x_1) \rightarrow \exists b : \ominus x_1 = b \wedge x_1 = b) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \\
&\quad \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{theorem 6.3} \\
&\equiv (p_{x_1} \vee \neg p_{x_1} \wedge x_1 = \ominus x_1) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \\
&\quad \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{theorem 6.3} \\
&\equiv (p_{x_1} \vee \neg p_{x_1} \wedge x_1 = 1) \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \\
&\equiv_m \neg p_{x_1} \wedge x_1 = 1 \wedge frame(x_1) \wedge len(1); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \\
&\quad \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{theorem 7.8} \\
&\equiv \neg p_{x_1} \wedge x_1 = 1 \wedge frame(x_1) \wedge more \wedge \bigcirc(empty); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \\
&\quad \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{FS4} \\
&\equiv \neg p_{x_1} \wedge x_1 = 1 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1)) \wedge \bigcirc(empty); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \\
&\quad \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{theorem 7.12 2} \\
&\equiv \neg p_{x_1} \wedge x_1 = 1 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1) \wedge empty); frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \\
&\quad \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{FD3} \\
&\equiv \neg p_{x_1} \wedge x_1 = 1 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1) \wedge empty; frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \\
&\quad \wedge x_1 :=^+ x_1 + \bigcirc x_2) && \text{FCH1}
\end{aligned}$$

At this point of time, p_f^0 is reduced to the normal form $p_c^1 \wedge \bigcirc p_f^1$, and p_f^1 is continuously re-reduced as follows:

$$\begin{aligned}
p_c^1 &\equiv \neg p_{x_1} \wedge x_1 = 1 \\
p_f^1 &\equiv lbf(x_1) \wedge frame(x_1) \wedge empty; frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 \\
&\equiv lbf(x_1) \wedge empty; frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{theorem 7.12 4} \\
&\equiv (\neg af(x_1) \rightarrow x_1 = 1) \wedge empty; frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 \\
& && \text{def 6.3} \\
&\equiv (\neg af(x_1) \rightarrow x_1 = 1) \wedge frame(x_1) \wedge x_1 \Leftarrow 9 \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 :=^+ x_1 + \bigcirc x_2 && \text{FEP5} \\
&\equiv frame(x_1) \wedge x_1 = 9 \wedge p_{x_1} \wedge x_2 = 3 \wedge frame(x_2) \wedge x_1 o\!=^+ 9 + \bigcirc x_2 \wedge skip \\
&\equiv x_1 = 9 \wedge p_{x_1} \wedge x_2 = 3 \wedge frame(x_1) \wedge frame(x_2) \wedge \bigcirc(x_1 \Leftarrow 9 + x_2) \wedge \bigcirc empty \\
& && \text{def 6.1, theorem 3.13, lemma3.17} \\
&\equiv x_1 = 9 \wedge p_{x_1} \wedge x_2 = 3 \wedge frame(x_1) \wedge frame(x_2) \wedge more \wedge \bigcirc(x_1 \Leftarrow 9 + x_2) \wedge \bigcirc empty && \text{FS4} \\
&\equiv x_1 = 9 \wedge p_{x_1} \wedge x_2 = 3 \wedge \bigcirc(lbf(x_1) \wedge frame(x_1)) \wedge \bigcirc(lbf(x_2) \wedge frame(x_2)) \wedge \bigcirc(x_1 \Leftarrow 9 + x_2) \\
&\quad \wedge \bigcirc empty && \text{theorem 7.12 2} \\
&\equiv x_1 = 9 \wedge p_{x_1} \wedge x_2 = 3 \wedge \bigcirc(lbf(x_1) \wedge lbf(x_2) \wedge frame(x_1) \wedge frame(x_2) \wedge x_1 \Leftarrow 9 + x_2 \wedge empty) \\
& && \text{FD3} \\
&\equiv x_1 = 9 \wedge p_{x_1} \wedge (\neg p_{x_2} \wedge x_2 = 3 \vee p_{x_2} \wedge x_2 = 3) \wedge \bigcirc(lbf(x_1) \wedge lbf(x_2) \wedge \\
&\quad frame(x_1) \wedge frame(x_2) \wedge x_1 \Leftarrow 9 + x_2 \wedge empty) \\
&\equiv_m x_1 = 9 \wedge p_{x_1} \wedge \neg p_{x_2} \wedge x_2 = 3 \wedge \bigcirc(lbf(x_1) \wedge lbf(x_2) \wedge frame(x_1) \wedge frame(x_2) \wedge x_1 \Leftarrow 9 + x_2 \wedge empty) \\
& && \text{corollary 7.7}
\end{aligned}$$

At state s_2, p_f^1 is reduced to the normal form $p_c^2 \wedge \bigcirc p_f^2$. Then p_f^2 is re-reduced again.

$$\begin{array}{lll}
p_c^2 & \equiv & x_1 = 9 \wedge x_2 = 3 \wedge p_{x_1} \wedge \neg p_{x_2} \\
p_f^2 & \equiv & lbf(x_1) \wedge lbf(x_2) \wedge frame(x_1) \wedge frame(x_2) \wedge x_1 \Leftarrow 9 + x_2 \wedge empty \\
 & \equiv & lbf(x_1) \wedge lbf(x_2) \wedge x_1 \Leftarrow 9 + x_2 \wedge empty \qquad \text{theorem 7.12 4} \\
 & \equiv & (\neg af(x_1) \rightarrow x_1 = \ominus x_1) \wedge (\neg af(x_2) \rightarrow x_2 = \ominus x_2) \wedge x_1 = 9 + x_2 \wedge p_{x_1} \wedge empty \qquad \text{def 6.2, 6.3} \\
 & \equiv & (p_{x_2} \vee \neg p_{x_2} \wedge x_2 = 3) \wedge x_1 = 9 + x_2 \wedge p_{x_1} \wedge empty \\
 & \equiv_m & \neg p_{x_2} \wedge x_2 = 3 \wedge x_1 = 9 + x_2 \wedge p_{x_1} \wedge empty \qquad \text{theorem 7.8} \\
 & \equiv & \neg p_{x_2} \wedge x_2 = 3 \wedge x_1 = 9 + x_2 \wedge p_{x_1} \wedge empty \\
 & \equiv & x_1 = 9 + 3 \wedge p_{x_1} \wedge \neg p_{x_2} \wedge x_2 = 3 \wedge empty \\
 & \equiv & x_1 = 12 \wedge p_{x_1} \wedge \neg p_{x_2} \wedge x_2 = 3 \wedge empty
\end{array}$$

Finally, p_f^2 is reduced to the form $p_e \equiv p_c^3 \wedge empty$, which indicates that the reduction process of p is successfully completed. Where

$$\begin{array}{lll}
P_c^3 & \equiv & x_1 = 12 \wedge p_1 \wedge x_2 = 3 \wedge p_{x_1} \wedge \neg p_{x_2} \\
P_f^3 & \equiv & empty
\end{array}$$

	t_0	t_1	t_2	t_3
	p_c^0	p_c^1	p_c^2	p_c^3
$af(x_1)$	f	f	t	t
$af(x_2)$	f	f	f	f
X^i	ϕ	ϕ	$\{x_1\}$	$\{x_1\}$
I_p	ϕ	ϕ	$\{p_{x_1}\}$	$\{p_{x_1}\}$
I_v	$\{x_1 : 1, x_2 : 2\}$	$\{x_1 : 1, x_2 : \_\}$	$\{x_1 : 9, x_2 : 3\}$	$\{x_1 : 12, x_2 : 3\}$

Fig. 7.1 The reduction record of program P

In the above example, variable x_1 is framed throughout the overall interval, while variable x_2 is framed only over the subinterval $< s_2, s_3 >$. Note that x_2 is unspecified at state s_1, so x_2 is unavailable (i.e. unaccessible) at that state.

7.4 Discussion

In this chapter, the temporal semantics of framed programs is captured using the minimal model. This model allows us to treat variables which are framed and not assigned new values by the positive immediate assignments in such a way that their values are inherited. The minimal model does it by means of perceiving the defaults of positive immediate assignments.

It should be emphasised that by using the minimal model, the underlying logic is changed from being monotonic to non-monotonic. In particular, the minimal model semantics has imposed a kind of default logic in which a special set of propositions within a program is considered as a domain in which the default technique is applied. This results in revisable reasoning in EITL.

Chapter 8

Communication and Synchronization

Summary: Using the framing operator, a synchronization operator, *await(c)*, is defined within EITL. Furthermore, a framed concurrent temporal logic programming language (FTLL) is presented. To illustrate how to use both the language, the await operator, and framing technique, some examples are given. In particular, a mutual exclusion problem, producer - consumer, is solved using FTLL.

Introducing the framing operator enables us not only to write concise programs but also to define *await* construct within the underlying logic. Hence, the synchronized communication can be handled in framed Tempura. Furthermore, with the help of the *await* operator, a more powerful framed concurrent programming language, FTLL, can be designed under the EITL framework. This allows us to specify reactive systems at a higher level.

The chapter is organized as follows: Section 8.1 formalizes the *await* construct; Section 8.2 describes the framed concurrent programming language, FTLL; Section 8.3 provides two examples of programs, the first, constructing magic square, is concerned with framed arrays and non-deterministic programs; the second, modeling producer - consumer scheme, is devoted to solving the mutual exclusion problem. Finally, conclusions are drawn in Section 8.4.

8.1 Await Construct

A number of temporal logic programming languages, e.g. XYZ/E [81, 83], Tempura [61], TLA [53], employ logic conjunction ($\wedge$) as a basic parallel operator for concurrent computations. The communication between processes is based on shared variables. However, the conjunction construct seems appropriate for dealing with fine-grained parallel operations that proceed in lock-step since processes combined through the conjunction operator share all the states and may interfere with one another. The projection operator, $(p_1, ..., p_m)$ prj q, introduces a new computational model in which the process q is executed in parallel with $p_1; \ldots; p_m$ over an interval obtained by taking the endpoints (rendezvous points) of the intervals over which $p_1, \ldots, p_m$ are executed. Although the communication between processes is still based on shared variables, the communication and synchronization only take place at the rendezvous points (global states), otherwise they are executed independently.

However, with both the conjunction and the projection constructs, a problem that must be dealt with in temporal logic programming is that of synchronized communication between concurrent processes.

As discussed earlier, to synchronize communication between parallel processes in a concurrent program (e.g. when solving the mutual exclusion problem) with the shared variable model, a synchronization construct, *await*(c) or some equivalent is required, similarly as in many concurrent programming languages [66]. The *await*(c) does not change any variable, but waits until the condition c becomes true, at which point it terminates.

One may think *await*(c) could be defined as

$$await(c) \overset{\text{def}}{=} (c \rightarrow empty) \land (\neg c \rightarrow \bigcirc await(c))$$

which amounts to

$$await(c) \overset{\text{def}}{=} (c \land empty) \lor (\neg c \land \bigcirc await(c))$$

This is exactly the definition of *halt*(c) (see Theorem 4.9 in Chapter 4). However, *halt*(c) is capable of changing variables contained in c at the final state over an interval but *await*(c) is not although they both wait for c to become true and terminate the interval over which they act. The key difference between *await*(c) and *halt*(c) is that the former can only wait until another process acting in parallel changes c to true, while the latter can change c itself at the final state without the help of other processes acting in parallel.

Therefore, *halt*(c), in general, is not suitable as a synchronization construct for concurrent computations. However, if the variables contained in c all are framed, and no positive assignments appear in c (this condition is usually satisfied because we consider c as a condition, i.e. a boolean expression), *halt*(c) is equivalent to *await*(c). This is clear since the variables in c are framed and only positive assignments are able to change framed variables. For instance, $frame(x) \land halt(x = 1)$ is similar to $await(x = 1)$.

Defining *await*(c) is difficult without some kind of framing construct since the values of variables are not inherited automatically from one state to another. But one requires some kind of indefinite stability, since it cannot be known at the point of use how long the waiting will last. At the same time one must also allow variables to change, so that an external process can modify the boolean parameter and it can eventually become true.

To define await construct, we make the following assumption:

Let $V_c = \{x_1, ..., x_h\}$ be the set of dynamic variables contained in c. Then

Definition 8.1

$$\begin{array}{lcl} frame(V_c) & \overset{\text{def}}{=} & frame(x_1, ..., x_h) \\ frame(x_1, ..., x_h) & \overset{\text{def}}{=} & frame(x_1) \land ... \land frame(x_h) \end{array}$$

□

With the help of the framing operator, we can define *await* construct as follows:

Definition 8.2 (await statement)

$$await(c) \overset{\text{def}}{=} frame(V_c) \land halt(c)$$

where V_c represents all dynamic variables contained in c.

□

Since *frame*(x) and *halt*(c) both are executable within the extended Tempura, *await*(c) is also an executable construct in the extended Tempura. Therefore, synchronized communication for concurrent computations can be implemented in the extended Tempura. For instance, the following program synchronizes variables x and y in a parallel computation (see Figure 8.1).

Example 8.1 A synchronized computation.

```
$frame(x) \wedge frame(y) \wedge x = 0 \wedge y = 0 \wedge$
(( while $(x < 5)$ do (
            $x :=^{+} x + 1;$
            $await(y \geq x)$
            )
 )
 ||
 (while $(y < 5)$ do (
            $await(y < x);$
            $y :=^{+} y + 1$
            )
 )
).
```

```
 s0    s1     s2     s3     s4     s5     s6     s7     s8     s9    s10
 |------|------|------|------|------|------|------|------|------|------|
x=0     1 wait 1      2 wait 2      3 wait 3      4 wait 4      5 wait 5
 |------|------|------|------|------|------|------|------|------|------|
y=0 wait 0      1 wait 1      2 wait 2      3 wait 3      4 wait 4      5
```

Fig. 8.1 A Synchronized Computation

□

8.2 Framed Programming Language

The introduction of the framing operator and the minimal model semantics enable us to define *await* statement within the underlying logic. It is therefore interesting to point out that all statements of the concurrent programming language defined semi-formally in [66] can easily be defined formally within the extended logic framework except that the parallel computation is modeled, in the notation of [66], by interleaving as claimed, while in our notation, by true concurrency [49, 12] (see Example 8.1). In addition, our language contains the projection construct. Thus, a more powerful concurrent temporal logic programming language can be designed without obstacles. The following is a brief presentation of the language called Framed Temporal Logic programming Language (FTLL). We repeat all constructs from Chapter 4 and Chapter 5 and use the terminologies from [66].

- **Empty statement**

  The *empty* is a trivial do-nothing statement. It is defined as

$empty \overset{\text{def}}{=} \neg \bigcirc\, true$

Note that the *empty* statement has the same meaning as *skip* statement in [66]. We use *empty* rather than *skip* because *skip* has already been employed in the underlying logic with different meaning.

- **Assignment statement**

  $\overline{y} := \overline{e}$ is an assignment statement defined as follows

  $\overline{y} := \overline{e} \overset{\text{def}}{=} (y_1 :=^+ e_1) \wedge \ldots \wedge (y_n :=^+ e_n)$

  where $\overline{y} = (y_1, \ldots, y_n)$ is a list of variables and $\overline{e} = (e_1, \ldots, e_n)$ is a list of expressions. $:=^+$ is the positive unit assignment operator. The types of e_i and y_i are compatible.

- **Await statement**

  The $await(c)$ statement is defined by

  $await(c) \overset{\text{def}}{=} frame(V_c) \wedge halt(c)$

  where c is a boolean expression, and V_c denotes all the dynamic variables occurring in c.

- **Conditional statement**

  If s_1, s_2 are statements and c is a boolean expression, then the conditional statement is defined by

  $if\, c\; then\; s_1\; else\; s_2 \overset{\text{def}}{=} (c \rightarrow s_1) \wedge (\neg c \rightarrow s_2)$

- **Sequential statement**

  For statements s_1, s_2, the sequential statement can be expressed by *projection* operator directly.

