INTEGRATING FORMAL SPECIFICATION AND VERIFICATION METHODS IN SOFTWARE DEVELOPMENT

by

Xudong He

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Department of Computer Science

APPROVED:

John A.N. Lee, Chairman	James D. Arthur
John W. Dickey	Charles D. Feustel
Sallie M. Henry	Dennis G. Kafura

June 1, 1989

Blacksburg, Virginia

Abstract

This dissertation is a part of an intended long-term research project with the objectives to make software development more scientific and rigorous, thereby to achieve better software quality and to facilitate automated software production; and has two major components: the design of the specification transition paradigm for software development and the theoretical study of the system specification phase in the paradigm.

First, after an extensive analysis and comparison of various formalisms, a paradigm for integrating various formal specification and verification methods (predicate transition Petri nets, first order temporal logic, the algebraic, the axiomatic, the denotational and the operational approaches) in software development has been developed. The model more effectively incorporates foremost formalisms than any other models (the Automatic Programming Project [Bal85], the CIP Project [CIP85], the Larch Project [GHW85] and the RAISE Project [MG87]) and has the following distinctive features: (1) specifications are viewed both as a set of products and a set of well-defined steps of a process, (2) specifications (as a set of products) at different development steps are to be written and verified by different formalisms, (3) specification (as a process) spans from the requirement phase to the detailed design phase, (4) specification for both concurrent and sequential software is supported, and (5) specifications for different aspects (concurrent control abstraction, data abstraction and procedural abstraction) of a piece of software are dealt with separatedly.

Second, an intensive and in-depth investigation of the system specification phase in the paradigm results in:

- a design methodology for predicate transition nets, which incorporates the separate definition technique in Ada [Ada83] and state decomposition technique in Statechart [Har88] into the traditional transformation techniques for Petri nets, and therefore

will significantly reduce the design complexity and enhance the comprehensibility of large predicate transition net specifications;

- the establishment of a fundamental relationship between predicate transition nets and first order temporal logic and the design of an algorithm for systematically translating predicate transition nets into equivalent temporal logic formulae. Therefore the goal to combine the strengths of both formalisms, i.e. to use predicate transition nets as a specification method and to use temporal logic as a verification method, is achieved; and

- the discovery of a special temporal logic proof technique based on a Hilbert-style logic system to verify various properties of predicate transition nets and the associated theorems. Thus temporal logic is effectively used as an analysis method for both safety and liveness properties of predicate transition nets.

Acknowledgements

I am very grateful to Professor John A.N. Lee, my thesis advisor. He brought me to Virginia Polytechnic Institute & State University, and enabled my three and half years' graduate study to be very productive and enjoyable. He was a constant source of encouragement and advice. He taught me a great deal about the organization and presentation of information and improved my writing of this dissertation and other papers greatly.

Thanks also to my thesis committee, Professors James Arthur, John Dickey, Charles Feustel, Sallie Henry and Dennis Kafura, for their efforts in reading this dissertation and for much useful advice. I wish to thank Professors John Dickey and Charles Feustel for their continuing friendship.

This dissertation research was supported initially by a fellowship from the *European and American Students Association* in China, and then by various scholarships through the *Center for Innovative Technology* (CIT) of the Commonwealth of Virginia, and the Department of Computer Science at VPI&SU. Several travel grants from the CIT and the University Education Foundation enabled me to present several papers at various international computer science conferences. A supplemental research development award from the Graduate School at VPI&SU enabled me to attend the 11th International Conference in Software Engineering.

This dissertation is dedicated to my parents for their consistent encouragement during my graduate study in the United States of America and to my wife for everything.

Table of Contents

List of Figures and Tables

Chapter 1

Introduction

With the rapid development and wide application of computer science and technology, computer software has become the dominant and critical component of computer systems [Boe81]. The software production process is labor intensive and often employs ad hoc methods which result in very low productivity. With the need for larger and larger software systems, the correctness of programs cannot be guaranteed; the software product is unreliable; and the maintenance cost is disproportionately high. These symptoms characterize what is known as "the software crisis".

To overcome this crisis, software engineering has emerged to insert proven methodologies into the development process. However there is still a lack of a systematic development methodology throughout the computer software lifecycle. Software is usually viewed as a final product instead of a well defined process [Ost87]. To respond to such a challenge, we need both evolutionary and revolutionary ideas and methods.

Using formalisms in software development has been advocated and practised by many well-known computer scientists ([Hoa87], [Par77]). Using formalisms can help (1) to structure the software development more finely, (2) to guarantee the correctness of software since the methods are precise and rigorous, (3) to improve the software productivity since errors can be found earlier, and (4) hence to reduce the software costs [Bjo87]. There is a great deal of interesting and promising research in this direction ([Bal85], [BJ82], [CIP85], [GHW85]). Yet, these techniques are still immature and far from the final solution. The open research problems include how to develop techniques and methodologies to derive formal specifications from informal requirements, how to transform high level specifications into low level specifications, how to derive implementation from specification [SB82], how to combine specification with verification [Gog81], and ultimately how to automate the software development process.

This research is motivated by the above observations and analysis. The long term objective of this research will be to make software development more scientific and rigorous, thereby to achieve better software quality and to facilitate automated software production. Ongoing research contains the following major stages:

1. To investigate the feasibility of applying various formalisms to the software life cycle; to study the interfaces between the formalisms; and to build a model which integrates the advanced formal methods.

2. To search for transition (transformation and translation) techniques; to combine specification with verification; and to develop a methodology for implementing the resulting model.

3. To evaluate the model and the methodology, and to design a spectrum of specification language to facilitate these applications.

This dissertation contains the preliminary results of the first two research stages.

First, after an extensive analysis and comparison of various formalisms, a model and the associated strategy for integrating various formal specification and verification methods (predicate transition Petri nets, first order temporal logic, the algebraic, the axiomatic, the denotational and the operational approaches) has been developed. The model effectively incorporates foremost formalisms than any other models (the Automatic Programming Project [Bal85], the CIP Project [CIP85], the Larch Project [GHW85] and the RAISE Project [MG87]) and has the following distinctive features:

(1) Specifications are viewed both as a set of products and a set of well-defined steps of a process,

(2) Specifications (as a set of products) at different development steps are to be written and verified by different formalisms,

(3) Specification (as a process) spans from the requirement phase to the detailed design phase,

(4) Specification for both concurrent and sequential software is supported, and

(5) Specifications for different aspects (concurrent control abstraction, data abstraction and procedural abstraction) of a piece of software are dealt with separatedly.

Second, an intensive and indepth investigation of predicate transition nets and first order temporal logic results in:

(1) a design methodology for predicate transition nets, which incorporates the separate definition technique in Ada [Ada83] and state decomposition technique in Statechart [Har88] into the traditional transformation techniques for Petri nets, and therefore will significantly reduce the design complexity and enhance the comprehensibility of large predicate transition net specifications;

(2) the establishment of a fundamental relationship between predicate transition nets and first order temporal logic and the design of an algorithm for systematically translating predicate transition nets into equivalent temporal logic formulae. Therefore the goal to combine the strengths of both formalisms, i.e. to use predicate transition nets as a specification method and to use temporal logic as a verification method is achieved; and

(3) the discovery of a special temporal logic proof technique based on Hilbert-style logic system to verify various properties of predicate transition nets and the associated theorems. Thus temporal logic is effectively used as an analysis method for both safety and liveness properties of predicate transition nets.

The organization of rest of the dissertation is:

Chapter 2 introduces formal specification and verification methods. Different views and kinds of specifications are clarified and classified. The development histories of various formalisms are briefly reviewed.

Chapter 3 compares various formal specification methods through the formal definitions of Ada tasking.

Chapter 4 summarizes the results of comparison and presents a model and the associated strategy for integrating various formalisms.

Chapter 5 contains a design methodology for predicate transition nets. New transformation techniques are illustrated with examples.

Chapter 6 introduces the formal definitions of predicate transition nets and first order temporal logic. A fundamental relationship between the computation models of these two formalisms is then established and an algorithm for systematically translating predicate transition nets into equivalent temporal logic formulae is designed.

Chapter 7 contains a special proof technique of using temporal logic to verify various properties of predicate transition nets. Several theorems are presented. The proof technique is demonstrated through several well-known examples.

Chapter 8 concludes what have been achieved and discusses future research problems.

Chapter 2

Formal Specification and Verification Methods

2.1. Fundamentals of Specification

2.1.1. Views of Specification

There are two views of specifications: the *product* view and the *process* view. The product view focuses on the static aspect of specification while the process view emphasizes the dynamic aspect of specification. *Specifications*, in terms of products, are the precise descriptions of the result of each of the following software development phases: requirements engineering, system design, module design, implementation and testing. *Specification*, as a process, is the activity undertaken during the above software development phases to specify the expected properties and structure of the software precisely.