  $s_1; s_2 \overset{\text{def}}{=} (s_1, s_2)\; prj\; empty$

- **When statement**

  With *sequential* statement and *await*, we can define *when* statement

  $when\; c\; do\; s \overset{\text{def}}{=} await(c); s$

- **Selection statement**

  For statements s_1, s_2, the selection statement can be defined directly by disjunction.

  $s_1\; or\; s_2 \overset{\text{def}}{=} s_1 \vee s_2$

  A multiple selection statement is defined as

  $OR_{k=1}^{n} s_k \overset{\text{def}}{=} s_1\; or\; s_2\; or\; \ldots or\; s_n$

- **Conditional selection statement**

  The guarded command in the language proposed by Dijkstra [17] of the form

  $if\ c_1 \rightarrow s_1 \Box c_2 \rightarrow s_2 \Box \ldots \Box c_n \rightarrow s_n fi$

  can be represented by a multiple selection statement formed out of several *when* statements:

  $OR_{k=1}^{n}(when\ c_k\ do\ s_k)$

- **While statement**

  While statement is defined as that in Chapter 4.

  $while\ c\ do\ p \overset{\text{def}}{=} (c \wedge p)^* \wedge fin(\neg c)$

- **Repeat statement**

  Repeat statement is defined as

  $repeat\ p\ until\ c \overset{\text{def}}{=} p\ ;\ while\ (\neg c)\ do\ p$

- **Conjunction statement**

  $s_1\ and\ s_2 \overset{\text{def}}{=} s_1 \wedge s_2$

- **Parallel Composition**

  Parallel composition statement is defined as:

  $s_1 || s_2 \overset{\text{def}}{=} s_1 \wedge (s_2; true) \vee s_2 \wedge (s_1; true) \vee \Box(more \wedge s_1 \wedge s_2)$

- **Projection statement**

  Projection statement is a primitive statement from the logic framework:

  $(s_1, ..., s_m)\ prj\ s$

  where $s_1, ..., s_m, s$ are statements.

- **Block statement**

  A block statement is of the form
  $local\ x : s \overset{\text{def}}{=} \exists x : s$ where x is a variable.

Comparing the framed Tempura and FTLL, we can see that most of the statements are the same except the disjunction construct and its derivativities such as choice statement. FTLL, therefore, is a non-deterministic programming language, and the normal form of a program in FTLL has to involve disjunctions. A program, in general, has its normal form as formalized in Theorem 8.1.

Theorem 8.1 If p is a framed program, then there is a program q as defined in (6.1) such that

$$p \equiv_m q$$

□

Proof Similar to the proof of Theorem 4.9.

□

Here we claim that the semantics of programs in FTLL can be captured by the minimal model under the model theory within EITL.

8.3 Programming in FTLL

In this section, two examples are given to illustrate how FTLL can be used.

8.3.1 Framed Arrays

Like variables, arrays (or lists) can also be framed. An array being framed means that all of its elements are framed. In temporal logic programming, a framed array is a useful construct. It leads to cleaner, more readable and efficient programs. To illustrate the framing technique for arrays, an algorithm for the construction of an even order magic square will be implemented. The example also shows how a non-deterministic program can operate.

Filling a magic square is an old mathematical puzzle. It requires one to fill in the consecutive positive integers $1, \ldots, n^2$ to an $n \times n$ array in such a way that the sums of the elements contained in each row, each column, and the two diagonals all are equal. For an odd number n, to fill a $n \times n$ magic square is simple. However, for an even n one requires a more subtle algorithm. The following algorithm presented in [19] can be used to construct a $4m \times 4m$ magic square.

Algorithm (constructing $4m \times 4m$ magic squares)
Let A be a $4m \times 4m$ array, and m be a positive integer.

1. Chop the sequence $1, \ldots, 16m^2$ into $4m$ subsequences of equal length, keeping the original order;
2. Insert the i^{th} subsequence into the i^{th} row of A according to the following rules:
   - when i is an odd number
     if $1 \leq i \leq 2m-1$, we fill in the numbers from the left to the right; if $2m+1 \leq i \leq 4m-1$, we fill in the numbers from the right to the left;
   - when i is an even number
     if $2 \leq i \leq 2m$, we fill in the numbers from the right to the left; if $2m+2 \leq i \leq 4m$, we fill in the numbers from the left to the right;
3. For each $i, 1 \leq i \leq 2m$, swap $2m$ pairs consisting of the i^{th} and the $(4m-i+1)^{th}$ row elements residing in the same column but not in the diagonal lines of A;

The above algorithm can be implemented by the following program with a framed array:

```
$(i,j) = (1,1)$ and $frame(i,j)$ and $frame(A)$ and
(
while $i \leq 4*m$ do                                {inserting $1,\ldots,16m^2$ into array $A$}
 (
 while $j \leq 4*m$ do
    (
    if $i \bmod 2 = 1$ then
       (if $1 \leq i \leq 2*m-1$ then $A[i,j] :=^+ (i-1)*4*m+j$
       else if $2*m+1 \leq i \leq 4*m-1$ then $A[i,j] :=^+ i*4*m-j+1$)
    else
      (if $2 \leq i \leq 2*m$ then $A[i,j] :=^+ i*4*m-j+1$
      else if $2*m+2 \leq i \leq 4*m$ then $A[i,j] :=^+ (i-1)*4*m+j$);
      $j :=^+ j+1$
    );
 $i :=^+ i+1$
 );

$(i,j) = (1,1)$ and $frame(i,j,c,temp)$ and
while $i \leq 2*m$ do                       {swapping $2m$ pairs of the elements of $A$}
 (
 $c :=^+ 0$;                                                          {c is a counter}
 while $j \leq 4*m$ do
    (
     if $(i \neq j$ and $(i+j) \leq 4*m$ and $c \leq 2*m)$ then     {the condition for swapping}
       (
       $temp :=^+ A[i,j]$;                                                     {swap}
       $A[i,j] :=^+ A[4*m-i+1,j]$;
       $A[4*m-i+1,j] :=^+ temp$;
       $c :=^+ c+1$
       );
     $j :=^+ j+1$
    );
 $i :=^+ i+1$
 )
)
```

Fig. 8.2 A program for constructing a $4m \times 4m$ magic square

In the program, the first two nested *while* statements insert $1,\ldots,16m^2$ into a $4m \times 4m$ array, as specified in the algorithm. The second two nested *while* statements swap $2m$ pairs of elements which do not reside on the diagonal lines. Each element $A[i,j]$ in the diagonal line from upper left to down right satisfies $i = j$, whereas each element $A[i,j]$ on the other diagonal from down left to upper right satisfies $i + j > 4m$; moreover, we only swap $2m$ pairs of elements for each $i, 1 \leq i \leq 2m$. Hence, the condition for swapping is $i \neq j \wedge (i+j) \leq 4m \wedge c \leq 2m$. Note that c is used as a counter to record the number of swapped pairs.

Example 8.2 Constructing a 4×4 magic square.
According to the above program ($m = 1$) the construction produces a square shown in Fig 8.3. It is easy to check its correctness, the sum is 34 for each row, column and diagonal line.

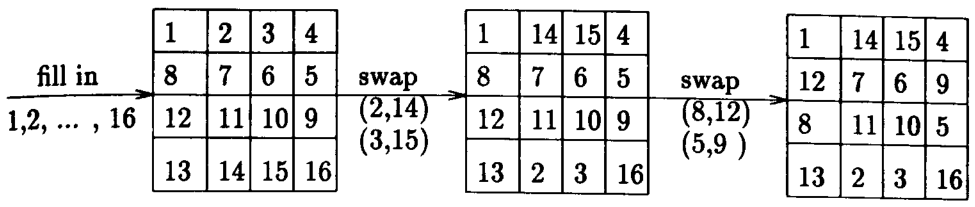


Fig. 8.3 Constructing a 4x4 magic square

□

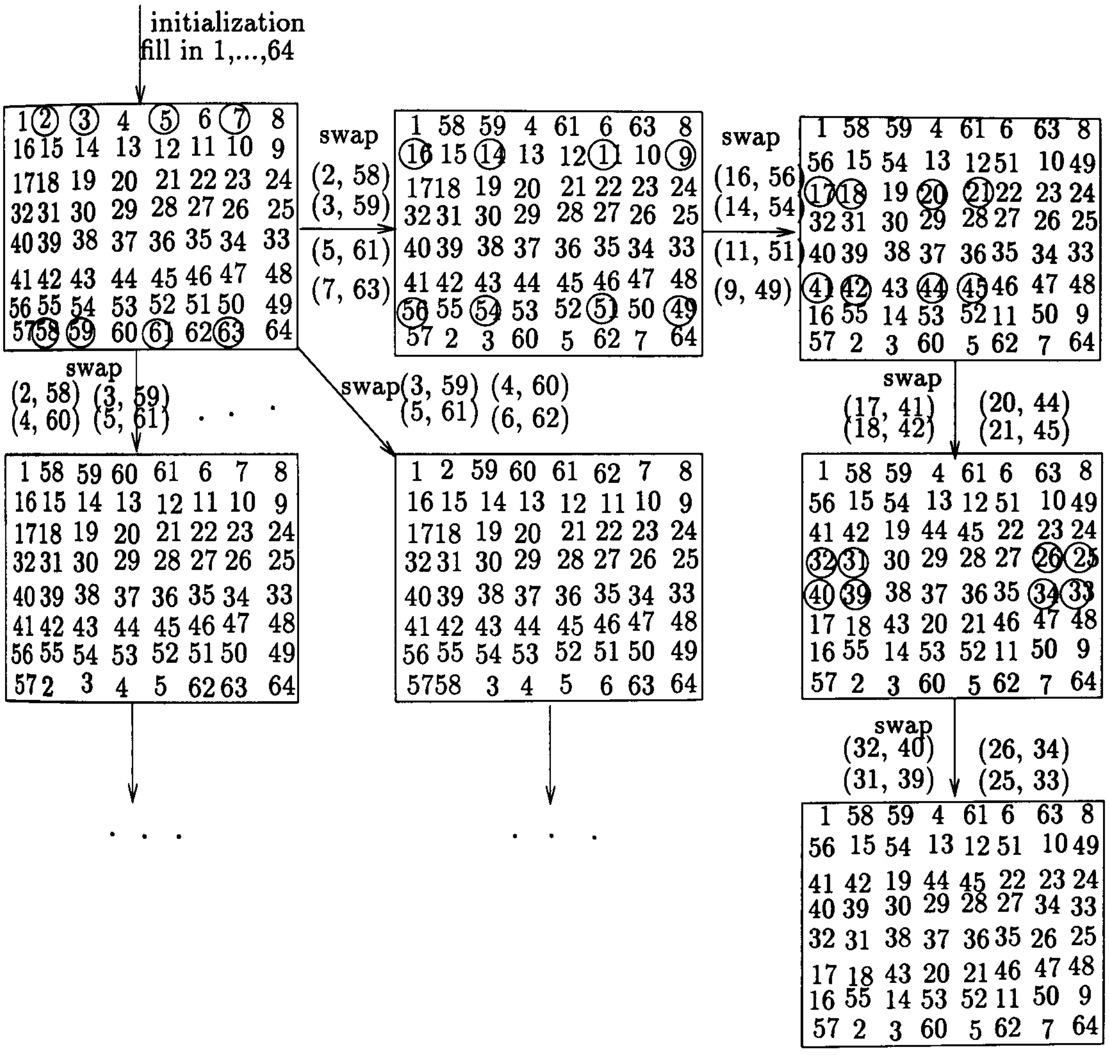


Fig. 8.4 Constructing 8x 8 magic squares

In the above example, the program is deterministic. However, according to the algorithm, step 3 requires $2m$ elements not residing at the diagonal lines in the i^{th} row be swapped with the corresponding elements in the $(4m - i + 1)^{th}$ row. Hence, each row has $4m - 2$ elements which are entitled to be chosen for swapping; but only $2m$ elements are needed to be swapped. If $m = 1$, i.e. for a 4×4 square, then $4m - 2 = 2m$; thus, all of elements entitled for swapping must be swapped. If $m \geq 2$ then $4m - 2 - 2m = 2m - 2 \geq 2$; so there are different choices of elements for swapping, leading to different squares.

The swapping part of the program can be rewritten as a non-deterministic program, as shown below:

```
$J = \{1, \ldots, m\}$;
while $c \leq 2 * m$ do
  ($OR_{j=1}^{m}$ when ($j \in J$ and $i \neq j$ and $i + j \leq 4 * m$)
              do($A[i,j] :=^{+} A[4 * m - i + 1, j]$
                and $A[4 * m - i + 1, j] :=^{+} A[i,j]$
                and $c :=^{+} c + 1$
                and $J :=^{+} J - \{j\}$
                )
  )
```

Using the non-deterministic program, the constructing process of a 8×8 magic squares is shown in Fig 8.4. Note that, in the non-deterministic program, J can be implemented by a list, and $j \in J$ can be implemented by a *while* statement.

8.3.2 Mutual Exclusion Problem

The mutual exclusion problem is one of the most important problems in concurrent programming. One solution to the problem is the Dekker's algorithm [6]. Below we show how it can be implemented in FTLL.

```
frame($c_1, c_2$, turn) and $c_1 = 1$ and $c_2 = 1$ and
turn = 1 and
(
repeat (                                {Process $P_1$ }
    Non − critical − section;
    $c_1 :=^{+} 0$;
    while not ($c_2 = 1$)do
        (
         if turn = 2 then
           ($c_1 :=^{+} 1$;
             await (turn = 1);
            $c_1 :=^{+} 0$;
           );
        );
    Critical − section − 1;
    $c_1 :=^{+} 1$;
    turn $:=^{+}$ 2;
    )
until empty
||
repeat(                                         {Process $P_2$ }
    Non - critical - section;
    $c_2 :=^+ 0$;
    while not $(c_1 = 1)$do
        (
        if turn = 1then
           ($c_2 :=^+ 1$;
             await (turn = 2);
            $c_2 :=^+ 0$;
           );
        );
    Critical - section - 2;
    $c_2 :=^+ 1$;
    turn $:=^+ 1$;
    )
until empty
).
```

Fig. 8.5 A program implementing Dekker's algorithm

In the above program, the individual variables in each process will ensure mutual exclusion, but upon detecting contention, a process, say P_1, will consult an additional global variable *turn* to see if it is its turn to insist upon entering its critical section. If not, it will reset c_1 and defer the choice to P_2, waiting on *turn*. When the P_2 completes its critical section, it will change *turn* to 1, freeing P_1. Even if P_2 immediately made other requests to enter the critical section, it will be blocked by *turn* once P_1 re-issued its request.

The program is correct [6]. It satisfies the mutual exclusion property, it does not deadlock, neither process can be starved and in the absence of contention a process can enter its critical section immediately.

8.3.3 Producer and Consumer

Using the Dekker's algorithm with the *await* operator, the producer - consumer problem can be solved by the program shown in Fig 8.6.

```
frame($c_1, c_2$, turn, x, s, ne, nf, buf) and $c_1 = 1$ and $c_2 = 1$ and
turn = 1 and ne = n and nf = 0 and buf = nil and
(
while true do                                   {Process $P_1$ }
   (
    $x :=^+ x + 1$;
    if $ne > 0$ then $c_1 :=^+ 0$;
    while $(c_2 = 0)$ do
        (
        if turn = 2 then
           ($c_1 :=^+ 1$;
             await (turn = 1);
```

```
            $c_1 :=^+ 0$;
          );
        );
    $buf :=^+ buf{\cdot}x$;                          {Critical section}
    $c_1 :=^+ 1$;
    $turn :=^+ 2$;
    )
||
while true do                                        {Process $P_2$ }
    (
    if $nf > 0$ then $c_2 :=^+ 0$;
    while $(c_1 = 0)$ do
        (
        if $turn = 1$ then
          ($c_2 :=^+ 1$;
            await $(turn = 2)$;
            $c_2 :=^+ 0$;
          );
        );
    $(y, buf) :=^+ (hd(buf), tl(buf))$;              {Critical section}
    $c_2 :=^+ 1$;
    $turn :=^+ 1$;
    )
).
```

Fig. 8.6 Producer-consumer

In the program, buf is an one-dimension array (or list) used as a buffer; ne, nf are dynamic variables denoting the number of free slots and the number of the items in the buffer respectively; and x, y, s are dynamic variables. c_1, c_2 and $turn$ are the variables serving for the mutually exclusive accesses to the buffer, the same as in Fig. 8.5.

The program presented in Fig.8.6 models a producer-consumer scheme. The producer computes a value and stores it in x; after successfully appending x to buf, it cycles back to complete the next value of x. The consumer, acting in parallel, removes an element from the head of the buffer buf and deposits it in y. After successfully obtaining such an element from buf, it proceeds to use the value for computations.

The buffer is represented by a list buf, whose initial value is the empty list, i.e. each element is *nil*. Adding an element to the end of buf is accomplished by the appending operation $buf{\cdot}x$. The head element of the buf is retrieved by the list function $hd(buf)$, and removal of this element from the buffer is accomplished by replacing buf with its tail $tl(buf)$. It is assumed that the maximal capacity of the buffer is $n > 0$.

In order to ensure correct synchronization between the processes, which also guarantees that the buffer neither overflows, nor overexhausts, we use three variables.

- The variables c_1, c_2 and $turn$ ensure that access to the buffer is protected and provide mutual exclusion between the two critical sections, in which buf is accessed and modified. Whenever one of the processes starts accessing and updating buf, the other process cannot access until the previous access is completed.

- The variable *ne* contains the number of free available slots in the buffer *buf*. It protects *buf* against overflowing.
- The variable *nf* contains the number of items currently in the buffer *buf*. It protects *buf* against overexhausting when the buffer is empty.

8.4 Conclusion

In this chapter, we discussed synchronization and communication for concurrent computations in the framed programming language FTLL. The major synchronization operator is *await*.

The language FTLL contains the extended Tempura as an executable subset. The hierarchy of the consistent three level languages: an extended logic language (EITL), a non-deterministic framed concurrent programming language (FTLL), and an extended Tempura, could facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way. In practice, we feel that the framing operators and minimal model semantics may enable us to narrow the gap between temporal logics and programming languages in a realistic way.

Chapter 9

A Framed Interpreter for Extended Tempura

Summary: An outline of the implementation of the framed interpreter including implementation strategy, data structures, program structure and some relevant reduction rules for the projection, parallel and framing operators are presented.

Tempura has several executable interpreters written in different languages including Moszkowski's Lisp version and Hale's C version [39]. All of them interpret statements from the basic Tempura presented in [61]. In order to use the framing technique and the projection operator in programming, a new interpreter for the extended Tempura has been developed using SICSTUS Prolog. The framed interpreter is intended to interpret the extended Tempura including the previous operator, the parallel and projection operators, as well as the framing and await operators. Since the extended Tempura is interpreted by the minimal model semantics under the model theory of a default logic, there is a radical change in the semantics of the underlying language in contrast with the other interpreters for Tempura. Moreover, the framed interpreter is written in Prolog, it is therefore closer to logic programming.

The interpreter accepts a well-formed program in the extended Tempura as its input, and interprets the program through a sequence of states. At each state the values of variables of the program are evaluated and output. Thus, if a program is eventually reduced to true then it is satisfiable and a model is found; otherwise the program has no model. In the latter case, the error message is indicated and the execution stops. During the reduction of a program, syntax is checked and errors are indicated by the interpreter. During the execution, any problems related to the semantics of the executed program are found and indicated in a dynamic manner.

This chapter is intended to discuss implementation techniques for the extended Tempura. Section 9.1 introduces implementation strategies; Section 9.2 introduces data structures; Section 9.3 introduces the program structure; Section 9.4 presents some relevant reduction rules for the projection, parallel, await and framing operators; and conclusions are given in Section 9.5.

9.1 Implementation Strategy

Basically, the implementation strategy for the framed interpreter is based on the Tableau method [88]. That is, to execute a program is to transform it to a logically equivalent conjunction of two formulas *Present* and *Remains*:

$$Present \wedge Remains$$

where the formula *Present* consists of assignments (by $=$ or $\Leftarrow$) to program variables, output of program variables, *true* and *more* or *empty*. The roles of *more* and *empty* are to indicate whether or not the interval over which a program is executed terminates. Formally,

$$Present = \bigwedge_{i=1}^{m} present_i$$

$$present_i ::= \; x = e \mid x \Leftarrow e \mid true \mid more \mid empty$$

The formula *Remains* is what is executed in the subsequent state if the interval does indeed continue. The *Remains* is said to be in a *reduced form* if either it is true or it consists of conjuncts leading with only the next operators. Formally,

$$Remains = \bigwedge_{i=1}^{n} \bigcirc w_i$$

where w_i is a Tempura formula.

When the execution of the next state is prepared, a function, *next_w*, is used to remove these next operators from the conjuncts contained in *Remains*. What is really executed at the next state is the formula *Next*

$$Next = \bigwedge_{i=1}^{n} w_i$$

$$w_i = next\_w(\bigcirc w_i)$$

Since only deterministic programs are considered within the interpreter, the *Remains* part has a simple form. Compared with the normal form presented in Theorem 7.9, it is different from the one used for reduction. The differences are twofold: one is that the reduced form is $\bigwedge \bigcirc w_i$ rather than a single $\bigcirc w$ for the *Remains* part. It is obvious that the two formulas are equivalent in the sense $w \equiv \bigwedge w_i$. The reason why we use the former for reduction is merely for convenient operation. The other is that the *Present* part may contain *more*, *empty* and *true* while in the normal form they are absent. Again, the reason for using them is merely for easy manipulation in reduction.

9.2 Data Structures

9.2.1 Variables

As mentioned earlier, there are different types of variables in the framed Tempura. Variables can be static or dynamic. A static variable keeps stable over an interval while a dynamic variable may change from state to state.

Variables can also be global or local. If a variable is introduced by the existential quantification it is local otherwise it is global. A local variable is accessed only within the program bound by the existential quantification, whereas a global variable is accessed everywhere within a program.

As discussed earlier, variables are also divided into framed and non-framed ones. The value of a framed variable is carried along to the next state if no assignment is encountered at the next state whereas the value of a non-framed variable is always cleared after the execution of a state. Hence, in the framed interpreter, a variable is stored in the form:

$$val(x, v, status, frmark)$$

where, x is the name of the variable, v is the value of the variable at the current state or next state indicated by the status which has value *state*, *next* or *static*, and the *frmark* can be *f* (framed) or *nf* (non-framed) to identify whether or not x is framed.

9.2.2 Constants

Constants consist of integers, boolean constants, lists and strings. They are defined as in Chapter 4.

9.2.3 Flags

The framed interpreter makes use of several flags to handle the reduction of a program. One of the important flags, *done*, indicates whether or not an interval terminates. At the beginning of execution at each state, *done* is set to *nil*. During the execution at a state, the interpreter places either *false* or *true* in the *done* according to *more* or *empty* being encountered. If a programmer fails to specify the interval for his program, the interpreter cannot set the *done* flag, and it remains *nil*. In such a case, an error will be detected and indicated.

The assigned flag *af* is the second important flag in the framed interpreter. The functions of *af* are twofold: one which indicates whether or not a variable will be changed at the current state; the other is to identify if a variable is accessible at the current state. If the $af(x)$ is true, then x will be assigned a new value at the current state and it is not accessible before the assignment is completed. $af(x)$ is set to true prior to the current state by using syntax check.

The third flag is *side* flag concerning side effects. When this flag is set to off, the interpreter is in the ordinary mode, whereas when the flag is set to *on*, it prevents side effects. For instance, *more* sets the *done* flag to false and *empty* sets the *done* flag to true as mentioned earlier, but in the case of the *side* flag being true, they do nothing. The *side* flag is set to *true* whenever a tail recursion is involved in the reduction of a program.

9.3 Program Structure

The execution of a program consists of a series of state reductions. The last state contains the conjunct *empty*. A state reduction is composed of the reductions of multiple passes. After the last pass, the executed program is reduced to the form:

$$Present \wedge Remains$$

In fact, the *Present* part has been dissolved during the reduction. Its effects are reflected in updating variables, displaying the values of some variables, and setting the *done* flag etc. What is left at the last pass of the reduction is the *Remains* which will be executed at the next state if the interval over which the program is executed does not end. Therefore, the execution of a program can be treated as a series of reductions or rewritings.

9.3.1 One Pass Reduction

Algorithm 1: One-pass-reduction

In SICSTUS Prolog, the algorithm can be written in the following form:

```
1) $retract(w(P))$,
2) $rw(P,Q)$,
3) $simple(Q,R)$,
4) $assert(w(R))$.
```

where $w(P)$ is a compound term, a structured data in Prolog; w is its principal functor; and P is a variable. $Retract(w(P))$ enables P to obtain a program which will be executed. The $rw(P,Q)$ is the most important reduction procedure which rewrites P to Q corresponding to reduction rules at different passes and different states. The $simple(Q,R)$ is intended to erase conjunct *true* contained in Q and the result is placed in the variable R. The $assert(w(R))$ stores the R into w again.

9.3.2 Reduction at One State

Algorithm 2: One-state-reduction

The algorithm proceeds as follows:

```
1) clear-done-flag,
2) show-time,
3) one-pass-reduction,
4) ready? if no go to 3) else 5),
5) checking done flag and preparing assigned flags,
6) preparing the next state.
```

At the beginning of the execution of each state, the interpreter clears *done* flag, i.e. sets it to *nil*. The 'show-time' displays the current state number and increases the number by one in the preparation for the next state. The initial number is 0. The one-pass-reduction, as discussed above, reduces a program in one pass. At the end of the process, a check is required to test whether or not the program is ready in a reduced form. This is the task of *ready*. If the program is not in a reduced form, then another reduction pass resumes. If the program is already reduced to a reduced form, an auxiliary check is required to ensure that *done* has been set to true or false. Otherwise an error 'done-not-set' is issued and the reduction interrupts immediately.

The 'preparing the next state' process does the following: if the current state is not the last one, then this process initializes two storages for each variable according to the information of framing and assignments, and releases the old storages. The process also erases the leading operator $\bigcirc$ from the conjuncts of the program held in w.

The 'prepare-assigned-flag' process is important for framing. After the *ready* procedure, it is the exact time for checking whether or not the interval is empty and there is therefore a chance to do the syntax check so that assigned flags for some variables involved in temporal assignments, fin, and *halt* etc. can be set prior to the next state.

9.3.3 Reduction of One Program

Algorithm 3: One-program-reduction

To execute a program is to call one-state-reduction repeatedly.

```
1) one-state-reduction,
2) empty? if no go to 1) else clear all variables.
```

9.3.4 Execution of Tempura

Algorithm 4: Executing tempura

This is the main algorithm employed in the framed interpreter.

```
1) show Tempura version,
2) initialization,
3) show Tempura number,
4) input(P), (user's task)
5) read(w(P)),
6) P=exit, quit, stop? if yes clear all then stop else 7),
7) one-program-reduction,
8) prepare-next-formula and then go to 3).
```

The interpreter can successively execute many programs. When a program is finished, the interpreter is waiting for the next input unless one intends to quit and issue an exit command.

9.4 Implementing New Operators

The most important reduction procedure is $rw(P, Q)$. It is relevant to all reduction rules for all constructs. For the most of general Tempura formulas, Moszkowski has described how to execute them in [61]. In this section, we discuss only how the new operators, projection, parallel, await and framing, can be implemented in the interpreter. The reduction rules are presented briefly.

9.4.1 Implementation of Projection Operator

The projection construct $(p_1, \ldots, p_m)\ prj\ q$ is implemented as follows: it is processed by first allocating a *done* flag initialized to *nil* to serve as a *done* flag for projected interval over which the statement q is executed and then transforming the statement to the internal construct IC,

$$IC = \begin{cases} project((p_2, \ldots, p_m), p_1, q, done(nil)) & \text{if } m > 1 \\ project(empty, p_1, q, done(nil)) & \text{if } m = 1 \end{cases}$$

which is immediately re-reduced. The construct $project(R, P, Q, done(D1))$ is executed by first saving the current *done* flag to OLD and setting the *done* flag to $done(D1)$. The statement Q is then reduced in the context to a new statement Q'. Afterwards, the current *done* flag is saved to $D2$, the old *done* flag, $done(OLD)$, is restored, and the statement P is then reduced in the context to a new statement P'. If P' or Q' is not fully reduced, the overall *project* statement is rewritten as

$$project(R, P', Q', done(D2)).$$

This is returned as the result of the reduction. On the other hand, if P' and Q' are both fully reduced, then, with the notation

$$\begin{array}{lcl} D & = & next\_w(P') \\ E & = & next\_w(Q') \\ choose(R1, (R2)) & = & (R1; choose((R2))) \\ choose(R1) & = & R1 \end{array}$$

the overall *project* statement is transformed as in Fig 9.1. This tests the *done* flag indexed by $done(D2)$. If it is true, the interval over which Q was reduced is finished and therefore the $next\_w(P')$ is executed followed by the remaining formulas $(R1, (R2))$, chosen by the procedure *choose*, if they were not empty. On the other hand, if $done(D2) = done(false)$ the interval over which Q was executed is not yet finished. Therefore, the formula D is executed followed by the resumption of the projection statement.

```
if $done(D2) = done(true)$
then
   if $R = (R1, (R2))$
   then $\bigcirc\,(D; choose(R1, (R2)))$
   else
      if $R = R1$
      then $\bigcirc\,(D; R1)$
      else
         if $R = empty$
         then $\bigcirc\, D$
else
   if $done(D2) = done(false)$
   then
      if $R = (R1, (R2))$
      then $\bigcirc\,(D; project((R2), R1, E, done(nil)))$
      else
         if $R = R1$
         then $\bigcirc\,(D; project(empty, R1, E, done(nil)))$
         else
            if $R = empty$
            then $\bigcirc\,(D; project(, empty, empty, E, done(nil)))$
```

Fig 9.1 Rewriting *project* statement

9.4.2 Implementing Parallel Operator

The interpreter handles a statement of the form $A||B$ by transforming the statement to the internal construct

$$parallel(A, B, done(nil))$$

This is immediately re-reduced. Here *done(nil)* is a flag initialized to nil. It serves as a local *done* flag for the interval over which the statement A is executed.

We execute the $parallel(A, B, done(D1))$ construct by first saving the value of the old *done* flag to OLD and setting it to $D1$. The statement A is then reduced in the context to a new statement A'. Afterwards the current *done* flag is saved to $D2$, and the old *done* flag value, OLD, is restored. The statement B is then reduced in the context to a new statement B'. If A' or B' is not fully reduced, then we simplify A' to A1 and B' to $B1$ and the overall parallel statement is rewritten as

$$parallel(A1, B1, done(D2))$$

This is returned as the result of the reduction. On the other hand, if A' and B' are both fully reduced, then the overall parallell is transformed to the following conditional statement and then immediately re-reduced:

$$\begin{array}{l} if\ done(OLD) = done(true)\ then\ (A1, done(D2)) \\ else\ if\ (done(D2) = done(true)\ then\ B1 \\ else\ \bigcirc parallel(A1, B1, done(nil)) \end{array}$$

Here $A1 = next\_w(A')$ and $B1 = next\_w(B')$, and $(A1, done(D2))$ means to set *done* flag to D2 and then to reduce $A1$ in the context.

This first tests B's *done* flag indexed by OLD. If it is true, the interval in which B was reduced is finished and therefore the overall parallel construct reduced to $(A1, done(D2))$. On the other hand, it tests A's *done* flag indexed by $D2$; if it is true, the interval over which A was reduced is finished and the overall parallel construct is reduced to $B1$. Otherwise, the reduction procedure advances to the next state.