In this research, specifications are viewed both as processes and products with the process aspect to be emphazised because of its intrinsic impact on software development.

2.1.2. Kinds of Specification

There are several ways to classify specifications

(1) in terms of the media (notation), the specification being written [LB79]:

Specifications can range from informal (written in some combination of natural language, diagrams and standard mathematical symbols) to very formal (written entirely in a well-defined formalism or precisely defined language);

(2) in terms of the methodology utilized during the development of the specification [Bjo87]:

Specifications can be developed

formally -- all steps of the development are carried out formally by providing all necessary step by step proofs,

rigorously -- the development is formal but without actual formal proofs,

systematically -- the development lacks rigor being without the formal relations between stages of development;

(3) in terms of the chronological order in which the specifications are built [Hor82]:

Specifications can be divided into *requirement specification* , *system specification*, *local specification*, *implementation specification*, and *test specification* in chronological order corresponding to the phases of software development; and

(4) in terms of the software properties expressed in the specifications [LB79]:

A specification can also be classified as a *functional specification* which describes the behavior of a piece of software on its external environment or a *performance specification* which describes the constraints on the resources utilized and the speed requirements of a piece of software.

In this research, the formal development of formal functional system specification, together with local specification and structural specification (the integration of system and local specifications) is studied.

2.1.3. Requirements for Specification

Regardless as to whether specification refers to a process or a product, some form of specification method is required. Several requirements for specification methods can be identified ([BG79], [RS77], [Bar84]) which in turn can be summarized as:

(1) foundation -- formality, preciseness, expressibility,

(2) applicability -- abstractness, modularity (compositionality), parallelism / sequentiality, machine processability, verifiability, and

(3) comprehensibility -- understandability, learnability, and usability.

While all of the above requirements are desirable, they can hardly be unified in a single specification method, and actually some of the requirements tend to be contradictory. Therefore, compromise needs to be made in selecting a specification method.

In this research, we give priority to the following requirements for a specification method: formality, preciseness, abstractness, modularity, parallelism, learnability and usability for the following reasons:

(1) formality and preciseness are the major advantages of formal specification methods,

(2) abstractness and modularity are the necessary requirements for any practical development methodology,

(3) parallelism is the major characteristic of most critical software systems and is the most challenging research problem for specification methods as evidenced in Chapter 3, and

(4) any useful specification method should be learnable and applicable to practical software development.

2.2. Fundamentals of Verification

The purpose of verification is to ensure the reliability of software. Two other closely related terminologies (and research areas) for the same purpose are validation and testing. These three concepts have occurred in various software engineering literatures and implied different meanings in various situations ([Deu82], [Hau84], [Win86]). We distiguish them by the following definitions:

(1) *Validation* is the activity undertaken to ensure that a specification and/or a final product meets its original informal requirements. Therefore human judgement is required. Validation can take various forms such as inspection, testing, symbolic execution [Hau84], paraphrasing, prototyping and simulation [Bal85];

(2) *Verification* is the activity undertaken to ensure that the product of each stage in the software development is consistent (correct) with respect to the specification of a previous stage. Verification takes following forms: proving ([Bak80], [BBFM82], [LS87]), and term rewriting ([Jau85], [Les87]). By its nature, verification is always formal, therefore a formal specification is a prerequisite for verification (a formal verification of an informal specification is meaningless);

(3) *Testing* is the activity undertaken to reveal possible errors in the implementation of requirements, specifications, or design by executing the implementation for a chosen set of test cases. Therefore a computer execution is always required. There are many testing techniques such as exhaustive, branching, path-covering, structural and functional techniques ([GG75], [How86]).

Since building a formal specification transformation model is our current primary concern, we concentrate on formal verification methods, especially their applications to software development phases prior to the program implementation stage.

2.3. Overview of Formal Methods

Over the past twenty years, formal methods for the specification and verification of computer software have emerged explosively. Most well-known methods are classified to be operational, axiomatic and denotational approaches, and their primary application areas have been the formal definition of the semantics of programming languages [Luc78] and the verification of the correctness of compilers ([BBFM82], [LS87]). On the other hand, algebraic approach has become widely used for the specification and verification of abstract data types ([EM85], [Fla79], [Gue81]). However the application of the above approaches is at present largely restricted to the development of sequential software systems [Bar84]. Research on formal methods for the specification and verification of concurrent and distributed software systems has been very active in recent years

[Bar85]. Two most promising and investigated methods are Petri nets ([Pet81], [Rei85a], [BRR87]) and temporal logic ([MP81], [Pnu86]).

In the following sections, we review the histories and applications of the above mentioned formal methods which are to be integrated in our software development model (the formalisms themselves will be introduced and illustrated in Chapter 3).

2.3.1. The Operational Approach

The *operational approach* is the earliest formalism developed in specifying software [Luc78]. It is based on the theory of computation process (state transitions). The meanings of programs are defined in terms of state transitions upon an *abstract machine* which interprets the programs. The machine is intended to be so simple that no possible misunderstanding can occur [Oll74]. The best known formalism of this approach is VDL ([Lee72], [Weg72]) which was used to define the formal semantics of several conventional sequential programming languages. The operational approach has also been extended to handle concurrency [Plo82]. Labelled transition systems and synchronization trees form a basic process model to provide structural operational semantics to a variety of concurrent programming languages. A labeled transition has the general form:

$$S_1 \rightarrow_\alpha S_2$$

where S_1 and S_2 are syntactical structures of a language and α is the action which transforms S_1 to S_2. The formal definition of a language consists of a set of transition rules of the form:

$$\frac{S_1 \rightarrow_\alpha S_2}{T_1 \rightarrow_\alpha T_2}$$

where S_1, S_2, T_1 and T_2 are language structures.

The formal semantics of both Ada and CSP were specified by using the *structural operational approach* (where abstract syntax and labeled transitions were used to overcome the over-specification problem of traditional operational approach) in [Li82] and [Plo82] respectively. Concurrency was modeled by the interleaving semantics by introducing the rules of following form:

$$\frac{P \rightarrow_\alpha P'}{PQ \rightarrow_\alpha P'Q} \text{ and } \frac{Q \rightarrow_\alpha Q'}{PQ \rightarrow_\alpha PQ'}$$

, i.e. by reducing the parallelism of a program into sequentialized execution of the components of the program.

In Li and Lauer [LL85], a general method for treating true (not interleaved) concurrency by using the structural operational approach was presented. The key was to use the vector of actions instead of a single action in labeled transitions. The rule for true concurrency thus had the following form:

$$\frac{P \rightarrow_\alpha P' \text{ and } Q \rightarrow_\beta Q'}{PQ \rightarrow_{(\alpha,\beta)} P'Q'}$$

The concurrent semantics of the COSY (COncurrent SYstems) specification language (in which a program consisted of a number of cyclic concurrent executing pathes) was specified by using this method as a demonstration.

The advantage of the operational approach is that specifications written using it are directly executable so that prototyping and simulation can be carried out. The operational approach can be very useful in the derivation of an implementation from a formal specification. Since the style of description is very much like that of a programming language, it is easy to learn and use. There are several well-known projects based on the operational approach:

- Zave's PAISLey (Process-oriented, Applicative, Interpretable Specification Language) [Zav82] and
- Balzer's Automatic Programming Paradigm [Bal85] at the University of Southern California.

The disadvantage of the operational approach is that there is no effective verification technique except testing techniques for it due to its lack of mathematical foundation [BBFM82]. Another drawback with the operational approach is its low level. Specifications in this approach usually contain much irrelevant detailed information (i.e. over-specification).

2.3.2. The Denotational Approach

The *denotational approach*, previously named *mathematical semantics*, is based on Lambda-Calculus [Bare81] and Scott's *domain theory* [Sto77]. In the denotational approach, entities in software are mapped onto mathematical objects (list, tuple, set, function, predicate etc.) In such a definition, three parts are identified:

(1) syntactic domains,

(2) semantic domains, and

(3) semantic functions which map objects in syntactic domains to those in semantic domains.

The denotational approach primarily developed for specifying the semantics of sequential programming languages ([MS76], [Ten77], [Sto77], [Gor79]) has been advanced to such a level that it forms the basis of well-proven sequential system specification and design methodologies such as VDM ([BJ82], [Jon86]). It is also extended to handle concurrency [Plo76] and [Smy78] by using power domain

$$R = S \rightarrow \wp[S + (S \times R)]$$

where S is a domain of states, R is a power domain recursively defined as a function from S to the power set $\wp$ of state domain.