9.4.3 Implementing await(c)

A statement of the form $await(c)$ is rewritten as a logically equivalent formula

$$halt(c) \wedge frame(V_c)$$

and then it is immediately re-reduced, where V_c stands for all dynamic variables contained in c.

The condition in *halt* cannot cause side effects unless empty. Hence, the $halt(c)$ can be reduced using the following rules:

1. $halt(c) \wedge empty \rightarrow c$
2. $halt(c) \wedge more \rightarrow \neg c \wedge \bigcirc halt(c)$
3. $halt(true) \rightarrow empty$
4. $halt(false) \rightarrow more \wedge \bigcirc halt(c)$

The $frame(V_c)$ can be reduced according to the rules given in the next subsection.

9.4.4 Implementation of Framing Operators

To implement the framing operators, we first consider programs containing no positive immediate assignments. Afterwards, we discuss the general situation.

(1) Implementing Framing in Programs without Positive Immediate Assignments

In the case in which no positive immediate assignments are contained in programs, the basic assignment relevant to framing is the positive next assignment. There are two ways to go about it: one way is by means of the look forward framing operator; the other way is to make use of the look backward framing operator. In the following, we consider the former. The latter can be handled by the case (2) in the sequel.

With the look forward framing operator, to reduce a program, we associate each variable with a mark in its storage to identify its framing status in addition to its value and state mark. So a variable is stored in the form, (Nam, Val, St, Fr). The parameter Nam and Val are simply the name and the value of the variable; St can be either c (current) or nx (next) or st (static); Fr is f (framed) or nf (non-framed). As a program is reduced, a variable, say x, within the program takes two storages at every state. One is a current storage initialized to the form, (x, nil, c, nil), at the beginning of the initial state s_0. The other is a next storage initialized to the form, (x, Val, nx, Ass), with Val and Ass being nil at the beginning of every reduction state. As the reduction proceeds, these parameters will be updated at various states. The detail of the algorithm for changing these parameters is omitted here.

When $frame(x)$ occurs within the context of the program, we invoke the look forward framing operator. By FE2, the framing operator can be written as

$$frame(x) \equiv (more \rightarrow lff(x)) \wedge \odot frame(x)$$

Thus, it is reduced to true if the final state is reached. Otherwise, we simply set a mark f in the current storage of x. This framing mark along with the status of the next assignment provides us with sufficient information to determine if the value of x is carried along to the next state, when the reduction is accomplished at the current state. When advancing to the next state, we check the two storages of each variable whose framing mark is f in its current storage. Three cases need considering:

1. if the value in the next storage is nil, then we copy its value from the current storage to the next one;
2. if the value in the next storage is the same as in the current storage, we do nothing;
3. if the value in the next storage differs from the value in the current storage but the mark $Ass = +$ in the next storage, we do nothing;

Afterwards, we release all of the current storages, and change the next storages to be the new current storages at the next state. This can be done by changing nx to c, setting Fr to nil, and keeping all of the other parameters. The new next storage is initialized to (x, nil, nx, nil) for the variable x.

A framed program is more efficient than a program without framing. For example, comparing program (2) with program (3) in Chapter 6, for the former, without framing, we have to reduce $\Box(more \rightarrow \bigcirc x = x)$ from state to state; while for the latter, with framing, we need to reduce $frame(x)$ with the whole reduction process. To reduce $\Box(more \rightarrow \bigcirc x = x)$, first, we reduce it to a form $(more\rightarrow\bigcirc x = x)\wedge\odot\Box(more\rightarrow\bigcirc x = x)$. Subsequently, if the final state is not reached, $more\rightarrow\bigcirc x = x$ is, in turn, reduced to $\bigcirc x = x$ otherwise to true. Afterwards, if the final state is still not reached, we reduce $\bigcirc x = x$ by accessing to the value of x, and reducing it to $\bigcirc(x = c)$ for some constant c in D. Finally, the whole formula is reduced to the normal form, $\bigcirc(x = c\wedge\Box(more\rightarrow\bigcirc x = x))$. The reduction process needs at least four reduction runs. Whereas, to reduce $frame(x)$, a mark f is placed in the current storage of x in one reduction run. An assignment $x :=^+ e$ is reduced by taking almost the same amount of work as $x := e$ except that we need to set Ass to $+$ in the next storage of x. However, this does not increase the number of reduction runs. The extra work connected with $frame(x)$ is to check the storages of x for inheriting its value. This is fast because only one copy of the value of x in the current storage is required if x is framed and next value is nil. The following is a brief reduction of program (3) in Chapter 6:

$$\begin{array}{llcl}
s_0: & \multicolumn{3}{l}{frame(x)\wedge(x = 1\wedge y :=^+ 2; y :=^+ x + y)} \\
 & (x, nil, c, nil) & & (x, 1, c, f) \\
 & (y, nil, c, nil) & & (y, nil, c, nil) \\
 & (x, nil, nx, nil) & \longrightarrow & (x, nil, nx, nil) \\
 & (y, nil, nx, nil) & & (y, 2, nx, +) \\
 & & \Downarrow & \\
s_1: & \multicolumn{3}{l}{frame(x)\wedge y :=^+ x + y} \\
 & (x, 1, c, nil) & & (x, 1, c, f) \\
 & (y, 2, c, nil) & & (y, 2, c, nil) \\
 & (x, nil, nx, nil) & \longrightarrow & (x, nil, nx, nil) \\
 & (y, nil, nx, nil) & & (y, 3, nx, +) \\
 & & \Downarrow & \\
s_2: & \multicolumn{3}{l}{frame(x)\wedge empty} \\
 & (x, 1, c, nil) & & \\
 & (y, 3, c, nil) & & \\
 & (x, nx, nil, nil) & & \\
 & (y, nx, nil, nil) & &
\end{array}$$

where $\rightarrow$ represents the reduction relation within one state, while $\Downarrow$ represents the relation between two states. Thus, it is clear that program (3) is more efficient than program (2) in Chapter 6.

The restriction for ruling out the positive immediate assignments in programs is not a problem since this does not reduce the expressive power of the language but makes more concise and constructive programs. Moreover, for most imperative programming languages, assignment statements take a unit of time, so the restriction makes the transformation between framed Tempura (or FTLL) and imperative languages easier.

However, the positive immediate assignment is a key formula in the framed Tempura, therefore it should be permitted within a specification. If a program involves positive immediate assignments, the reduction of the program is somewhat complicated.

(2) Implementing Framing in Programs with Positive Immediate Assignments

In general, a program may involve any kind of assignments presented in Chapter 4 and Chapter 6 including the positive immediate assignment. In this case, the basic assignment relevant to framing is the positive immediate assignment. To reduce a program, we also need two storages for a variable at every state except the initial one. However, a previous storage rather than the next storage is required besides the current storage. As the reduction proceeds, these parameters are updated from state to state according to an algorithm which is omitted here.

When $frame(x)$ occurs within the context of the program, we invoke the look back framing operator. By FE2, the framing operator can be written as

$$frame(x) \equiv (more \rightarrow \bigcirc lbf(x)) \wedge \odot frame(x)$$

Thus, it is reduced to true if the current state is the final state. Otherwise, we simply set a mark f in the current storage of x. This framing mark and the assignment status of x at the next state provide us with useful information to determine if the value of x is carried along when the reduction advances to the next state. After progressing to the next state, we release the past storages, and change the current storages to be new past storages at the next state.

However, since a program may involve the positive immediate assignments, from an operational point of view, we can not foresee the assignment status of a variable by the current knowledge at the present state because a positive immediate assignment may appear after a chop operator. For instance, to reduce the program $frame(x) \wedge x = 1 \wedge skip; x \Leftarrow 2 \wedge y :=^{+} x + 3$, we can obtain a framing mark f in the current storage of x but cannot perceive the existence of $x \Leftarrow 2$ after the chop operator in an operational manner. To solve this problem, some syntax check algorithms are required. We omit the detail here.

9.5 Conclusions

The framing technique for variables can easily be generalized to arrays (or lists). In such a case, all elements of an array are treated as a single unit, so $frame(A)$ means that all elements of array A are framed. For implementation of $frame(A)$, only one framing mark is needed to associate with the array name A. Hence checking the framing mark for inheriting the value of A is very quick. In contrast with the repeated assignment approach to an array, the framing approach is clearly more efficient.

A reduction may not involve any syntax checking if we rule out the positive immediate assignments within programs. Therefore, when we allow a program to invoke the positive immediate assignment, the reduction of the program is somewhat complicated. In practice, as a convention, positive immediate assignments could be ruled out in a program but can be used to specify high level executable specification. For most programs, this is not a problem. Finally, we conclude that framed programs are more concise, simpler, and easier to understand. In the case without positive immediate assignments, the implementation of a framed program is also more efficient than a program without framing.

A question one can ask is why we need to introduce the previous operator in this thesis since $frame(x) \equiv frame'(x)$ as Theorem 7.1 tells us. The answer is not obvious. Although, from a theoretical point of view (temporal semantical view), the framing operators involving *next* and *previous* operators are equivalent, the implementations (operational view) of them are different. Therefore, in addition to the well known reasons for introducing past operators to the temporal logic [54], there is also a particular reason for us to do so. That is, the reduction of the positive immediate assignment can be implemented in a manageable way. It is clear that the reduction of a program containing positive immediate assignments would be difficult without the previous operator. In such a case, we can only obtain a set of next assigned variables at the current state. However, a positive immediate assignment possibly occurs after a chop operator. It may destroy the current view of the assignment status of a variable for the next state. Thus, a difficulty arises when we determine if a value of a framed variable is carried along to the next state as the reduction advances. This indicates the fact that the previous operator is really needed for the framing.

The reduction rules used in the framed interpreter are based on the logic laws and the minimal model provided in the preceding chapters. The consistency between the framed interpreter and the minimal model semantics could also be discussed. However, a detailed justification of the problem lies outside the scope of the thesis and is left for futhure research.

Chapter 10

Conclusion

An extended propositional ITL, an extended first order ITL, a projection construct, an extended Tempura, a framing technique and a synchronous communication construct have been investigated in detail; a framed interpreter for the extended Tempura has also been briefly introduced. However, some issues need to be discussed further. This chapter is devoted to discussing them.

10.1 Extended Propositional ITL

- Although the extended propositional ITL is based on the original ITL, the semantics of EPITL is much different from PITL. First, the extension to include past operators destroys the logic law, $\mathit{empty}; p \equiv p$. That is, in general, $\mathit{empty}; p \equiv p$ no longer holds. Second, the extension to use infinite models destroys the logic law, $p; \mathit{empty} \equiv p$. In other words, in general, $p; \mathit{empty} \equiv p$ is not valid in EPITL. However, $p; \mathit{empty}; q \equiv p; q$ holds and the association law, $(p; q); r \equiv p; (q; r)$, is still valid.

- The negation of the chop construct, $\neg(p; q)$, is hard to express in EITL. Instead, we use some weak laws (see Theorem 2.28) relevant to the negation of the chop construct rather than using *yield* operator [78, 60].

- The next operator refers to one state forward but not beyond the final state within an interval while the previous operator refers to one state backward but not beyond the first state within an interval. In fact, the chop operator (;) and the past chop ($\bar{;}$) operator make the borders of sub-intervals over which the next and previous operators act. Barringer[11] pointed out the possibility for extending choppy logic to use the previous operator in this manner.

- We claim a number of times in the thesis that we extend ITL, in some sense, from an interval-based notation to a point-based notation. As seen, terms and formulas are interpreted w.r.t. an interpretation (σ, i, k, j) rather than an interval as in the original ITL. Unlike the original ITL, σ is fixed in the notation (σ, i, k, j) while a formula p is interpreted over σ. To interpret p, we start with $(\sigma, 0, 0, |\sigma|)$; subsequently, as the interpretation proceeds, we interpret subformulas of p over some subintervals introduced by the partitions of σ obtained through the chop operators. The notation (σ, i, k, j) provides us with the necessary information: the first parameter σ is the whole interval over which the full formula p is interpreted. That is, σ represents the computation trace of formula p, whereas the other three parameters i, k and j specify a subinterval $\sigma_{(i..j)}$ of σ with the current state being s_k over which a subformula of p is interpreted. When the interpretation proceeds, two situations may take place. In the first, the current position may swing forward or backward between s_i and s_j because the subformula may involve both the next and the previous operators in an arbitrary order. So parameters i and j serve as delimiters, while the parameter k is used as an indicator of the current position so that the move of the position can be consistent with the interpretation. Since the chop operator can be used recursively, another situation may occur in which a chop operator is involved in the subformula of p; it gives rise to the partition of the subinterval $\sigma_{(i..j)}$ into two new subintervals $\sigma_{(i..h)}$ and $\sigma_{(h..j)}$ with the delimiters i, h and h, j respectively. The current position for the former is k while for the latter it is h in the future chop case. For the past chop case, the current position for the former is h while for the latter it is k. The change of the current state and the partition of intervals are all handled by means of increments or decrements in the values of the parameters i, k, j. In a sense, we generalize ITL from an interval-based notation to a point-based notation since we refer to no explicit subintervals but to points i, k, j, over a fixed interval. This could allow us to compare the extended logic system with a point-based linear temporal logic such as [66, 51] in an easy way and to formalize a proof system possibly by adding axioms concerning the chop operators to these existing systems.

- In this thesis, EPITL has been studied within the model theory. A proof theory could be formulated. However, a lot of work is needed and this is beyond the scope of this thesis.

10.2 Extended First Order ITL

EITL has been developed based on EPITL. EITL is much different from the first order ITL.

- A problem we need to point out is the use of *nil*. In most point-based temporal logics [51, 66, 53], a model is an infinite sequence of states. However, within EITL (and in the original ITL), the chop operators force us to use finite intervals as models in addition to infinite models. Thus, problems arise when we define the value of $\bigcirc x$ at a final state and the value of $\ominus x$ at a starting state. In this thesis, they are treated as undefined (denoted by *nil*). Introducing *nil* to the logic poses another problem: should we have $nil = nil$ or $nil \neq nil$ when we interpret the equality $(=)$? If we adopt the convention, $nil = nil$, then $\bigcirc e_1 = \bigcirc e_2 \equiv \bigcirc(e_1 = e_2)$ does not hold at the final state and $\ominus e_1 = \ominus e_2 \equiv \ominus(e_1 = e_2)$ does not hold at the first state over an interval. This destroys some intuitions. On the other hand, however, if we make the convention, $nil \neq nil$, then it poses more problems since different equalities are needed to interpret terms and the equality $(=)$ in the semantics, and some very useful laws such as replacement of equal term by equal term in a term does not hold. Within this thesis, for our purpose, we adopt $nil = nil$ as a convention.

- Another issue is a proof system for EITL. A lot of work is needed to develop a deductive system for EITL, and this is left for future research.

10.3 Projection

- The projection operator subsumes the chop operator as stated in Theorem 5.3. However, the projection operator is only defined as a future operator in this thesis. It is possible to define a projection over a subinterval $\sigma_{(i..j)}$ with the current state being s_k so that the previous operator can be used. However, further research is required.

- The projection construct has a potential use in real time systems, especially, in hybrid systems [41, 4] consisting of continuous activities and discrete events. Intuitively, continuous activities can be interpreted over a series of local but continuous intervals, while discrete events can be interpreted over a projected interval (discrete interval). Thus, the continuous activities and the discrete events can be mixed in a uniform way. Therefore, the projection construct is useful to handle hybrid systems. However, further research is needed to investigate the technical aspects in detail.

10.4 Extended Tempura

- In the thesis, we extended Tempura in several ways. However, the input, output statements, data types declaration statements and pointers are excluded. It seems that there is no straightforward way to include these statements in Tempura under the EITL notation. Further research is required to solve these problems.
- In the dissertation, the temporal semantics of programs within the extended Tempura is investigated under the model theory. Operational and axiomatic semantics of programs are still waiting for further research.
- Another very active research field is the real time programming. At the moment, the extended Tempura is concerned with a sequence of states without absolute time. We could find a way to extend Tempura to use time explicitly so that real time systems can be handled by Tempura.
- The extended Tempura is not a deterministic language. A subset of deterministic programs in the extended Tempura can be defined. However, in this thesis, a formal definition of deterministic programs has not been given.

10.5 Framing

- Framing is managed in a default logic within this thesis. This inevitably leads to revisable reasoning [84] for framed programs. However, we have not investigated the revisable reasoning within EPITL and EITL.
- Moreover, the narrative reasoning [67] may also be done with framed Tempura. Therefore, framed Tempura could be a useful tool in AI field.

10.6 Synchronous Communication

In the thesis, the await operator is defined with the help of framing, and a useful language FTLL is defined in EITL.

- The language FTLL contains the extended Tempura as a sub-language. The hierarchy of the consistent three level languages: an extended logic language, a non-deterministic framed concurrent programming language (FTLL), and an extended Tempura, could facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way. From our experience, we believe that the framing operator and the minimal model semantics may enable us to narrow the gap between temporal logic programming languages and conventional programming languages in a realistic way.

- For synchronous communication, we have not found a way to handle the group statement which is useful to define *P-V* operation which is the traditional approach for solving the mutual exclusion problem.

10.7 Interpreter

Although the previous operator, the projection operator, the framing operator, and the await operator have all been implemented, and the framed interpreter is workable, the current version of the interpreter is not complete. The domain includes only integers, and there is no library of functions etc. A compiler for the extended Tempura is also required.

10.8 Comparison with other works

The formalism presented in this thesis refers to Kröger's temporal logic [51], Manna and Pnueli's temporal logic [66], Lamport's TLA [53], Moszkowski's ITL [61] and others (e.g., [11, 24]).

EITL is close to the original ITL. They share several common operators, such as the next ($\bigcirc$) and chop (;). However, ITL is a linear temporal logic based on intervals of states. Its basic temporal operators are future operators, the next ($\bigcirc$) and chop (;). A model of ITL is a finite interval of states. In contrast with ITL, EITL uses both future temporal operators, the next and chop, as well as past temporal operators, the previous ($\ominus$) and past chop ($\bar{;}$). A model of EITL can be a finite or infinite interval of states. Furthermore, the semantics of EITL is different from ITL. The interpretations of terms and formulas in ITL are by means of intervals, whereas within EITL, they are interpreted over a fixed interval with changing current end points. Hence, EITL is a point-based linear temporal logic in this sense. Many constructs in EITL have the same names as in ITL. However, their meaning can be different. For instance, *empty* means a singleton interval in ITL, but it means that the right end of the current interval is reached in EITL. For projection constructs, $p\ prj\ q$ in ITL and $(p_1, ..., p_m)\ prj\ q$ in EITL, their meaning is different, as discussed in Chapter 5.

ITL has a simple proof system [62]. However, the model theory in ITL has not been explored. EITL provides model theories for both the propositional and first order logics. However, a proof system for EITL has not been formalised.

Kröger's temporal logic is one of the earliest versions of temporal logic, and it is a main reference here. This logic includes future operators such as $\bigcirc$, $\Diamond$, *atnext* and *until*, as the basic operators. A model of the logic is an infinite sequence of states. The logic has a substantial model theory and a proof system. Within the propositional logic, the completeness of the logic has been proved. However, no executable sub-set of the underlying logic has been formalised. This logic cannot handle the chop and projection constructs. Conversely, EITL cannot handle the *atnext* operator.

Manna and Pnueli's temporal logic [66] includes both future operators, such as $\bigcirc$, $\Diamond$ and U (until), and past operators, such as $\ominus$, $\diamondminus$ and S (since). A model of this logic is an infinite sequence of states. The definition of a state is similar to EITL. It also has a proof system. However, the model theory of this temporal logic has not been investigated in detail. It does not handle the chop and projection constructs. To verify programs, Manna and Pnueli define separately a programming language in the normal programming context rather than a sub-set of the underlying logic. It seems that EITL cannot handle *until* and *since* constructs.

Lamport's TLA [53] is a simple temporal logic of actions. This logic includes the always operator $\Box$ as the basic future temporal operator. An important point we need to make is that Lamport [52] objects to the use of the next operator in a specification language, claiming that it enables the expression of distinctions between programs that should be considered equivalent. He consistently uses reflexive operators instead of the next operator. In this logic, fairness is investigated in detail. This logic has an executable sub-set and also a proof system. It cannot handle the chop and projection constructs, as well as the *until* and *atnext* constructs. In contrast with TLA, EITL does not deal with the fairness.

The extended Tempura is close to the original Tempura and Hale's work [39]. Many statements of the extended Tempura are similar to the constructs of the original Tempura. However, the semantics of the language is given in EITL model theory. Moreover, the extended Tempura has a new projection statement $(p_1, ..., p_m)$ *prj* q and a communication and synchronization statement *await*(c) as well as the framing operator. These allow us to handle concurrent complex computations.

Another temporal logic programming language, Tokio [32], is also based on ITL. This language combines temporal operators with Prolog. The extended Tempura does not refer to this language.

The XYZ/E [79, 82] is one of the earliest temporal logic programming languages. This language is based on Manna and Pnueli's temporal logic [63]. Moreover, 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 [83]. However, XYZ/E cannot handle the chop and projection constructs.

The predominant approach to the extension of the logic programming paradigm to temporal logic is TEMPLOG [3]. In TEMPLOG, Temporal Horn Clauses, which can be categorised as either *initial* clauses (effectively if they contain **start**) or globale clauses, are restricted still further using the following constraints.

1. The '$\Diamond$' operator cannot occur in the head of a clause.

2. The '$\Box$' operator can only occur in the head of what are termed, *initial definite permanent* clauses, which can be characterised as

$$\Box e \leftarrow d, \mathbf{start}, \mathbf{c}.$$

Gabbay developed the language USF [35], which follows an imperative future approach. The METATEM language [9, 29] is a development of USF consisting of a larger range of operators, a better defined execution mechanism [31] and a more practical normal form [30]. 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 from one moment to the next. A temporal rule is given in the following clausal form

past time antecedent *implies* present and future consequent

These languages are significantly different from the extended Tempura.

As this thesis was not intended to provide a survey of executable temporal logics, merely a comparison with some of the basic mechanisms, there are obviously languages utilising this paradigm that we have not mentioned. The other temporal programming languages can be found in [87, 36, 14, 80, 48, 55, 70].

The framing technique presented in this thesis is concerned with framing in a particular manner in which framed and non-framed variables are mixed. This framing technique makes use of a framing operator $frame(x)$, an assignment flag $af(x)$ and a positive immediate assignment $x \Leftarrow e$.

The minimal model for framed Tempura is based on the minimal model (or fixed point) semantics [13] for logic programming languages and default logics [76, 56, 57]. However, within logic programming, the minimal model is based on Herbrand model which applied to both variables and propositions. Within the extended Tempura, the minimal model is only applied to propositions.

Considerable attention has been given to framing in recent years [68, 40, 53, 83, 39]; however, no intensive study has been done.

Ness [68] claims that a framing technique has been used within L.0 but no formal definition is presented. Hehner [40], Lamport [53], and Manna and Pnueli [66] define their programming languages implicitly using framing technique. As mentioned in Chapter 6, the assignment is defined as follows:

$$x := e \overset{\text{def}}{=} x' = e \wedge y_i' = y_i \ (1 \leq i \leq m)$$

where x is a variable, and x' represents the new value of x. $y_1, ..., y_m$ which are different from x are all the other variables within a program. Intuitively, this means that whenever a variable x is assigned a value, the other variables remain stable. However, this method can only manage the case in which all variables are framed, and the conjunction of assignments is forbidden.

Hale [39] first introduced explicitly, as far as we know, a framing technique based on inertial idea into the temporal logic programming area. However, no formal theory is presented to convince us that the technique works well. As he said in his thesis: ‘ ... but it has to be admitted that the definition is rather tricky, and has not yet been proved to work in all such programs’.

Many other researchers deal with framing in a simplified manner, e.g. assuming the values of variables are automatically inherited [83].

The framed interpreter developed using SICSTUS Prolog is the first version including the framing, projection and await constructs. Hence, it is different from the interpreter written in C for Tempura developed by Hale [39].

10.9 Future Work

In this section, we outline three main areas for our future research.

10.9.1 A Proof System for EITL

In this thesis, a collection of logic laws for both the extended propositional ITL and the extended first order ITL have been formalised and proved. However, we have not worked out proof systems for EPITL and EITL. Therefore, we intend to formalise a deductive system for EPITL and EITL. This could be attempted by adding new axioms and inference rules to the existing proof systems such as those of Manna and Pnueli [66], Kröger [51] and Moszkowski [62].

10.9.2 Axiomatic Semantics of the Extended Tempura

In the thesis, we have investigated the temporal semantics of programs in the extended Tempura within the model theory. In particular, we have introduced the minimal model to capture temporal semantics of framed programs in the extended Tempura. As for conventional imperative programming languages, we envisage that the semantics of programs and framed programs in the extended Tempura can be captured by an axiomatic system based on the proof system for EPITL. For framed programs, such a system would contain a default rule to capture the meaning of the minimal model semantics.

10.9.3 Operational Semantics of the Extended Tempura

The semantics of (framed) programs in the extended Tempura can be expressed in an operational style. To achieve this, we plan to formalise an inference rule system by following the approach advocated by Plotkin [74]. We expect that such an inference rule system would be close to the existing interpreter for Tempura programs. Moreover, we intend to investigate the consistency between the temporal, axiomatic and operational semantics of programs within the extended Tempura.

Finally, we conclude that the thesis has successfully extended ITL to include infinite models, past operators and new projection operator. The resulting extended Tempura is still a non-deterministic language, but we expect that a sufficiently general deterministic subset of the extended Tempura can be defined syntactically. The work on minimal model based temporal semantics of framed programs can provide a new direction for research in temporal logic programming. Although the substitution laws presented in Chapter 7 are sufficient for the purpose of the reduction of framed programs, additional substitution laws concerning the general case under the minimal model are required to carry out substitutions within the model in a more flexible manner.

Appendix

Definition A.1 Two interpretations $\mathcal{I}_1 = (\sigma_1, i_1, k_1, j_1)$ and $\mathcal{I}_2 = (\sigma_2, i_2, k_2, j_2)$ are equivalent, denoted by $\mathcal{I}_1 \sim \mathcal{I}_2$, if for every term e, $\mathcal{I}_1[e] = \mathcal{I}_2[e]$, and for every formula p, $\mathcal{I}_1 \models p$ iff $\mathcal{I}_2 \models p$.

Let σ be an interval, i, k, i_1, k_1 integers, and j, j_1 integers or ω such that $0 \leq i_1 \leq i \leq k \preceq j \preceq j_1$, then we can show $(\sigma, i, k, j) \sim (\sigma_{(i_1..j_1)}, i - i_1, k - i_1, j - i_1)$ (see Theorem A.1).

Intuitively, the two interpretations use the same segment of σ, i.e. the sub-interval, $< s_i, ..., s_j >$, and refer to the same state s_k as the current state. To prove the conclusion formally, some details regarding terms and formulas need considering.

Theorem A.1 Let $\mathcal{I} = (\sigma, i, k, j)$ and $\mathcal{I}' = (\sigma_{(i'..j')}, i - i', k - i', j - i')$ be interpretations with $0 \leq i' \leq i \leq k \preceq j \preceq j'$. Then $\mathcal{I} \sim \mathcal{I}'$.

Proof

Let e be a term, and p a formula. The proof proceeds by induction on the structure of terms and formulas. We first prove $\mathcal{I}[e] = \mathcal{I}'[e]$.

- If $e = c \in D'$, then $\mathcal{I}[e] = c = \mathcal{I}'[e]$.
- If $e = x \in V$, then $\mathcal{I}[e] = s_k[x] = s'_{k-i'}[x] = \mathcal{I}'[e]$, where $s'_0, s'_1, ...$ are the states of $\sigma_{(i'..j')}$.

Suppose $\mathcal{I}[e] = \mathcal{I}'[e]$ for any interval σ and for all i, k, j, i', j', such that $0 \leq i' \leq i \leq k \preceq j \preceq j' \preceq |\sigma|$. Thus,

- For a term of the form: $\bigcirc e$, if $k < j$, by the interpretation I-next, then $\mathcal{I}[\bigcirc e] = (\sigma, i, k + 1, j)[e]$ and $\mathcal{I}'[\bigcirc e] = (\sigma_{(i'..j')}, i - i', k - i' + 1, j - i')[e]$. By hypothesis, $(\sigma, i, k + 1, j)[e] = (\sigma_{(i'..j')}, i - i', k - i' + 1, j - i')[e]$. Hence, $\mathcal{I}[\bigcirc e] = \mathcal{I}'[\bigcirc e]$.

  If $k = j$, then $\mathcal{I}[\bigcirc e] = nil = \mathcal{I}'[\bigcirc e]$.
- For a term of the form: $\circleddash e$, if $k > i$, by the interpretation I-pre, then $\mathcal{I}[\circleddash e] = (\sigma, i, k - 1, j)[e]$ and $\mathcal{I}'[\circleddash e] = (\sigma_{(i'..j')}, i - i', k - i' - 1, j - i')[e]$. By hypothesis, $(\sigma, i, k - 1, j)[e] = (\sigma_{(i'..j')}, i - i', k - i' - 1, j - i')[e]$. Hence, $\mathcal{I}[\circleddash e] = \mathcal{I}'[\circleddash e]$.

  If $k = i$, then $i - i' = k - i'$. So, $\mathcal{I}[\circleddash e] = nil = \mathcal{I}'[\circleddash e]$.
- For a term of the form: $beg(e)$, $\mathcal{I}[beg(e)] = (\sigma, i, i, j)[e] = (\sigma_{(i'..j')}, i - i', i - i', j - i')[e] = \mathcal{I}'[beg(e)]$.
- For a term of the form: $end(e)$, if $j < \omega$, then $\mathcal{I}[end(e)] = (\sigma, i, j, j)[e]$ and $\mathcal{I}'[end(e)] = (\sigma_{(i'..j')}, i - i', j - i', j - i')[e]$. By hypothesis, $(\sigma, i, j, j)[e] = (\sigma_{(i'..j')}, i - i', j - i', j - i')[e]$. Hence, $\mathcal{I}[end(e)] = \mathcal{I}'[end(e)]$.

  If $j = \omega$, then $j - i' = \omega$. Thus, $\mathcal{I}[end(e)] = nil = \mathcal{I}'[end(e)]$.

- For a term in the form of a function $f(e_1, ..., e_m)$, notice that, by hypothesis, $\mathcal{I}[e_h] = nil$ iff $\mathcal{I}'[e_h] = nil$ for all $h, 1 \leq h \leq m$. Thus, if $\mathcal{I}[e_h] = \mathcal{I}'[e_h] = nil$ for some $h, 1 \leq h \leq m$, then $\mathcal{I}[f(e_1, ..., e_m)] = f(\mathcal{I}[e_1], ..., \mathcal{I}[e_m]) = nil = f(\mathcal{I}'[e_1], ..., \mathcal{I}'[e_m]) = \mathcal{I}'[f(e_1, ..., e_m)]$.

  On the other hand, if for all $h, 1 \leq h \leq m, \mathcal{I}[e_h] \neq nil$, then, by hypothesis, $\mathcal{I}[e_h] = \mathcal{I}'[e_h]$. Thus, $\mathcal{I}[f(e_1, ..., e_m)] = f(\mathcal{I}[e_1], ..., \mathcal{I}[e_m]) = f(\mathcal{I}'[e_1], ..., \mathcal{I}'[e_m]) = \mathcal{I}'[f(e_1, ..., e_m)]$.