In Berry [Ber85], it was shown that a denotational semantics could be systematically constructed from a VDL interpreter definition of the same language such that a more abstract functional meaning from states to sets of states was obtained. The obtained denotational semantics was capable of modeling arbitrary interleaved access to shared memory and was finitely expressible with finite denotations.

Bakker and Zucker [BZ83a] proposed a model allowing a unified and rigorous definition of the semantics of concurrency. The mathematical model introduced processes as elements of process domains which were obtained as solutions of domain equations in the sense of Scott [Sco76] and Plotkin [Plo76], and then were used as meanings of concurrent statements in a language. Three main concepts were treated: parallelism (arbitrary interleaving of sequences of elementary actions), synchronization and communication. These notions were embedded in languages which also featured classical sequential concepts such as assignment, tests, iteration or recursion, and guarded commands. In the definition, a sequence of process domains of increasing complexity was used. The language discussed included Milner's CCS [Mil80] and Hoare's CSP [Hoa78].

In Bruin and Bohm [BB85], a denotational semantics for a simple language (which allowed the creation new of processes dynamically) of dynamic networks of processes was developed. Processes in the language could only communicate via channels without other variable sharing mechanisms. An infinite domain was used to model the history of a possibly infinite sequence of the values transmitted through channels.

The denotational approach is an ideal method to specify and verify the functionality of sequential software systems [BJ82]. It is precise and mathematical so that formal proofs and transformation can be constructed [Pol81]. It is abstract and compositional so that stepwise refinement and modular design can be achieved ([BJ82], [Jon86]). Various verification techniques based on denotational semantics such as fixpoint induction, subgoal induction and structural induction can be found in [LS87]. However, the denotational approach to the definition of concurrency is usually involved with the construction of complicated power domains, but is not general enough to deal with real concurrency (instead of interleaving). It is hard to learn and use the pure denotational approach in practical software specification.

The most significant application of denotational approach is in the development of VDM (Vienna Development Method) which has become the standard formal methodology for software specification in Europe ([VDM87], [VDM88]). One major project based on VDM is the RAISE (Rigorous Approach to Industrial Software Engineering) Project at Dansk Datamatik Center [MG87]. The RAISE project aims at the construction of a mathematics based method for the development of software in industry, and a collection of computer based tools to support such a method.

2.3.3. The Axiomatic Approach

The *axiomatic approach* is based on the principles of Hoare logic [Hoa69] which has a set of axioms and inference rules associated with program concepts. The meaning of a program/language construct (expression, statement etc.) is defined by the *pre-conditions* which must be satisfied initially and *post-conditions* which must be satisfied after the evaluation or execution of the construct. Since its invention, Hoare logic has been extensively and intensively studied; it has been applied to programming language definition [HW73], and more importantly, it has become a major verification method for sequential software systems ([BBFM82], [LS87]). An excellent survey of the applications of Hoare logic can be found in [Apt81]. One of the best known formalisms of this approach is *predicate transformer* which is first developed by Dijkstra [Dij75] and later augmented by Gries [Gri81]. Extension of Hoare logic to deal with concurrency has been undertaken by several researchers ([OG76], [Lam80], [Bro85]).

In Owicki and Gries [OG76], a language for parallel programming with a **cobegin** statement:

cobegin S_1 || ... || S_n **coend**

and an **await** statement:

await B **then** S

was presented and was specified by the axiomatic approach. Two new axioms for the above statements were introduced respectively:

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}}$$

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, \ldots, \{P_n\}\ S_n\ \{Q_n\}\ \text{are interference-free}}{\{P_1 \wedge \ldots \wedge P_n\}\ \textbf{cobegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{coend}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

The first axiom reflected that **await** statement was noninterruptable and the second axiom stated that as long as S_i ($1 \leq i \leq n$) were interference-free (the preconditions P_i would not be changed by any **await** statement in any S_i ($j \neq i$)), the individual (pre and post) conditions of each statement S_i ($1 \leq i \leq n$) guaranteed the combined (pre and post) conditions for the **cobegin** statement.

Lamport [Lam80] generalized the Hoare's logical system for specifying sequential programs to concurrent programs by giving a new interpretation to the assertion

{P} S {Q}

in the following way:

if execution was begun *anywhere* in S with the predicate P true, then executing S would have P true while control was inside S, and would make Q true if and when S terminated.

This new interpretation was strong enough to specify the safety properties [Lam89] of concurrent programs. However, it was not powerful enough to deal with liveness properties [Lam89] of concurrent programs.

In Brookes [Bro85], a semantically-based axiomatic treatment of a parallel programming language with shared variable concurrency and conditional critical regions was described. The semantics of the language was specified by using the structural operational approach and a class of assertions for expressing properties of commands was defined from the semantic structures (the tree structure) of labeled transition system. The syntactic operations over assertions corresponding precisely to syntactic constructs of the language were designed. Axioms and inference rules based on the assertion operations were developed. The evaluation of expressions and the execution of assignment statements were assumed to be atomic. The parallel composition rule:

$$\frac{\Gamma_1 \text{ sat } \phi \quad \Gamma_2 \text{ sat } \psi}{[\Gamma_1 || \Gamma_2] \text{ sat } [\phi || \psi]}$$

did not guarantee to produce *safe assertion* $[\phi || \psi]$ (successful termination condition), even if the component assertions: ϕ and ψ were safe (terminating).

The axiomatic approach is appropriate for specifying and verifying the properties of sequential programs [Apt81]. The method is abstract and easy to understand -- meeting well the principle of least astonishment. The principle is as simple as to make it suited for use by programmers at a comparatively low developmental level. When a proof system is constructed, correctness proving can be performed automatically. While the axiomatic approach facilitates the verification process, it reveals almost nothing about the actual concurrent control structure of the specified software; therefore it alone cannot constitute a total specification paradigm. Another problem with the axiomatic approach is that it is very difficult to construct invariant assertions from an arbitrary piece of program.

2.3.4. The Algebraic Approach

The *algebraic approach* is based on Birkhoff's *heterogenous algebra* [EM85] and was first developed by Guttag, Zilles and ADJ-group[1] ([Gut80], [LZ75], [GTWW79]). In the algebraic approach, every (abstract) data type is treated as an algebra which consists of:

(1) a set of types indicating values in domains and ranges,

(2) a set of operations upon the domains, and

[1] ADJ-group refers to Guttag, Thatcher, Wagner and Wright.

(3) equations relating various operations.

The first two sets together are usually called a *signature*. The meanings of entities (data structure, procedure, or process) are given implicitly via the various relationships among the operations. Programs are viewed as entities of the above algebra.

In Zamfir [Zam87], a category of continuous many-sorted algebras called *parallel-nondeterministic* algebras was defined by analogy with the *initial algebraic semantics* of ADJ group [GTWW79]. It was shown that there was a class of free abstract objects called *parallel nondeterministic* terms or *diamonds* in the continuous parallel-nondeterministic algebras. Then the parallel and nondeterministic behaviors of communicating computing systems were rigorously formulated as sequences of rewriting on diamonds.

The algebraic approach is an excellent formalism for studying abstract data types and has been actively studied in recent years ([Gut80], [NR85]). Since it is mathematically well-defined, formal analysis can be performed and proofs or correctness can be constructed over the specifications using it. Various transformations can be made automatically according to algebraic laws and equations in the same specification framework [Mus80]. Abstraction and modularity provide a means by which information hiding can be achieved [EW85] and the method is very useful for describing the interfaces between modules [CIP85]. However it is hard to deal with concurrent control structures since only a static relationship is specified.

Verification techniques for the algebraic approach are algebraic transformations and term rewriting ([NHN78], [GTWW79], [Fla79]). Experimental reasoning systems for algebraic specifications have also been built ([GHM78], [LW82]).

There are several major projects based on algebraic approach - the Larch project at MIT [GHW85] and the CIP (Computer-aided, Intuitition-guided Programming) project at the Technical University of Munich ([CIP85], [CIP87]).

The Larch family of specification languages consists of the Larch Shared Language and the Larch interface languages particular to specific programming languages. The Larch Shared Language is mainly algebraic, and used to write equations for defining the relations among operations and giving meaning to equality among terms appearing in interface specifications. The Larch interface languages are used to specify program modules and to provide information needed to write programs that use these modules.