We now prove $\mathcal{I} \models p$ iff $\mathcal{I}' \models p$. Note that, in the proof, we will frequently use the conclusion $\mathcal{I}[e] = \mathcal{I}'[e]$ without clarification.

- For a proposition p, $\mathcal{I} \models p$ iff $I_p^k[p] =$ true iff $I_p'^{k-i'}[p] =$ true iff $\mathcal{I}'[p] \models p$.

- For a predicate $P(e_1, ..., e_n)$, since $\mathcal{I}[e_h] = \mathcal{I}'[e_h]$, for all $h, 1 \leq h \leq n$, if there exists a $k, 1 \leq k \leq n, \mathcal{I}[e_k] = nil$, then $P(\mathcal{I}[e_1], ..., \mathcal{I}[e_n]) = P(\mathcal{I}'[e_1], ..., \mathcal{I}'[e_n]) =$ false; otherwise, $\mathcal{I} \models P(e_1, ..., e_n)$ iff $P(\mathcal{I}[e_1], ..., \mathcal{I}[e_n]) =$ true iff $P(\mathcal{I}'[e_1], ..., \mathcal{I}'[e_n]) =$ true iff $\mathcal{I}' \models P(e_1, ..., e_n)$.

- For an equality $e_1 = e_2$, since $\mathcal{I}[e_h] = \mathcal{I}'[e_h]$ for $h = 1, 2$, $\mathcal{I} \models e_1 = e_2$ iff $\mathcal{I}[e_1] = \mathcal{I}[e_2]$ iff $\mathcal{I}'[e_1] = \mathcal{I}'[e_2]$ iff $\mathcal{I}' \models e_1 = e_2$.

Suppose $\mathcal{I} \models p$ iff $\mathcal{I}' \models p$ for any interval σ and for all i, k, j, i', j', such that $0 \leq i' \leq i \leq k \preceq j \preceq j' \preceq |\sigma|$. Thus,

- For the formula $\neg p$, we have, $\mathcal{I} \models \neg p$ iff $\mathcal{I} \not\models p$ iff $\mathcal{I}' \not\models p$ iff $\mathcal{I}' \models \neg p$.

- For the formula $p \wedge q$, $\mathcal{I} \models p \wedge q$ iff $\mathcal{I} \models p$ and $\mathcal{I} \models q$ iff $\mathcal{I}' \models p$ and $\mathcal{I}' \models q$ iff $\mathcal{I}' \models p \wedge q$.

- For a formula in the form: $\bigcirc p$, $\mathcal{I} \models \bigcirc p$ iff $i \leq k < j$ and $(\sigma, i, k+1, j) \models p$. By hypothesis, $(\sigma, i, k+1, j) \models p$ iff $(\sigma_{(i'..j')}, i-i', k-i'+1, j-i') \models p$. Moreover, $(\sigma_{(i'..j')}, i-i', k-i'+1, j-i') \models p$ iff $\mathcal{I}' \models \bigcirc p$. Therefore, $\mathcal{I} \models \bigcirc p$ iff $\mathcal{I}' \models \bigcirc p$.

- For a formula in the form: $\ominus p$, $\mathcal{I} \models p$ iff $i < k \preceq j$ and $(\sigma, i, k-1, j) \models p$. By hypothesis, $(\sigma, i, k-1, j) \models p$ iff $(\sigma_{(i'..j')}, i-i', k-i'-1, j-i') \models p$. Further, $(\sigma_{(i'..j')}, i-i', k-i'-1, j-i') \models p$ iff $\mathcal{I}' \models \ominus p$. Therefore, $\mathcal{I} \models \ominus p$ iff $\mathcal{I}' \models \ominus p$.

- For a formula in the form $p;q$, we have $\mathcal{I} \models p;q$ iff for some r, such that $k \leq r \preceq j$ $(\sigma, i, k, r) \models p$ and $(\sigma, r, r, j) \models q$.

  $\mathcal{I}' \models p;q$ iff for some h, such that $k - i' \leq h \preceq j - i'$ $(\sigma_{(i'..j')}, i-i', k-i', h) \models p$ and $(\sigma_{(i'..j')}, h, h, j-i') \models q$. That is, for some r, such that $k \leq r \preceq j$ $(r = h + i')$ $(\sigma_{(i'..j')}, i-i', k-i', r-i') \models p$ and $(\sigma_{(i'..j')}, r-i', r-i', j-i') \models q$.

  Since $0 \leq i' \leq i \leq k \leq r \preceq j'$ and $0 \leq i' \leq r \leq r \preceq j \preceq j'$ hold, by hypothesis, we obtain

  $$(\sigma, i, k, r) \models p \text{ iff } (\sigma_{(i'..j')}, i-i', k-i', r-i') \models p$$

  $$(\sigma, r, r, j) \models q \text{ iff } (\sigma_{(i'..j')}, r-i', r-i', j-i') \models q$$

  Therefore, $\mathcal{I} \models p;q$ iff $\mathcal{I}' \models p;q$.

- For a formula in the form $p\bar{;}q$, a similar proof can be given.

- For a formula in the form $\exists x : p$, since $< s'_{i'}, ..., s'_{j'} >_{(i-i'..j-i')} = < s'_i, ..., s'_j >$ and $< s_{i'}, ..., s_{j'} >_{(i-i'..j-i')} = < s_i, ..., s_j >$, and, by hypothesis, $(\sigma', i, k, j) \models p$ iff $(\sigma'_{(i'..j')}, i - i', k - i', j - i') \models p$, we have

$$\begin{array}{lll} \mathcal{I} \models \exists x : p & \text{iff} & \text{for some } \sigma', (\sigma', i, k, j) \models p \text{ with } \sigma_{(i..j)} \overset{x}{=} \sigma'_{(i..j)}, \text{ i.e. } < s_i, ..., s_j > \overset{x}{=} < s'_i, ..., s'_j > \\ & \text{iff} & \text{for some } \sigma', (\sigma'_{(i'..j')}, i - i', k - i', j - i') \models p \text{ with } < s'_i, ..., s'_j > \overset{x}{=} < s_i, ..., s_j > \\ & \text{iff} & \text{for some } \sigma', (\sigma'_{(i'..j')}, i - i', k - i', j - i') \models p \text{ with } < s'_{i'}, ..., s'_{j'} >_{(i-i'..j-i')} \overset{x}{=} \\ & & < s_{i'}, ..., s_{j'} >_{(i-i'..j-i')} \\ & \text{iff} & \mathcal{I}' \models \exists x : p. \end{array}$$

- For a formula in the form $(p_1, ..., p_m)\ prj\ q$, we have,

$$(\sigma, i, k, j) \models (p_1, ..., p_m)\ prj\ q$$

iff there exist integers $r_0, r_1, ..., r_{m-1}$ and $r_m \in N_\omega$ such that $r_0 = k$ and $(\sigma, i, k, j) \models [p_1, ..., p_m](r_1, ..., r_m)$ and

  - $r_m = j$ and $\sigma \downarrow (r_0, ..., r_h) \models q$ for some h, $0 \leq h \leq m$ or
  - $r_m < j$ and $\sigma \downarrow (r_0, ..., r_m).\sigma_{(r_m+1..j)} \models q$.

That is, iff there exist integers $r_0, r_1, ..., r_{m-1}$ and $r_m \in N_\omega$ such that $r_0 = k$ and $(\sigma, i, r_0, r_1) \models p_1$ and for all $1 < l \leq m$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ and

  - $r_m = j$ and $\sigma \downarrow (r_0, ..., r_h) \models q$ for some h, $0 \leq h \leq m$ or
  - $r_m < j$ and $\sigma \downarrow (r_0, ..., r_m).\sigma_{(r_m+1..j)} \models q$.

By induction hypothesis, we have,

$$(\sigma, i, r_0, r_1) \models p_1 \text{ iff } (\sigma_{(i'..j')}, i - i', k - i', r_1 - i') \models p_1,$$

$$(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l \text{ iff } (\sigma_{(i'..j')}, r_{l-1} - i', r_{l-1} - i', r_l - i') \models p_l \text{ for all } l, 1 < l \leq m,$$

$$\sigma\downarrow(r_0, ..., r_h) \models q \text{ iff } \sigma_{(i'..j')}\downarrow(r_0 - i', ..., r_h - i') \models q,$$

$$\sigma\downarrow(r_0, ..., r_m).\sigma_{(r_m+1..j)} \models q \text{ iff } \sigma_{(i'..j')}\downarrow(r_0 - i', ..., r_m - i').\sigma_{(r_m-i'+1..j-i')} \models q.$$

Thus, we obtain,

$$(\sigma, i, k, j) \models (p_1, ..., p_m)\ prj\ q$$

iff there exist integers $r'_0 = r_0 - i', r'_1 = r_1 - i', ..., r'_{m-1} = r_{m-1} - i'$ and $r'_m = r_m - i' \in N_\omega$ such that $r'_0 = k - i'$ and $(\sigma_{(i'..j')}, i - i', r'_0, r'_1) \models p_1$ and $(\sigma_{(i'..j')}, r'_{l-1}, r'_{l-1}, r'_l) \models p_l$ for all $l, 1 < l \leq m$ and

  - $\sigma_{(i'..j')}\downarrow(r'_0, ..., r'_h) \models q$ for some h, $0 \leq h \leq m$ or
  - $\sigma_{(i'..j')}\downarrow(r'_0, ..., r'_m).\sigma_{(r'_m+1..j-i')} \models q$.

iff there exist integers $r'_0, r'_1, ..., r'_{m-1}$ and $r'_m \in N_\omega$ such that $r'_0 = k - i'$ and $\mathcal{I}' \models [p_1, ..., p_m](r'_1, ..., r'_m)$ and

  - $\sigma_{(i'..j')}\downarrow(r'_0, ..., r'_h) \models q$ for some h, $0 \leq h \leq m$ or
  - $\sigma_{(i'..j')}\downarrow(r'_0, ..., r'_m).\sigma_{(r'_m+1..j-i')} \models q$.

iff $\mathcal{I}' \models (p_1, ..., p_m)\ prj\ q$.

- For a formula in the form p^+, we have,

$\mathcal{I} \models p^+$ iff there are finitely many $r_0, ..., r_n \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \leq r_{n-1} \preceq r_n = j$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $1 < l \leq n, (\sigma, r_{l-1}, r_{l-1}, r_l) \models p$; or $j = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq r_2 \leq ...$ such that $\lim\limits_{i \to \infty} r_i = \omega$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

By induction hypothesis, we have,

$$(\sigma, i, r_0, r_1) \models p \text{ iff } (\sigma, i - i', r_0 - i', r_1 - i') \models p,$$

$$(\sigma, r_{l-1}, r_{l-1}, r_l) \models p \text{ iff } (\sigma_{(i'..j')}, r_{l-1} - i', r_{l-1} - i', r_l - i') \models p \text{ for all } l > 1$$

Thus, we obtain,

$\mathcal{I} \models p^+$ iff there are finitely many integers $r'_0 = r_0 - i', ..., r'_n = r_n - i' \in N_\omega$ $(n \geq 1)$ such that $k - i' = r'_0 \leq r'_1 \leq ... \leq r'_{n-1} \preceq r'_n = j - i'$ and $(\sigma_{(i'..j')}, i - i', r'_0, r'_1) \models p$ and, for all l, $1 < l \leq n, (\sigma_{(i'..j')}, r'_{l-1}, r'_{l-1}, r'_l) \models p$;
or $j = \omega$ and there are infinitely many integers $k - i' = r'_0 = r_0 - i' \leq r'_1 = r_1 - i' \leq, ...,$ such that $\lim\limits_{i \to \infty} r'_i = \omega$, and $(\sigma, i - i', r'_0, r'_1) \models p$, and for all $l > 1$, $(\sigma, r'_{l-1}, r'_{l-1}, r'_l) \models p$.

iff $\mathcal{I}' \models p^+$

□

If $i = i'$ and $j = j'$ in Theorem A.1, then we obtain,

Corollary A.2 $(\sigma, i, k, j) \sim (\sigma_{(i..j)}, 0, k - i, j - i)$.

□

The proof of Theorem 3.7 (continued)

Let formulas $p_i \equiv p'_i$ $(1 \leq i \leq m)$ and $q \equiv q'$. We need to prove the following in addition to the proofs given in Theorem 2.7.

1. $\exists x : q \equiv \exists x : q'$
2. $(p_1, ..., p_m)\ prj\ q \equiv (p'_1, ..., p'_m)\ prj\ q'$

Let σ be a model and k an integer, $0 \leq k \preceq |\sigma|$.

The proof of 1:

$$\begin{aligned}(\sigma, 0, k, |\sigma|) \models \exists x : q &\iff \text{there exists } \sigma', \sigma' \overset{x}{=} \sigma \text{ , and } (\sigma', 0, k, |\sigma'|) \models q \\ &\iff \text{there exists } \sigma', \sigma' \overset{x}{=} \sigma \text{ , and } (\sigma', 0, k, |\sigma'|) \models q' \\ &\iff (\sigma, 0, k, |\sigma|) \models \exists x : q'\end{aligned}$$

The proof of 2:
Let $\mathcal{I} = (\sigma, 0, k, j), j = |\sigma|$.

$$\mathcal{I} \models (p_1, \cdots, p_m)\ prj\ q$$

iff there exist integers $r_1, ..., r_{m-1}$ and $r_m \in N_\omega$ such that $r_0 = k$ and $(\sigma, i, k, j) \models [p_1, ..., p_m](r_1, ..., r_m)$ and

- $r_m = j$ and $\sigma{\downarrow}(r_0, \ldots, r_h) \models q$ for some $h, 0 \leq h \leq m$ or
- $r_m < j$ and $\sigma{\downarrow}(r_0, \ldots, r_m){\cdot}\sigma_{(r_m+1..j)} \models q$.

Since $p_i \equiv p_i'$ and $q \equiv q'$, by hypothesis, we have

$$\mathcal{I} \models (p_1, \ldots, p_m)\ prj\ q$$

iff there exist integers $r_1, ..., r_{m-1}$ and $r_m \in N_\omega$ such that $r_0 = k$ and $(\sigma, i, k, j) \models [p_1', ..., p_m'](r_1, ..., r_m)$ and

- $r_m = j$ and $\sigma{\downarrow}(r_0, \ldots, r_h) \models q'$ for some $h, 0 \leq h \leq m$ or
- $r_m < j$ and $\sigma{\downarrow}(r_0, \ldots, r_m){\cdot}\sigma_{(r_m+1..j)} \models q'$.

iff $\mathcal{I} \models (p_1', \ldots, p_m')\ prj\ q'$

□

We now turn our attention to the laws regarding the chop plus and chop star. We first introduce two auxiliary definitions.

Definition A.2

$$\begin{array}{lll} 1.\ p^0 & \stackrel{\text{def}}{=} & empty \\ 2.\ p^1 & \stackrel{\text{def}}{=} & p \\ \quad p^n & \stackrel{\text{def}}{=} & p; p^{n-1}\ \ (n > 1) \end{array}$$

The semantics of p^n can be given as follows:

$(\sigma, i, k, j) \models p^n$ iff there are extended finitely many integers $r_0, ..., r_n = j \in N_\omega$ $(n \geq 1)$ such that $k = r_0 \leq r_1 \leq ... \leq r_{n-1} \preceq r_n$ and $(\sigma, i, r_0, r_1) \models p$ and, for all $1 < l \leq n, (\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

Definition A.3
We define an auxiliary formula p^∞. Its semantics is given as follows:

$(\sigma, i, k, j) \models p^\infty$ iff $j = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq ...$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, i, r_0, r_1) \models p$, and for all $l > 1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$.

Thus, following conclusions are obviously true.

Theorem A.3 Let $\mathcal{I} = (\sigma, i, k, j)$ be any interpretation. Then,

1. $\mathcal{I} \models p^+$ iff $\mathcal{I} \models p^n \vee p^\infty$ for some $n \geq 1$
2. $\mathcal{I} \models p^*$ iff $\mathcal{I} \models p^n \vee p^\infty$ for some $n \geq 0$

□

The following 'negative' conclusions regarding $p^\imath$ and p^∞ are also obviously true.

Theorem A.4

$$\begin{array}{llll} NPW1. & p^\infty; q & \equiv & false \\ NPW2. & (p^\infty)^\infty & \equiv & false \\ NPW3. & p^{0+\imath} & \equiv & p^{\imath+0} \ \equiv\ p^\imath \text{ does not hold} \end{array}$$

The logic laws regarding p^k are given in Theorem A.5.

Theorem A.5

$$\begin{array}{llll} PW1. & p^i;p^j & \equiv & p^{i+j} \quad (i,j \geq 1) \\ PW2. & (p^i)^j & \equiv & p^{ij} \quad (i,j \geq 1) \\ PW3. & p^i;p^\infty & \equiv & p^\infty \quad (i \geq 1) \\ PW4. & (p^i)^\infty & \equiv & p^\infty \quad (i \geq 1) \end{array}$$

Proof

We prove only 4. Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

$(\sigma,0,k,|\sigma|) \models (p^i)^\infty$ iff $|\sigma| = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq ...$ such that $\lim\limits_{i\to\infty} r_i = \omega$ and $(\sigma,0,r_0,r_1) \models p^i$ and $(\sigma,r_{l-1},r_{l-1},r_l) \models p^i$ for all $l > 1$.

$(\sigma,0,r_0,r_1) \models p^i$ iff there are finitely many integers $r_0^0, r_1^0, ..., r_{i-1}^0$ and a $r_i^0 = r_1$ such that $r_0 = r_0^0 \leq r_1^0 \leq ... \leq r_{i-1}^0 \preceq r_i^0 = r_1$ and $(\sigma,0,r_0^0,r_1^0) \models p$ and for all t, $1 < t \leq i$, $(\sigma,r_{t-1}^0,r_{t-1}^0,r_t^0) \models p$.

$(\sigma,r_{l-1},r_{l-1},r_l) \models p^i$ iff there are finitely many integers $r_0^{l-1}, r_1^{l-1}, ..., r_{i-1}^{l-1}$ and a $r_i^{l-1} = r_l$ such that $r_{l-1} = r_0^{l-1} \leq r_1^{l-1} \leq ... \leq r_{i-1}^{l-1} \preceq r_i^{l-1} = r_l$ and for all t, $1 \leq t \leq i$, $(\sigma, r_{t-1}^{l-1}, r_{t-1}^{l-1}, r_t^{l-1}) \models p$.

Thus, let $rr_{ix+y} = r_y^x$ for $x \geq 0$ and $0 \leq y \leq i-1$. We have,

$(\sigma,0,k,|\sigma|) \models (p^i)^\infty$ iff $|\sigma| = \omega$ and there are infinitely many integers $rr_0, rr_1, ...$ such that $rr_0 = k$ and $rr_h \leq rr_{h+1}$ for all $h \geq 0$ and $\lim\limits_{i\to\infty} rr_i = \omega$ and $(\sigma,0,rr_0,rr_1) \models p$ and $(\sigma,rr_{l-1},rr_{l-1},rr_l) \models p$ for all $l > 1$.

iff $(\sigma,0,k,|\sigma|) \models p^\infty$.

□

Theorem A.6

1. $(p^+)^n \supset p^+$
2. $(p^+)^\infty \supset p^+$

Proof

The proof of 1

Let σ be an interval, k an integer, $0 \leq k \preceq |\sigma|$, and $\mathcal{I} = (\sigma,0,k,|\sigma|)$. The proof proceeds by induction on n. Thus,

1. When $n = 1$ the conclusion is obviously true.

2. Suppose when $n = m \geq 1$ the conclusion holds. That is, $(p^+)^m \supset p^+$.

3. When $n = m + 1$, we have,

$\mathcal{I} \models (p^+)^{m+1}$

$\Longleftrightarrow$ $\mathcal{I} \models p^+; (p^+)^m$ definition A.2

$\Longrightarrow$ $\mathcal{I} \models p^+; p^+$ induction hypothesis, theorem 3.5

$\Longleftrightarrow$ $(\sigma, 0, k, r) \models p^+$ and $(\sigma, r, r, |\sigma|) \models p^+$ for some $r, k \leq r \preceq |\sigma|$

$\Longleftrightarrow$ $(\sigma, 0, k, r) \models p^j \vee p^\infty$ for some $j \geq 1$ and $(\sigma, r, r, |\sigma|) \models p^l \vee p^\infty$ for some $l \geq 1$,

and for some $r, k \leq r \preceq |\sigma|$, theorem A.3

$\Longleftrightarrow$ $(\sigma, 0, k, |\sigma|) \models (p^j \vee p^\infty); (p^l \vee p^\infty)$ for some $j \geq 1$ and for some $l \geq 1$

$\Longleftrightarrow$ $\mathcal{I} \models (p^j; p^l) \vee (p^\infty; p^l) \vee (p^j; p^\infty) \vee (p^\infty; p^\infty)$ for some $j \geq 1$ and for some $l \geq 1$,

FD9,10

$\Longleftrightarrow$ $\mathcal{I} \models p^{j+l} \vee p^\infty$ for some $j \geq 1$ and for some $l \geq 1$, theorems A.4, A.5

$\Longleftrightarrow$ $\mathcal{I} \models p^i \vee p^\infty$ for some $i = j + l \geq 2$

$\Longrightarrow$ $\mathcal{I} \models p^h \vee p^\infty$ for some $h \geq 1$

$\Longleftrightarrow$ $\mathcal{I} \models p^+$ theorem A.3

The proof of 2

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

$(\sigma, 0, k, |\sigma|) \models (p^+)^\infty$ iff $(\sigma, 0, k, |\sigma|) \models (p^j \vee p^\infty)^\infty$ for some $j \geq 1$.

iff $|\sigma| = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq \ldots$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, 0, r_0, r_1) \models (p^j \vee p^\infty)$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models (p^j \vee p^\infty)$ for all $l > 1$.

Since $(\sigma, 0, r_0, r_1) \not\models p^\infty$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \not\models p^\infty$ for all $l > 1$, we obtain,

$(\sigma, 0, r_0, r_1) \models p^j$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p^j$ for all $l > 1$.

This implies $(\sigma, 0, k, |\sigma|) \models (p^j)^\infty$. By Theorem A.5 PW4, $(\sigma, 0, k, |\sigma|) \models p^+$.
Therefore $(p^+)^\infty \supset p^+$.

□

Theorem A.7 If p is a lec-formula, then

1. $(p^*)^n \supset p^*$
2. $(p^*)^\infty \supset p^+$

Proof

The proof of 1

Let σ be an interval, k an integer, $0 \leq k \preceq |\sigma|$, and $\mathcal{I} = (\sigma, 0, k, |\sigma|)$. The proof proceeds by induction on n. Thus,

1. When $n = 1$ the conclusion is obviously true.
2. Suppose when $n = m \geq 1$ the conclusion holds. That is, $(p^*)^m \supset p^*$.
3. When $n = m + 1$, we have,

$\mathcal{I} \models (p^*)^{m+1}$

$\Longleftrightarrow$ $\mathcal{I} \models p^*; (p^*)^m$ definition A.2

$\Longrightarrow$ $\mathcal{I} \models p^*; p^*$ induction hypothesis, theorem 3.5

$\Longleftrightarrow$ $(\sigma, 0, k, r) \models p^*$ and $(\sigma, r, r, |\sigma|) \models p^*$ for some $r, k \leq r \preceq |\sigma|$

$\Longleftrightarrow$ $(\sigma, 0, k, r) \models \mathit{empty} \vee p^j \vee p^\infty$ for some $j \geq 1$ and $(\sigma, r, r, |\sigma|) \models \mathit{empty} \vee p^l \vee p^\infty$ for some $l \geq 1$, and for some $r, k \leq r \preceq |\sigma|$, theorem A.3

$\Longleftrightarrow$ $\mathcal{I} \models (\mathit{empty} \vee p^j \vee p^\infty); (\mathit{empty} \vee p^l \vee p^\infty)$ for some $j \geq 1$ and for some $l \geq 1$

$\Longleftrightarrow$ $\mathcal{I} \models (\mathit{empty}; \mathit{empty}) \vee (p^j; \mathit{empty}) \vee (p^\infty; \mathit{empty}) \vee (\mathit{empty}; p^l) \vee (p^j; p^l) \vee (p^\infty; p^l)$ $\vee(\mathit{empty}; p^\infty) \vee (p^j; p^\infty) \vee (p^\infty; p^\infty)$ for some $j \geq 1$ and for some $l \geq 1$, FD9,10

$\Longleftrightarrow$ $\mathcal{I} \models \mathit{empty} \vee (p^j; \mathit{empty}) \vee p^l \vee p^{j+l} \vee p^\infty$ for some $j \geq 1$ and for some $l \geq 1$, EMP1, FEP2, theorems A.4, A.5

$\Longrightarrow$ $\mathcal{I} \models \mathit{empty} \vee (p^j; \mathit{empty}) \vee (p^j \wedge \Box \mathit{more}) \vee p^l \vee p^{j+l} \vee p^\infty$ for some $j \geq 1$ and for some $l \geq 1$

$\Longleftrightarrow$ $\mathcal{I} \models \mathit{empty} \vee p^j \vee p^l \vee p^{j+l} \vee p^\infty$ for some $j \geq 1$, and for some $l \geq 1$, TER

$\Longleftrightarrow$ $\mathcal{I} \models \mathit{empty} \vee p^h \vee p^\infty$ for some $h \geq 1$

$\Longleftrightarrow$ $\mathcal{I} \models p^*$ theorem A.3

The proof of 2

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

$(\sigma, 0, k, |\sigma|) \models (p^*)^\infty$ iff $(\sigma, 0, k, |\sigma|) \models (\mathit{empty} \vee p^+)^\infty$.

iff $|\sigma| = \omega$ and there are infinitely many integers $k = r_0 \leq r_1 \leq \ldots$ such that $\lim_{i \to \infty} r_i = \omega$ and $(\sigma, 0, r_0, r_1) \models (\mathit{empty} \vee p^+)$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models (\mathit{empty} \vee p^+)$ for all $l > 1$.

We now use the following algorithm to rule out some integers from the sequence of $r_0, r_1, \ldots$.

1. if $(\sigma, 0, r_0, r_1) \models \mathit{empty}$ but $(\sigma, 0, r_0, r_1) \not\models p^+$, then we delete r_0;
2. for all $l > 1$, if $(\sigma, r_{l-1}, r_{l-1}, r_l) \models \mathit{empty}$ but $(\sigma, r_{l-1}, r_{l-1}, r_l) \not\models p^+$, then we delete r_{l-1};

(Note that, in the above algorithm, 1. whenever a r_0 or r_{l-1} is deleted, an identical integer r_1 or r_l is reserved; 2. whenever $(\sigma, 0, r_0, r_1) \models empty \wedge p^+$ or $(\sigma, 0, r_0, r_1) \models more \wedge p^+$, and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models empty \wedge p^+$ or $(\sigma, r_{l-1}, r_{l-1}, r_l) \models more \wedge p^+$, we do nothing.)

Then, we obtain a sequence of integers $r'_0, r'_1, \dots$ from $r_0, r_1, \dots$ by deleting some duplicates. Thus, $r'_0, r'_1, \dots$ is an infinite sequence of integers, and $\lim_{i \to \infty} r_i = \omega$ and $r'_0 = k$ and $r'_h \le r'_{h+1}$ for all $h \ge 0$. Furthermore, $(\sigma, 0, r'_0, r'_1) \models p^+$ and, for all $l > 1$, $(\sigma, r'_{l-1}, r'_{l-1}, r'_l) \models p^+$.

This implies $(\sigma, 0, k, |\sigma|) \models (p^+)^\infty$. By Theorem A.6 2, $(\sigma, 0, k, |\sigma|) \models p^+$.

Therefore $(p^*)^\infty \supset p^+$. □

The proof of Theorem 2.25 (continued)

The proof of $p^+; p \supset p; p^+$ (FPS4)

Let σ be an interval and k an integer, $0 \le k \preceq |\sigma|$.

$$\begin{array}{ll}
 & (\sigma, 0, k, |\sigma|) \models p^+; p \\
\Longleftrightarrow & (\sigma, 0, k, r) \models p^+ \text{ and } (\sigma, r, r, |\sigma|) \models p \text{ for some } r, k \le r \preceq |\sigma| \\
\Longleftrightarrow & (\sigma, 0, k, r) \models (p^i \vee p^\infty) \text{ for some } i \ge 1 \text{ and } (\sigma, r, r, |\sigma|) \models p, \text{ theorem A.3} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models (p^i \vee p^\infty); p \text{ for some } i \ge 1 \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models (p^i; p) \vee (p^\infty; p) \text{ for some } i \ge 1, \text{FD10} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models p^{i+1} \text{ for some } i \ge 1, \text{ theorems A.4, A.5} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models p; p^i \text{ for some } i \ge 1, \text{ theorem A.5} \\
\Longrightarrow & (\sigma, 0, k, |\sigma|) \models p; p^+ \qquad p^i \supset p^+, \text{ theorem 3.5}
\end{array}$$

The proof of $p; p^+ \supset p^+; p^+$ (FPS5)

Since $p^+ \equiv p \vee (p; p^+)$, so $p \supset p^+$. By Theorem 3.5, we obtain $p; p^+ \supset p^+; p^+$.

The proof of $p^+; p^+ \supset p^+$ (FPS6)

This is an immediate consequence of Theorem A.6 (1).

The proof of $p^{++} \equiv p^+$ (FPS7)

Let σ be an interval and k an integer, $0 \le k \preceq |\sigma|$. Since $p^{++} \equiv p^+ \vee (p^+; p^{++})$, so $p^+ \supset p^{++}$. Conversely,

$$\begin{array}{ll}
 & (\sigma, 0, k, |\sigma|) \models p^{++} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models (p^+)^j \vee (p^+)^\infty \text{ for some } j \ge 1 \\
\Longrightarrow & (\sigma, 0, k, |\sigma|) \models p^+ \vee p^+ \quad \text{theorem A.6} \\
\Longleftrightarrow & (\sigma, 0, k, |\sigma|) \models p^+
\end{array}$$

Therefore, $p^{++} \equiv p^+$. □

The proof of Theorem 2.26 (continued)

The proof of $p^{**} \equiv p^*$ (FST8)

Let σ be an interval and k an integer, $0 \leq k \preceq |\sigma|$.

$$\begin{aligned} p^{**} &\equiv \mathit{empty} \vee (p^*)^+ \\ &\equiv \mathit{empty} \vee p^* \vee (p^*;(p^*)^+) \end{aligned}$$

So $p^* \supset p^{**}$. Conversely,

$$\begin{aligned} & (\sigma, 0, k, |\sigma|) \models p^{**} \\ \Longleftrightarrow \quad & (\sigma, 0, k, |\sigma|) \models \mathit{empty} \vee (p^*)^+ \\ \Longleftrightarrow \quad & (\sigma, 0, k, |\sigma|) \models \mathit{empty} \vee (p^*)^j \vee (p^*)^\infty \text{ for some } j \geq 1 \\ \Longrightarrow \quad & (\sigma, 0, k, |\sigma|) \models \mathit{empty} \vee p^* \vee p^+ \quad \text{theorem A.7} \\ \Longleftrightarrow \quad & (\sigma, 0, k, |\sigma|) \models p^* \end{aligned}$$

Therefore, $p^{**} \equiv p^*$.

□

In a previous paper [22], we introduced a kind of validity of formulas as follows, if $\mathcal{I} \models p$, for every interpretation $\mathcal{I}$, then p is valid. This is equivalent to say that $\Box p$ is valid in the sense of the validity defined within this thesis. We justify the claim in Theorem A.8.