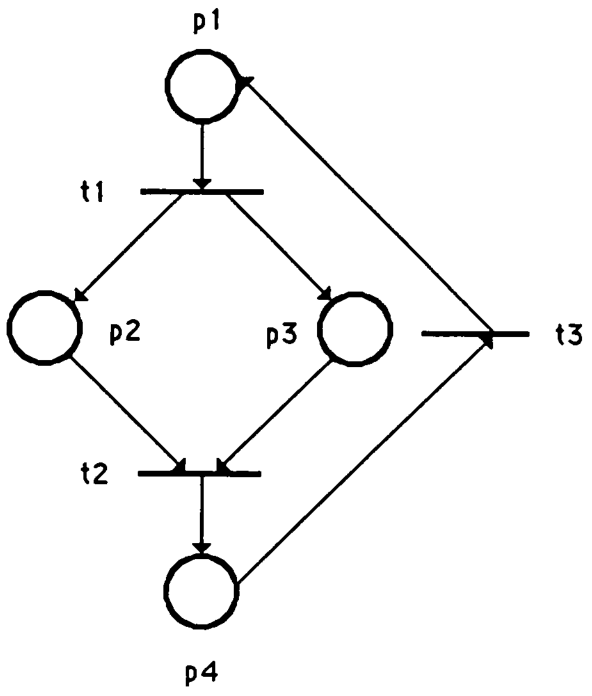
The CIP project consists of two components: a wide spectrum language CIP-L for writing specifications, and a transformation system CIP-S to assist users in this methodology. CIP-L provides separate mechanisms for writing specifications: a scheme language for the control structure and algebraic types for data structures.

2.3.5. Petri Nets

Petri nets are named after their inventor Petri [Pe62]. A *Petri net* structure is a triple (P,T,A) where

- P is a set of *places* which are represented by circles,
- T is a set of *transitions* represented by bars (boxes), and
- A is a set of *arcs* which connect places to transitions and transitions to places and is called the set of flow relations. These concepts are shown in Figure 2.1.

A Petri net is *marked* with dots at various places Figure 2.2. The execution of a marked Petri net is by *firing* a sequence of *enabled* transitions, all of whose incoming places have more dots than the number specified by the corresponding flow relations. The effect of firing a transition is to take away the number of dots as specified by the incoming flow relations from all incoming places and to put the number of dots as specified by the outcoming flow relations into all outcoming places Figure 2.3. Disjointly enabled transitions can fire simultaneously. A timed Petri net is one in which timing information is associated with each transition. The timing information specifies



P = { p1, p2, p3, p4}
T = { t1, t2, t3}
F = { (p1,t1), (t1,p2), (t1,p3), (p2,t2),
(p3,t2), (t2,p4), (p4,t3), (t3,p1)}

Figure 2.1 - Basic Concepts of Petri Nets

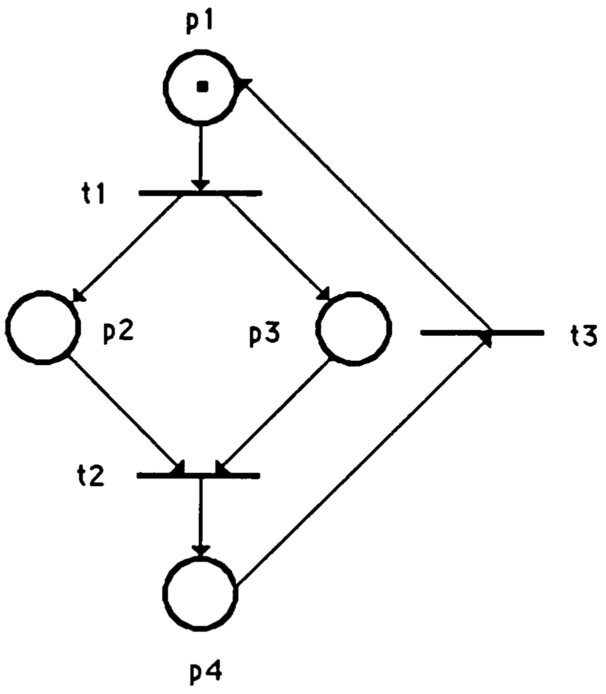


Figure 2.2 - A Marked Petri Nets

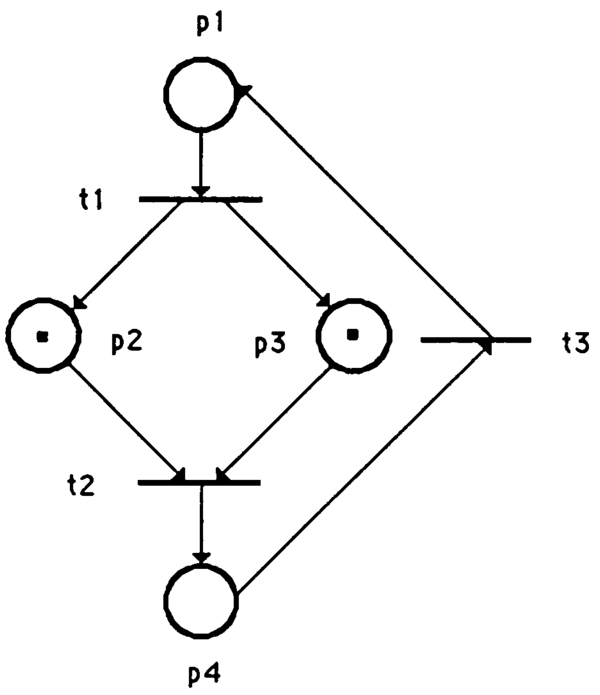


Figure 2.3 - The Marking after the Firing of t1 in Figure 2.2

the minimum number of time units which have to elapse before the associated transition can fire and the maximum time units before which the associated transition must fire. There are several classes of Petri nets with increasing expressive power:

- *condition-event* nets (CE-nets) [Thi87] in which there is at most one token in each place at any time (capacity one),
- *place-transition* nets (PT-nets) [Rei87c] in which there can be more than one token in any place,
- *high-level* Petri nets which have structural places and can further be divided according to labelling conventions of flow relations:
  (1) *predicate-transition* nets in which labels of flow relations are symbolic sums,
  (2) *colored* nets in which labels of flow relations are functions, and
  (3) *transition-relation* nets in which labels of flow relations are relations (a generalization of functions).

In Petri nets, the control structures and causal relations between various components of a software system is modeled and the meaning of the system is simulated by the behavior of the nets.

Petri nets are well suited for specifying both sequential and concurrent control structures of software systems. There are also effective analysis techniques for Petri nets ([Lau87], [MV87], [HJJJ85]). Petri nets have a simple graphical representation which makes them especially attractive and easy to understand. There is an overwhelming number of the applications of Petri nets to computer systems - both software and hardware ([Pet81], [BRR87]) in the following areas: software engineering [Rei87a], data bases [Vos87a], communication protocols [Dia87], human-machine interaction [Obe87], office automation [Vos87b], computer architecture [Bae87] and performance evaluation [PN87]. A recent survey [Roz87] has shown there are more than 2000 Petri nets related publications over the past twenty years.

However to avoid producing very large, unstructured descriptions of computer systems and to avoid large computations to analyze the nets for some properties, Petri nets should not be used at very low level. High level Petri nets [BRR87] is needed to achieve more concise representation and to provide more information ([Bar84], [RS77]).

2.3.6. Temporal Logic

Temporal logic is a form of *modal logic* [RU71] which deals with time concepts. It is first order logic [And86] plus additional temporal operators such as *always* (represented by a square □), *sometimes* (represented by a diamond ◇), *next* (represented by a circle ○) and *until* (represented by a U). The application of temporal logic to computer programs was pioneered by Pnueli [Pnu77]. In temporal logic, the meaning of a program is specified by temporal formulas over the state transitions of computation sequences. The most common form of temporal logic is linear time temporal logic [MP81] in which only one unique successor state exists in a computation sequence. Other forms of temporal logics are:

- *branching time temporal logic* [CE81]: in which each state in the computation sequence may have more than one successor state so that a computation sequence can be a tree structure, and
- *interval temporal logic* [MM83]: in which temporal formulae are interpreted over a specified time interval so that real time systems can be studied.

Temporal logic is an ideal method for studying concurrent and distributed systems and has attracted many researchers around the world in recent years ([Pnu86], [Vog88]).

In Pnueli [Pnu81], the basic elements of temporal logic were introduced in a particular logic framework DX. The usefulness of temporal logic in specifying the properties of concurrent programs was demonstrated. It was argued that temporal logic was the first formalism which could express the liveness properties effectively.

In Lamport [Lam83], a method for specifying concurrent program modules was described based upon temporal logic. Temporal assertations on program states were developed to make the specifications easier to understand. Labeling of state transitions to distinguish external actions from internal actions was introduced. Actions sets were designed to achieve the compositionality of temporal specification. The effort was made to combine the best features of state machine and temporal logic methods. A specification of the alternating-bit communication protocol consisting of three modules was given.