Theorem A.8 Let p be a formula of EITL. Then the following are equivalent:

1. $\mathcal{I} \models p$ for every interpretation $\mathcal{I}$.
2. $\sigma \models \Box p$ for every model σ.

Proof

Suppose for every interpretation $\mathcal{I}$, $\mathcal{I} \models p$. If there is a model σ such that $\sigma \not\models \Box p$, then $(\sigma, 0, k, |\sigma|) \not\models p$ for some k, $0 \leq k \preceq |\sigma|$. Thus, a contradiction arises.

Conversely, suppose for every model σ, $\sigma \models \Box p$. That is, $(\sigma, 0, k, |\sigma|) \models p$ for every σ and all integers k, $0 \leq k \preceq |\sigma|$. If there is an interpretation $\mathcal{I}$, $\mathcal{I} \not\models p$, then, there are σ, i, k, j and $(\sigma, i, k, j) \not\models p$. Thus, by Theorem A.1, we have $(\sigma_{(i..j)}, 0, k, |\sigma_{(i..j)}|) \not\models p$. This causes a contradiction.

□

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design and analysis of computer algorithms.* Addison-Wesley Publishing Company, 1974.

[2] M. Abadi and Z. Manna: *Nonclausal temporal deduction.* Logics of programs, R. Parikh (Ed.), LNCS Vol 193, 1–15, Springer-Verlag, 1985.

[3] M. Abadi and Z. Manna : *Temporal logic programming.* Journal of Symbolic Computations, 8:277-295, 1989.

[4] T.Anderson, R. de Lemos, J.S. Fitsgerald and A. Saeed: *On formal support for industrial-scale requirements analysis.* Robert L.Grossman, Anil Nerode, Anders P.Ravn, and Hans Rischel (Eds) LNCS Vol 736, 426-451, Springer-Verlag, 1993.

[5] M. Ben-Ari: *Principles of concurrent programming.* Prentice-Hall International, London, England, or Englewood Cliffs, New Jersey, 1982.

[6] M. Ben-Ari: *Temporal logic proofs of concurrent programs.* Report CS82-12, Weizmann Ins. of Science, Rehovot, Israel, 1982.

[7] M. Ben-Ari, Z. Manna and A. Pnueli: *The temporal logic of branching time.* Acta Informatica 20, 207-226, 1983.

[8] B. Banieqbal and H. Barringer: *Temporal logic with fixed points.* B. Banieqbal, H. Barringer and A. Pnueli (Eds.), LNCS Vol 389, 62-74, Springer-Verlag, 1987.

[9] H. Barringer, M. Fisher, D. Gabbay, G. Gough and R. Owens: *METATEM: A framework for programming in temporal logic.* In Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formulisms, Correctness, Mook, Netherlands, 1989. (published in LNCS Vol 430, Springer-Verlag).

[10] H. Barringer: *A survey of verification techniques for parallel programs.* LNCS Vol 191, Springer-Verlag, 1985.

[11] H. Barringer, R. Kuiper, and A. Pnueli: *Now you may compose temporal logic specifications.* Proc. 16th ACM Symp. on Theory of Computing, 51–63, 1984.

[12] E. Best and R. Devillers: *Sequential and concurrent behaviour in Petri net theory.* Theoretical Computer Science 55, 87-136, 1987.

[13] N.Bidoit: *Negation in rule-based data base languages: a survey.* Theoretical Computer Science 78 3-83, North-Holland, 1991.

[14] C. Brzoska and K. Schäfer: *LIMETTE: Logic programming integrating metric temporal extensions – language definition and user manual.* Interner Bericht 9/93, Fak für Informatik Universität Karlsruhe.

[15] R. M. Burstall: *Program proving as hand simulation with a little induction.* Information Processing, 308–312, 1974.

[16] E. M. Clarke, E.A. Emerson, and A. P. Sitla: *Automatic verification of finite-state concurrent systems using temporal logic specifications.* ACM Transactions on Programming Languages and Systems, 244–263, 8, 1986.

[17] E. W. Dijkstra: *A discipline of programming.* Prentice-Hall, Inc., 1976.

[18] E. W. Dijkstra: *Guarded commands, nondeterminacy, and formal derivation of programs.* Communication of the ACM 18, 453–457, 1975.

[19] Z.Duan: *An algorithm of constructing even order magic square.* Chinese Journal of Microelectronics and Computer, No 4, 13-16, 1990.

[20] Z. Duan, C. Holt, and B. Moszkowski: *An interpreter for an executable subset of extended interval temporal logic with framing and concurrent operators.* BCTCS 8, Newcastle upon Tyne, March 1992.

[21] Z.Duan, and C.Holt: *Negation by default for framing in Temporal Logic Programming.* Technical Report No.441, Computing Science Department, University of Newcastle upon Tyne, July, 1993.

[22] Z. Duan, M. Koutny, and C. Holt: *Projection in temporal logic programming.* In F. Pfenning (ed.), Proceedings of Logic Programming and Automatic Reasoning, Lecture Notes in Artificial Intelligence, LNCS Vol 822, 333-344, Springer-Verlag, July, 1994.

[23] Z. Duan, M. Holcombe, and A. Linkens: *Timed interval temporal logic and modelling of hybrid systems.* In A. Guasch and R. M.Huber (eds.), Proceedings of Modelling and Simulation ESM'94 534-541, Barcelona, Spain, June 1-3, 1994.

[24] B. Dutertre: *Complete proof systems for first order interval temporal logic.* In Proceedings of LICS'95 36-43, 1995.

[25] E. A. Emerson and E. M. Clarke: *Characterizing correctness properties of parallel programs as fixpoints.* In Proc. 7th Int. Colloq. Aut. Lang. Prog. LNCS Vol 85, 169–181, Springer-Verlag, 1981.

[26] E. A. Emerson and E. M. Clarke: *Using branching temporal logic to synthesize synchronization skeletons.* Science of Computer Programming , Vol 2, 241–266, 1982.

[27] E. A. Emerson and J.Y. Halpern: *'Sometime' and 'not never' revisited: On branching time versus linear time* . Journal of the ACM, 33, 151–178, 1986.

[28] H.B. Enderton: *A mathematical introduction to logic.* Academic press, London, 1972.

[29] M. Fisher: *Towards a semantics for concurrent MTETATEM.* In M. Fisher and R. Owens Eds., Executable Modal and Temporal Logics, Lecture Notes in Artificial Intelligence, Vol 897, Springer-Verlag, 1995.

[30] M. Fisher and P. Noel: *Transformation and synthesis in METATEM-Part I: propositional METATEM.* Technical Report UMCS-92-2-1, Department of Computer Science, University of Manchester, Manchester, M13 9PL, U.K.

[31] M. Fisher and R. Owens: *From past to future: Executing temporal logic programs.* In proceedings of Logic Programming and Automated Reasoning (LPAR), St. Petersberg, Russia, 1992. Lecture Notes in Artificial Intelligence, Vol 624, Springer-Verlag, 1995.

[32] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka: *Tokio: Logic programming language based on temporal logic and its compilation to PROLOG.* Third International Conference on Logic Programming, London, LNCS Vol 225, 695–709, Springer-Verlag, July, 1986.

[33] R. W. Floyd: *Assigning meaning to programs.* Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Vol 19, 27–56, 1967.

[34] D. M. Gabbay: *Theoretical foundations for non-monotonic reasoning in expert systems.* Research Report 84/11, Dept. of Computing, Imperial College, 1984.

[35] D. M. Gabbay: *Modal and temporal logic programming.* Temporal Logic and Their Applications, Antony Galton, 197–237, Academic Press, London, 1987.

[36] D. M. Gabbay: *Modal and temporal logic II (A temporal prolog machine).* In T. Dodd, R. Owens, and S. Torrance (Eds), Logic programming – expanding the Horizon. Intellect Book Ltd., 1991.

[37] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi: *On the temporal analysis of fairness.* Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, 163–173, 1980

[38] A. Galton: *Temporal logics and their applications.* Academic Press, London, 1987.

[39] R. W. S. Hale: *Programming in temporal logic.* Ph.D. Thesis, Trinity College Computer Laboratory, Cambridge University, Cambridge, England, (TR No. 173) July, 1989.

[40] Eric C.R. Hehner: *A practical theory of programming.* Science of Computer Programming 14, 133–158, North-Holland, 1990.

[41] T.A. Henzinger, Z. Manna and A. Pnueli: *Towards Refining Temporal Specifications into Hybrid Systems.* Robert L.Grossman, Anil Nerode, Anders P.Ravn, and Hans Rischel (Eds) LNCS Vol 736, 60-76, Springer-Verlag, 1993.

[42] D. Harel, D. Kozen, and R. Parikh: *Process logic: expressiveness, decidability, completeness.* Journal of Computer and System Sciences, 2, 144-170, 1982.

[43] C. A. R. Hoare: *An axiomatic basis for computer programming.* Communications of the ACM, 12, 576–583, 1969.

[44] C.A.R. Hoare: *Communicating sequential processes.* Communications of the ACM 21, 666-677, 1978.

[45] C.A.R. Hoare: *Communicating sequential processes.* Prentice Hall International, London, 1985.

[46] J. Halpern, Z. Manna and B. Moszkowski: *A hardware semantics based on temporal intervals.* Proceedings of the 10th International Colloquium on Automata, Languages and Programming, LNCS Vol 154, Springer-Verlag, Barcelona, 1983.

[47] B. T. Hailpern and S. Owicki: *Verifying network protocols using temporal logic.* Report 192, Computer Systems Laboratory, Stanford University, 1980.

[48] T. Hrycej: *A temporal extension of Prolog.* The journal of Logic Programming, 15 (1& 2): 113-145, 1993.

[49] R. Janicki and M. Koutny: *Structure of concurrency.* Theoretical Computer Science 112, 5-52, 1993.

[50] S.A. Kripke: *Semantical analysis of modal logic I: normal propositional calculi.* Z. Math. Logik Grund. Math. 9, 67-96, 1963.

[51] F. Kröger: *Temporal logic of programs.* Springer-Verlag, 1987.

[52] L. Lamport: *What good is temporal logic.* In Proc. IFIP 9th World Congress, R.E.A. Mason (ed.), North-Holland, 657-668, 1983.

[53] L. Lamport: *The temporal logic of actions.* ACM TOPLAS, Vol 16 872–923, May 1994.

[54] O. Lichtenstein, A. Pnueli, and L. Zuck: *The glory of the past.* Proc. Conf. on Logics of Programs, LNCS Vol 193, 196–218, 1985.

[55] S. Merz: *Efficiently executable temporal logic programs.* In M. Fisher and R. Owens Eds., Executable Modal and Temporal Logics, Lecture Notes in Artificial Intelligence, Vol 897, Springer-Verlag, 1995.

[56] J. McCarthy: *Circumscription–a form of non-monotonic reasoning.* Artificial Intelligence, Vol 13, 1980.

[57] D. McDermott: *Non-monotonic logic I.* Artificial Intelligence, Vol 13, 1980.

[58] R. Milner: *A calculus of communicating systems.* Springer-Verlag, LNCS Vol 92, 1980

[59] R. Milner: *Communication and concurrency.* Prentice Hall, London, 1989.

[60] B. Moszkowski: *Reasoning about digital circuits.* Ph.D Thesis, Stanford University. TR STAN-CS-83-970, 1983.

[61] B. Moszkowski: *Executing temporal logic programs.* Cambridge University Press, Cambridge, 1986.

[62] B. Moszkowski: *Some very compositional temporal properties.* Programming Concepts, Methods, and Calculi, 307–326. Elsevier Science B.V. (North-Holland), 1994.

[63] Z. Manna and A. Pnueli: *Verification of concurrent programs: the temporal framework.* The Correctness Problem in Computer Science, Robert S. Boyer and J. Strother Moore (Eds.), 215–273, Academic Press, New York, 1981.

[64] Z. Manna and A. Pnueli: *How to cook a temporal proof system for your pet language.* Principles of Programming Languages, Proc. 10th Ann. ACM Symp. 141–154, 1983.

[65] Z. Manna and A. Pnueli: *Verification of concurrent programs: A temporal proof system.* Foundations of Computer Science IV, Amsterdam: Mathematical Center Tracts, Vol 159, 163–255, 1983.

[66] Z. Manna and A. Pnueli: *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

[67] R. Miller and M. Shanahan: *Narratives in the situation calculus.* The Journal of Logic and Computations (Special Issue on Actions and Processes), Vol 4, No 5, 513-530, Michael Georgeff (ed.), Oxford University Press, 1994.

[68] L. Ness : *L.0: A parallel executable temporal logic language.* Proceedings of the ACM SIGSOFT, International Workshop on Formal Methods in Software Development, Napa, California, 1990.

[69] S. Owicki and L. Lamport: *Proving liveness properties of concurrent programs.* ACM Transactions on Programming Languages and Systems, No.3, 455–495, 1982.

[70] M.A. Orgun and W. Ma: *An overview of temporal and modal logic programming.* In First International Conference on Temporal Logic (ICTL), Bonn, Germany, 1994. (published in LNCS, Vol 827).

[71] M.A. Orgun and W. Wadge: *Towards a unified theory of intensional logic programming.* The Journal of Logic Programming, 13 (1,2,3 and 4): 413-440, 1992.

[72] B. Paech: *Gentzen-systems for propositional temporal logics.* Proceedings of the 2nd Workshop on Computer Science Logic, Duisburg (FRG), E. Borger and H. Kleine Buning and M.M. Richter (eds.), LNCS Vol 385, 240-253, Springer-Verlag, 1988

[73] A. N. Prior: *Diodoran modalities.* Philosophical Quarterly, Vol 5, 205–213, 1955.

[74] G. Plotkin: *A structural approach to operational semantics.* Lecture Notes, Aarhus, University, Denmark, 1981.

[75] A. N. Prior: *Past, present, and future.* Clarendon Press, Oxford, 1967.

[76] R. Reiter: *A logic for default Reasoning.* Artificial Intelligence, Vol 13, 1980.

[77] J. A. Robinson: *A machine-oriented logic based on the resolution principle.* Journal of the ACM, Vol 12, 23–41, 1965.

[78] R. Rosner and A. Pnueli: *A choppy logic.* First Annual IEEE Symposium on Logic In Computer Science, LICS, 306–314 1986.

[79] C.S. Tang: *Toward a unified logic basis for programming languages.* Report No. STAN-CS-81-865, Dept. Computer Science, Stanford University, 1981.

[80] T. Tang: *Temporal logic CTL + Prolog.* Journal of Automated Reasoning, 5:49-65.

[81] C.S. Tang: *Toward a unified logic basis for programming languages.* Proceedings of IFIP Congress 83, Amsterdam, Elsevier Science Publishers B.V. (North-Holland), 425–429, 1983.

[82] C.S. Tang: *XYZ: A programming development environment based on temporal logic.* In Bormann (ed.), Programming Languages and Systems, North-Holland, 1983.

[83] C.S. Tang: *A temporal logic language oriented toward software engineering, -introduction to XYZ system (I).* Chinese Journal of Advanced Software Research, Vol 1, 1-27, 1994.

[84] A. Thayse: *From modal Logic to deductive databases, introducing a logic based approach to artificial intelligence.* Wiley, Chichester, 1988.

[85] N. Rescher and A.Urquhart: *Temporal logic.* Springer-Verlag, New York, 1978.

[86] Moshe Y. Vardi, Pierre L. Wolper: *An automata theoretic approach to automatic program verification.* Proceedings of 1st Symposium on Logic in Computer Science, Mathematical Society, Cambridge MA, IEEE Computer Society, 244–332,1986.

[87] W. W. Wadge: *Tense logic programming: a respectable alternative.* Symposium on lucid and Intensional Programming, Sidney B.C. Canada, 26–32, 1988.

[88] P. L. Wolper: *Temporal logic can be more expressive.* Information and Control, Vol 56, 72-99, 1983.

[89] P. L. Wolper: *The tableau method for temporal logic: an overview.* Logique et Analyze, Vol 110-111, 119-136, 1985.