Temporal logic has been widely used to specify and verify program properties such as *safety* (partial correctness, clean behavior, mutual exclusion, deadlock freedom) and *liveness* (total correctness, accessibility, fairness) [MP81]. It is especially useful in specifying and verifying concurrent programs, distributed systems and network protocols ([Hai82], [Lam83], [Pnu86]). Temporal logic is not only a ideal method for specifying various properties of concurrent and distributed systems but is also one of the most significant verification methods for concurrent systems. There are several widely investigated proof techniques for temporal logic:

- the finite state automata technique for propositional temporal logic [VW86],
- the tableaux technique [Fit72],
- the Hilbert and Gentzen technique [MP81], and
- temporal resolution technique [AM86].

The automata-theoretic approach to automatic verifying propositional temporal logic specifications was first introduced in [CES83] and further improved in [Brow86], [VW86] and [V87]. The technique is very effective for a class of concurrent programs called finite-state programs, i.e. programs in which the variables range over finite domains. Such programs are very common in the communication and synchronization protocols, and can be described by propositional temporal logic formulae. The temporal formulae can then be translated into equivalent finite-state automaton specifications which in turn can be verified by an algorithm called *model checker*.

Other proof techniques will be discussed in Chapter 7.

The problem with temporal logic is its lack of compositionality [Bar84] in the sense that there is no effective technique for deriving a specification for a large system from the specifications of its components, though this problem has recently been intensively investigated ([Wol82], [MW84], [BKP84]).

It is usually very difficult to write a temporal logic specification for a large software system; moreover it is inefficient and unnecessary to apply temporal logic to a sequential software system.

In later Chapters, predicate transition nets will be studied in depth.

Chapter 3

A Comparison of Formalisms through Ada Tasking

3.1. Introduction

In the previous Chapter, the histories and applications of various formalisms were overviewed. However in to order to compare the formalisms, a common source where the formalisms have been applied is needed; the programming language Ada[2] is such a rich source ([BBH80], [BO80], [Che84], [DZ83b], [Ger82], [Li82], [MZGT85], [PD82]).

There are many experimental formal definitions of Ada subsets, especially the concurrent portion -- tasking, which is also considered to be the hardest part of the language to describe. The formalisms used include: operational semantics, denotational semantics, axiomatic semantics, Petri nets, and temporal logic. The definitions differ greatly in their appearances and abstract levels from very formal and mathematical to semi-formal and graphical, and from very high level and user oriented to very low level and implementation oriented.

Since the definition of a language serves many different purposes (verification, implementation etc.) and different professionals (language designers, language theoreticians, language implementers, system programmers and application programmers), let us to take a close look at various formal definitions. It will be especially useful to make a comparison of various definition methods and to point out their relative merits so that they can be applied appropriately and effectively.

In this Chapter, several formal definitions of Ada tasking are briefly reviewed so that various formalisms used in the definitions can be examined and their relative merits can be compared.

[2] Ada is the trademark of the Department of Defense of USA.

3.2. Basic Concepts of Ada Tasking

Task units are the only form of program units of Ada which execute concurrently. A task unit consists of a specification and a body, with the syntax shown below [Ada83]:

```
task_declaration ::= task_specification;
task_specification ::=
    TASK [TYPE] identifier [IS
         {entry_declaration}
         {representation_clause}
    END [task_simple_name] ]

task_body ::=
    TASK BODY task_simple_name IS
         [declarative_part]
    BEGIN
         sequence_of_statements
      [exception
         exception_handler
         {exception_handler}]
    END [task_simple_name];
```

3.2.1. Task Creation, Execution and Termination

Tasks can be defined by task declarations or by defining task objects in terms of existing task types. They are created by elaboration of the declarations or by evaluation of allocators respectively. The execution of a task is activated by the elaboration of the declarative part of its body which defines the execution of any task of the corresponding task type. Execution of different tasks proceeds in parallel.

Each task has at least one parent on which it depends. The direct dependency relation is defined by following rules [Ada83]:

(1) The task designated by a task object which is the object, or a subcomponent of the object, created by the evaluation of an allocator depends on the parent that elaborates the corresponding access type definition; and

(2) The task designated by any other task object depends on the parent whose execution creates the task object.

Therefore, a non-cyclic tree structure of dependency relations is expected.

A task may *complete* its execution in two situations:

(1) when it has finished the execution of its body; or

(2) when an exception is raised during its execution.

The *termination* of a task takes place:

(1) when it has completed its execution and it does not have any dependent task; or

(2) when it has completed its execution and all of its dependent tasks have terminated.

3.2.2. Synchronization and Communication (Rendezvous)

Tasks can synchronize and communicate each other through entry calling and acceptance, which is known as *rendezvous.* The syntax of entry declaration, entry call statement and accept statement is shown below [Ada83]:

```
entry_declaration ::=
   ENTRY identifier [(discrete_range)] [formal_part];

entry_call_statement ::=
   entry_name [actual_parameter_part];

accept_statement ::=
   ACCEPT entry_simple_name [(entry_index)] [formal_part] [DO
         sequence_of_statements
   END [entry_simple_name] ];

entry_index ::= expression
```

Synchronization takes place in the following situations:

(1) if the calling task issues an entry call statement before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is *suspended* until the calling is accepted; or

(2) if a task reaches an accept statement prior to any call of that entry, the execution of that task is *suspended* until such a call is received.

The *communication* takes place when both calling and called tasks have reached the corresponding entry call statement and an accept statement respectively. The accept statement is then executed by the called task. The interaction is known as a *rendezvous*. Entry queues are used when more than one task call the same entry before a corresponding accept statement is reached.

3.2.3. Real-Time Construct (Delay Statements)

The execution of a task can be delayed by executing a delay statement which has the following syntax [Ada83]:

```
delay_statement ::= DELAY simple_expression;
```

The type of simple_expression must be of the predefined fixed point type DURATION and its value is in seconds.

3.2.4. Select Statement

Both accept statements and entry call statements can be made selective. The syntax of select statement is [Ada83]:

```
select_statement ::= selective_wait | conditional_entry_call |
                     timed_entry_call
```

The alternative accept statements can be guarded; an alternative is said to be *open* if it is not guarded or if the guarded condition becomes true. For an entry call issued by a conditional entry call, if a rendezvous is not immediately possible, the call is then canceled. When an entry call is issued by a timed entry call, and a rendezvous is not started within a given delay, then the call is canceled.

3.2.5. Other Task Related Issues

Other task related issues such as *priority*, *abort* statement, *shared variables*, and *exception handling* will not be discussed here. They are either not included in all the following formal definitions or are too difficult to be formalized ([Coh86], [Lam85]).

3.3. Various Formal Definitions and Their Coverages

There are various formal definitions of the Ada tasking concept for different Ada versions ([Ada79], [Ada81], [Ada83]). The formalisms used can be classified as: the operational approach ([BBH80], [Li82]), the denotational approach ([BZ83b], [LBj80]), the axiomatic approach [Ger82], Petri nets ([Che84], [MZGT85]), and temporal logic [PD82].

In the following sections, we will discuss the coverage of different definitions and the relative merits of different approaches.

3.3.1. The Operational Approach to Ada Tasking

In Belz et al [BBH80], a multi-processing implementation-oriented formal definition of preliminary Ada was described in SEMANOL.

3.3.1.1. The SEMANOL System, Definition and Notation

The SEMANOL system consists of:

(1) a meta language for defining formal operational definitions of the syntax and semantics of programming languages; and

(2) an interpreter which executes SEMANOL metaprograms.

A SEMANOL metaprogram definition of a programming language L has the following parts:

(1) the context-free syntax of L (in a BNF-Like notation);

(2) the static semantics; and

(3) the dynamic semantics of control defined by both applicative SEMANOL definitions and executable imperative SEMANOL command statements.

The SEMANOL notations are very similar to those in a high level procedural programming language. Its style is procedural and constructive. The following commands are most frequently used:

(1) ASSIGN_VALUE! x=y has the normal interpretation of assignment statement as in most high level programming languages;

(2) COMPUTE! x causes (meta) evaluation of the expression x; and

(3) CO-COMPUTE! causes each of the processes to execute concurrently.

3.3.1.2. The SEMANOL Definition of Ada

In [BBH80], both sequential and concurrent structures of Ada [Ada79] were defined. The definition was given in two parts labelled by SEMANTIC-DEFINITIONS and CONTROL-COMMANDS respectively. The first part provided the semantics of elaboration and evaluation (definitions in this part started with the key words DFs and PROC-DFs) and the second part controled the execution of metaprogram. The excerpt from the SEMANOL definition is:

```
CONTROL-COMMANDS:

BEGIN
  ASSIGN-VALUE! Ada-program-text =
                             lexically-transformed(total program)
  ASSIGN-VALUE! program-tree = CONTEXT-FREE-PARSE-TREE
                  (Ada-program-text,[complete-compilation])
  IF [program-tree] is-not-statically-valid THEN STOP!
  COMPUTE! initialize-system-state (semanol-processor-list)
  CO-COMPUTE! control IN-PROCESSES (semanol-processor-list)
END.
```

The first two statements create a syntactically correct abstract tree from an Ada program; then static semantics checking is performed followed by the initialization of system state (tables, semaphores etc.); finally, CO-COMPUTE! activates the concurrent execution of meta-processes.

Each meta-process has the following definition:

```
PROF-DF control
  WHILE TRUE
  BEGIN
    COMPUTE! op-kernel
    WHILE sequential-processing-may-continue
    BEGIN
      COMPUTE! effects-of (current-step)
      COMPUTE! update-to-successor(current-step)
    END
  END
```

a process executes sequentially until one of the following occurences happens: a rendezvous, an abort, an exception or an interrupt; then the process calls the kernel by executing COMPUTE! op-kernel.

The definition acts like a multi-pass compiler and the op-kernel serves as the kernel of an operating system. The concurrect execution of tasks is modeled by a multiprogramming system running on a fixed number of processes with a shared memory. Rendezvous between tasks is simulated by traditional P and V operations for synchronization, and ordinary parameter passing for communication. A piece of the definition is:

```
PROC-DF entry-call-effect-of (step)
BEGIN
  ASSIGN-VALUE! entry-name = dynamic-entry-name (name (step))
  ASSIGN-VALUE! owner = owner-task (entry-name)
  COMPUTE! locked-enque ([:current-task,param-part(step):],
           "on" entry-q (entry-name))
  COMPUTE! update-to-successor (current-step)
  ASSIGN-VALUE! non-sequential-condition = 'entry-call'
END.
```

```
PART-DF op-kernel
  =: complete-entry-call IF [non-sequential-condition]
                             is-entry-call

PROC-DF complete-entry-call
  P! semaphore (delay-list)
  P! semaphore (entry-gates(owner))
  IF [entry-name] is-open THEN
    BEGIN
      COMPUTE! enque-ready-task(owner)
      COMPUTE! close-entries(owner)
      V! semaphore (entry-gates(owner))
      COMPUTE! locked-decrement(running-tasks)
      COMPUTE! scheduler
      RETURN-WITH-VALUE! NIL
    END
  V! semaphore (delay-list)
  COMPUTE! scheduler.
```

The execution of entry-call-effect-of is invoked from the COMPUTE! effects-of in the meta-process definition which controls the calling task, and it puts the name of the calling task in the entry queue; sets the condition to be 'entry-call'; and leaves the sequential execution (disconnects the process from the calling task). Then the op-kernel in the above definition starts the following actions: prevents the multiple acceptances by using two semaphores in case of a select statement has an open delay statement and/or open accept statements; puts the called task in the ready queue when the entry is open; calls the scheduler (COMPUTE! scheduler) for reassigning the process to some task (possiblly to the called task).

The definition is at a very low level and has many implementation related decisions such as multi-stack simulation of environment, P and V operations for synchronizations and a fixed number of multi-processes with a shared memory model for execution. These are the typical drawbacks of operational approach which complicates the understanding and limits the application of the formal definition. In [BBH80], no verification issues were addressed so that the correctness of the definition and the programs could not be assured. Though exception raising was briefly treated, the definition was only based on the very simple case of concurrency.

3.3.2. The Denotational Approach to Ada Tasking

In Lovengreen and Bjorner [LBj80], a formal model of the Ada [Ada79] tasking was developed in VDM which was highly pragmatic. A more formal treatment was given in [BZ83b].

3.3.2.1. The VDM META-IV

The VDM approach uses a special meta-language called META-IV ([BJ78],[BO80]) which is a syntactically sugared Lambda-Calculus combined with an imperative style and an exit mechanism. It is not purely denotational and actually is a hybrid of the denotational and operational approaches, but by convention belongs to denotational approach [BJ78].

A VDM definition of a programming language L consists of:

(1) an abstract syntax (syntactic domains) which is an abstraction and refinement of the concrete BNF-Like syntax of L;

(2) semantic domains that are mathematically well-defined objects with types of trees, lists, sets and mappings;

(3) auxiliary functions (is-well-formed functions) which are predicates for checking the legality of the programs and define the static semantics of the L; and

(4) interpretation functions which give the meanings of the execution of programs and define the dynamic semantics of L.

Auxiliary functions and interpretation functions are called semantic functions and are mappings from abstract syntax to semantic domains.

The VDM META-IV notations are very close to the notations in high level programming languages such as Pascal and Ada. For a tree structure domain definition, the symbol :: is used; otherwise, = is used as the definition symbol. The ***mk-*** operator shows the components of a composition structure and the symbol = = > is used as a convention in the ***type*** clause of a semantic function to represent the state changing.
The ***def*** x: f; e(x) combines the effects of first evaluating f and then e(x). The ***let*** command is often used in semantic functions as an imperative assignment statement. McCarthy style case statement is most often employed in the semantic functions. The ***exit*** mechanism acts like a structured goto statement [BJ78] and serves as a substitution for the traditional *continuation* concept.

3.3.2.2. The VDM Definition of Ada Tasking

In [LBj80], the META-IV was extended with a process concept and a very simple communication and synchronization concept both based on the ideas of CSP [Hoa78]. The process concept had a fixed number of process types (called processors) upon which a dynamically varying number of process instances were based. The communication concept used input and output constructs based on the value passing hand-shaking idea of CSP.

The definition has a meta-process model which consists of the fixed processes: SYSTEM, STORAGE, TIMER, CLOCK and MONITOR. The SYSTEM process is responsible for the initialization and termination of whole system. The STORAGE process simulates the shared memory. The TIMER and CLOCK facilitates the real-time tasking. The MONITOR is the key unit of the model which performs all task creation, termination and interaction. All dynamic processes are of the TASK type which describes the sequential interpretation of an Ada task.

Some simplified semantic functions for rendezvous is shown below to provide some flavor of the VDM definition, for a complete definition refer to [MZGT85] and [BO80].

```
int-entry-call(tv,ev,parmassoc,ecmode) env =
  LET req = MK-EntryCall(tv,ev,parassco,ecmode) IN
 (DEF MK-EndCall(callresult):
      monitor-call(req);
      RETURN (callresult)   )
TYPE: EntryCall --> ( ENV ==> (CALLED|CLOSED|EXPIRED)
```

The interpretation function for entry call has four parameters: tv (task value containing task identifier), ev (entry identifier), parmassoc (parameters of entry call statement), and ecmode (mode indicating whether the call is conditioned or timed); and is interpreted in the environment env which associates variable names with various attributes. Upon the receipt of an entry call, the monitor is invoked by monitor-call(req) statement. The monitor process will put the above parameters in appropriate queues until a rendezvous occurs, then it will return the call result to the process simulating the calling task.

```
int-accept(MK-accept(ename,stmtl)) env =
  (DEF MK-(ev, ): eval-EntryName(ename) env;
   LET req = MK-ACCEPT(ev) IN
   DEF MK-START-MEETING( ,parmassoc) : monitor-call(req) ;
   int-acceptbody(stmtl,parmassoc)                      )
TYPE: Accept --> (ENV ==> )
```

This interpretation function will call the monitor process once an accept statement is encountered with the name of the task containing the statement. The elaboration of monitor-call and int-acceptbody is noninterruptable marked by ;. The name is also put in an appropriate queue by the monitor until a corresponding call occurs; then the body of the accept statement is executed by: int-acceptbody(stmtl, parmassoc)
with the actual parameters from the calling task passed by the monitor.

This definition is at a low level and is dependent on design details. The model uses a MONITOR to create tasks and control the communications between tasks which makes it only applicable to a central controlled environment. The definition is comparatively lengthy and difficult to understand. Though the definition covers almost all aspects of the tasking concept except real time issues, no verification issues are discussed.

3.3.2.3. The Concepts of Process and the Semantics of a CSP-Like Language

In [BZ83b], a more formal treatment of Ada [Ada81] tasking in denotational semantics was given, based on the concept of *process* domains. A *uniform process* is a tree-like construct which abstracts the structure of the sequences of elementary actions generated during the execution of a program. Example of finite processes is:

$\{ <a, \{ <b,p>, <c,q> \} > \}$

where a, b, c are elements of an alphabet A and p, q are uniform processes defined upon A. One distinguishable uniform process is the ***nil*** process denoted by p_0. Operations on finite processes are recursively defined as following:

- ***Composition*** "•" is defined by:

$p \bullet p_0 = p,$

$p \bullet X = \{p \bullet x \mid x \in X\},$

$p \bullet <a,q> = <a, p \bullet q>;$

- ***Union*** "U" is defined by:

$p \cup p_0 = p_0 \cup p = p,$

for $p,q <> p_0$, $p \cup q$

is the set theoretic union of the sets p and q;

- ***Merge*** "||" is defined by:

$p \| p_0 = p_0 \| p = p,$

$X \| Y = (X \|_L Y) \cup (X \|_R Y),$

$X \|_L Y = \{x \| Y \mid x \in X\},$

$X \|_R Y = \{X \| y \mid y \in Y\},$

$<a,p> \,||\, X = <a, p||Y>$, and

$X \,||\, <b,q> = <b, X \,||\, q>$.

Processes with additional structures (synchronization, function and communication) and corresponding operations were defined individually in [CE81]. In the following sections, a CSP-Like language L is defined in process concepts.

The syntax of L is:

S ::= x:=s | *skip* | b | S1;S2 | S1 U S2 | S1 || S2 | S* | c?x | c!s |S\c | b=>S

where x is a variable, b is an expression and c is channel name.

The semantics of L is defined in terms of λ notations - the approach originated from Oxford University (OFU) [Sto77], where V is the set of *values* (meanings of variables and expressions), $\Sigma = \mathrm{Var} \rightarrow V$ is the set of *states*, $\alpha \in V$, $\sigma \in \Sigma$, $\sigma\{\alpha/x\}$ is a state which differs from α only at x, V,W are functions which for each s, b and σ determine values $V(s)(\sigma) \in V$ and $W(b)(\sigma)$ in {true, false}:

1. $M(x:=s) = \lambda\sigma.\{ <\sigma\{ V(s)(\sigma)/x\}, p_0> \}$

   $M(\mathit{skip}) = \lambda\sigma.\{ <\sigma, p_0> \}$

   $M(b) = \lambda\,\sigma.$ ***if*** $W(b)(\sigma)$ ***then*** $\{<\sigma, p_0>\}$ ***else*** $\emptyset$

2. $M(S1;S2) = M(S2) \bullet M(S1)$

   $M(S1 \cup S2) = M(S1) \cup M(S2)$

   $M(S1 \,||\, S2) = M(S1) \,||\, M(S2)$

   $M(S^*) = \lim_i p_i,$

with p_0 as always, and $p_{i+1} = (p_i \bullet M(S)) \cup \lambda\sigma.\{<\sigma, p_0>\}$

3. $M(c?x) = \lambda\sigma.\{<\gamma, \lambda\alpha.<\sigma\{\alpha/x\}, p_0> >\}$

   $M(c!s) = \lambda\sigma.\{<\bar{\gamma}, V(s)(\sigma),\sigma, p_0>\}$

4. $M(S\backslash c) = M(S)\backslash\gamma$

   $M(b=>S) = \lambda\,\sigma.$ ***if*** $W(b)(\sigma)$ ***then*** $M(S)(\sigma)$ ***else*** $\emptyset$

The meaning of a assignment statement is defined by the process $<\sigma\{\ V(s)(\sigma)/x\}$, $p_0>$ which differs from the original process only at x. The meaning of an input statement c?x is defined by waiting for the value of α under port name γ. The meaning of a restriction statement S\c is defined by the process after removing the communication pairs $<\gamma,...>$ and $<\bar{\gamma},...>$.

3.3.2.4. The Process Based Semantics of Ada Tasking

In the following section, the process based semantics of Ada tasking is given in terms of the CSP-Like language in the last section.

The syntax of a fragment of Ada tasking L_A is defined as:

- Programs $S \in L_A$ are:

$S ::= T_1 \parallel T_2 \parallel ... \parallel T_m;$

- Tasks T are:

$T ::= x{:=}s \mid$ *skip* $\mid$ *if* b *then* T_1 *else* $T_2 \mid$

while b *do* T | e(s,z) | T_1; T_2 |

accept e(x,y) *do* T |

select $b_1 \rightarrow$ *accept* $e_1\ (x_1,y_1)$ *do* T_1'; T_1''

■ ... ■

$b_n \rightarrow$ ***accept*** $e_n(x_n,y_n)$ ***do*** T_1'; T_1''

In the definition, the parameters of entry call statement are only restricted to two with the first parameter having call-by-value mode and the second having call-by-value-result mode. The numbers of tasks and entry names are also fixed.

The semantics is defined as following, where the Ada statements are overmarked by ′ for distinction.

1. $(x{:=}s)' \equiv (x{:=}s)$

   skip′ ≡ *skip*

$(T_1; T_2)' \equiv (T_1; T_2)$

2. $(\textit{if}\ b\ \textit{then}\ T_1\ \textit{else}\ T_2)' \equiv (b; T_1') \cup (\neg\ b; T_2')$

   $(\textit{while}\ b\ \textit{do}\ T)' \equiv (b;T')^*; \neg\ b$

3. $e(s,z)' \equiv e!s; e!z; e?z$

   $(\textit{accept}\ e(x,y)\ \textit{do}\ T)' \equiv e?x; e?y; T'; e!y$

   $(\textit{select}\ \ldots)' \equiv \bigcup_{i=1}^{n} (b_i => e_i ? x_i; e_i ? y_i; (T_i')'; e_i ! y_i ; (T_i'')') \cup (\neg\ b_1 \wedge \ldots \wedge \neg\ b_n); \Delta$

4. $S' \equiv (T_1' \parallel \ldots \parallel T_m') \setminus \{e_1, \ldots, e_s\}$

The meaning of entry call e(s,z)′ is simulated by three communication statements in L which are fairly easy to understand. The meanings of ***accept*** statement and ***select*** statement are also straightforward. The meaning of the concurrent statement: $T_1' \parallel \ldots \parallel T_m'$ is defined by simultaneous removing matched communicating pairs.

In [BZ83b], a fragment of Ada [Ada81] tasking was modeled by a CSP-Like language. The semantics was denotational and based on the process concept. Various operations and properties for processes were defined. The treatment was very theoretical; however, no proof rules and verification issues were discussed. It might not be useful for implementation purpose.

3.3.3. The Axiomatic Approach to Ada Tasking

In Gerth [Ger82], a sound and relatively complete axiomatization of the Ada [Ada81] rendezvous was simulated by CSP.

3.3.3.1. CSP and Its Proof System

CSP was developed by Hoare [Hoa78]. It contains the traditional sequential programming constructs such as assignment, branching and looping and with following concurrent constructs:

(1) a *concurrent* statement of the form: $[P_1 \| ... \| P_n]$ expressing the concurrent execution of processes $P_1, ..., P_n$ where each P_i refers to a statement S_i, (as denoted by $P_i :: S_i$) and each pair P_i and P_j ($i \neq j$) have no shared variables;

(2) *input and output* commands of the forms: P_j ? x (in S_i) and

P_i ! y (in S_j). P_j ? x expresses a request to P_j to assign a value to the (local) variable x of P_i and P_i ! y expresses a request to P_i to receive a value from P_j. The only communication and synchronization mechanism between processes is via a matched pair input and output commands; and

(3) *guarded* commands which have the form B $\rightarrow$ S where B can be a Boolean expression, an input and output command, or any combination of both. A Boolean guard is *passable* if it is true; an I/O command is *passable* when a corresponding I/O command in the process addressed is ready. Both guarded *selection*:

[B1 $\rightarrow$ S1 ■ B2 $\rightarrow$ S2] and guarded *iteration*:

*[B1 $\rightarrow$ S1 ■ B2 $\rightarrow$ S2] can be built.

The proof system of CSP was developed in Astesiano et al [Ast86]. The idea is to use Hoare's Logic for the sequential part of the language and use two axioms for local reasoning about processes in isolation: {p} P_i ! a {q} and {p} P_j ? x {q}, then apply a *cooperation test* to verify whether these pre- and post-assertions are compatible with the communication. To express this test, a *general invariant*, GI, is used to group the local reasonings for each process globally together. GI is especially used to distinguish among all communication possibilities, the syntactically (statically) matching ones and the semantically (dynamically) matching ones. Auxiliary variables are introduced to express the necessary assertions and invariants.

3.3.3.2. The CSP Definition and Proof System of Ada Tasking

In Gerth [Ger82], a subset called Ada-CS was defined by the following BNF-like grammar:

program ::= ***begin*** task {task} ***end***

task ::= ***task*** task_id decl ***begin*** stats ***end***

decl ::= {entry_decl} {var_decl}

entry_decl ::= ***entry*** entry_id {formal_part}

var_decl ::= var_id_list : ***int*** | var_id_list : ***bool***

var_id_list ::= var_id {,var_id}

formal_part ::= [var_id_list] [# var_id_list]

stats ::= stat {;stat}

stat ::= ***null*** | ass_st | if_st | while_st | call_st | acc_st | sel_st

ass_st ::= var_id := expr

if_st ::= ***if*** bool_expr ***then*** stats ***else*** stats

while_st ::= ***while*** bool_expr ***do*** stats

call_st ::= ***call*** task_id.entry_id (actual_part)

actual_part ::= {expr} [# var_id_list]

acc_st ::= ***accept*** entry_id (formal_part) ***do*** stats

sel_st ::= ***select*** sel_br {v sel_br}

expr ::= "expression"

bool_expr ::= "boolean expression"

id ::= "identifier"

The subset Ada-CS mainly contains task related constructs: entry declaration, entry call statement, accept statement and select statement. The actual parameters of entry call are of two modes: ***in*** before the separator # and ***in out*** after #.

The semantics of Ada-CS is defined by axioms and inference rules in the form of Hoare Logic. A rendezvous is simulated by two CSP communication commands on the right:

Ada-CS:	CSP
T_1 : ***call*** T_2 entry(e#x)	T_2!(e,x); T_2?x
T_2 : ***accept*** entry(u#v) ***do*** S	T_1?(u,v); S; T_1!v

The following axioms (denoted by A) and proof rules (denoted by R) are adopted:

A1. entry call:

{p} ***call*** T.entry($\vec{e}$ # $\vec{x}$) {q}

R1. accept:

$$\frac{\{p_1\}\ S\ \{q_1\}}{\{p\}\ \textbf{\textit{accept}}\ \text{entry}(\vec{u}\#\ \vec{v})\ \textbf{\textit{do}}\ S\ \{q\}}$$

R2. select:

$$\frac{\{p \wedge b_1\}\ S_1\ \{q\},\ldots,\{p \wedge b_n\}\ S_n\ \{q\}}{\{p\}\ \textbf{\textit{select}}\ b_1\text{: } S_1 \text{ v} \ldots \text{ v } b_n\text{: } S_n\ \{q\}}$$

The usual assignment-axiom, the null-axiom and the if-, while-, composition-, consequence- and conjunction-rules are defined as in CSP. The communication rule is shown below:

R3. formation:

$$\frac{\begin{array}{l}\{p_1 \wedge p_2 \wedge GI\}\ S_1';\ S_1''[.]\ \{\bar{p}_1 \wedge \bar{p}_2\} \\ \{\bar{p}_2\}\ S\ \{\bar{q}_2\} \\ \{\bar{p}_1 \wedge \bar{q}_2[.] \wedge GI\}\ S_2'[.];\ S_2''\ \{q_1 \wedge q_2 \wedge GI\}\end{array}}{\{\,p_1{\wedge}p_2{\wedge}GI\}\ S_1';\ \textbf{\textit{call}}\ T_j\,.a(\vec{e}\ \#\ \vec{x});\ S_1''\ ||\ \textbf{\textit{accept}}\ a(\vec{u}\ \#\ \vec{v})\ \textbf{\textit{do}}\ S_2';\ {>}S{<};\ S_2''\ \{q_1{\wedge}q_2{\wedge}GI\}}$$

where the call is contained in task T_i and the accept statement in task T_j; [.] is the abbreviation of parameter substitution $[\vec{e}/\vec{u},\ \vec{x}/\vec{v}]$; $\{p_1\}\ T_i\ \{q_1\}$ and $\{p_2\}\ T_j\ \{q_2\}$. The assertion $\bar{p}_1$ specifies the condition satisfied by the variables in T_i not appearing in the actual result-parameter list; $\bar{p}_2$ [.] and $\bar{q}_2$[.] specify the precondition and postcondition that S must satisfy, $>$S$<$ represents the fact that S does not contain any other entry call statement or accept statement. The interpretation of the rule is if the three premises are true, then the conclusion (under the line) holds for all matching communication pairs: ***call*** in T_i and ***accept*** in T_j

In [Ger82], a very small subset of Ada [Ada81] was defined by axiomatic approach. The following restrictions were imposed: there was a fixed number of tasks, no shared variables, no entry-queues, no delay statement, deadlock simulating abortion. The definition is not useful for implementation, but it is very useful for verification purpose. The axiomatization was proved to be sound and relatively complete. Only partial correctness was addressed.

3.3.4. The Algebraic Approach to Ada Tasking

Unfortunately, there is no algebraic definition of Ada tasking occurred in literature yet. The initial algebra approach [GTWW79] has been extended to define the semantics of sequential programming language constructs ([BWP87], [Mos83]), its application to concurrency has also been explored in Zamfir [Zam87]. There are other efforts to treat concurrent processes as algebras [BK84] based on the work of Milner's CCS [Mil80] which largely depends on the special meanings of predefined operators upon processes. A translation from a subset of Ada to CCS was presented in Hennessy and Li [HLi83], however this definition did not have the flavor of the algebraic approach to be adopted in our model and thus was not illustrated here.

3.3.5. Petri Net Modeling of Ada Tasking

In Mandrioli et al [MZGT85], Ada [Ada83] tasking was modeled by Petri nets. The rendezvous between two tasks T_1 and T_2 can simply be represented by Figure 3.1. and be interpreted by the firing rules of Petri nets: i.e. T_1 and T_2 must reach the point *start rendezvous* together to make the transition t1 to fire.

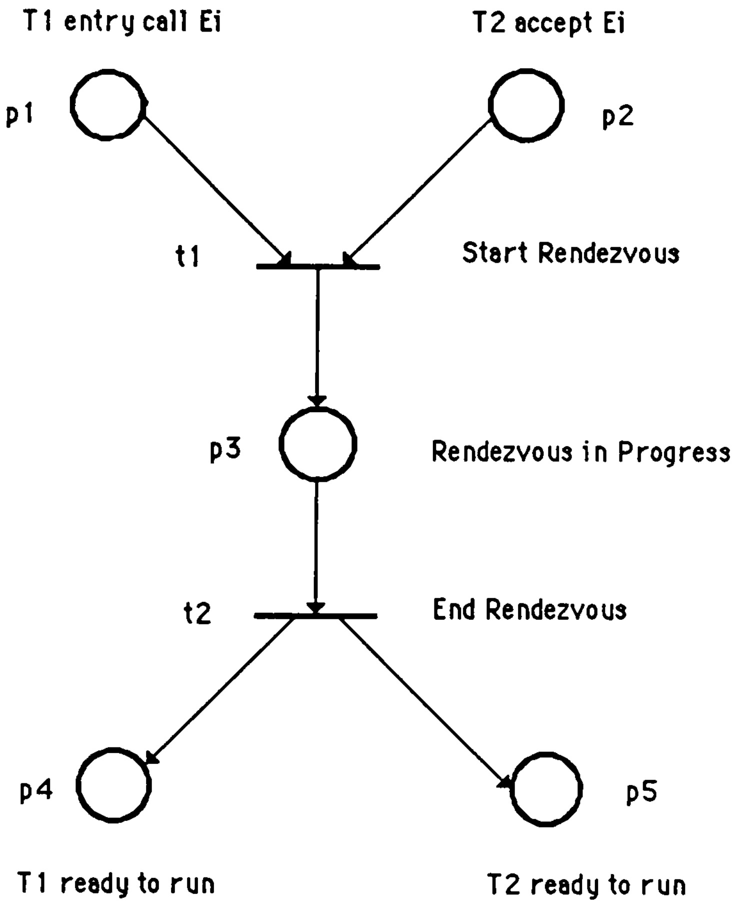


Figure 3.1 - A Petri Net Specification of Rendezvous

This definition covered almost all aspects of Ada tasking: task activation, task dependency, delay statement, rendezvous, exception, selective wait, and timed entry call. It is the only formal definition so far dealing with real time issues of Ada tasking. The definition was graphical and very easy to understand. However the definition is not systematic nor structured and it is not appropriate for implementation use.

3.3.6. Temporal Logic Definition of Ada Tasking

In Pnueli and DeRoever [PD82], a small fragment of Ada [Ada81] relating to rendezvous was studied under the assumption: no shared variables, no new dynamically created tasks, no procedures and subprograms, no nested blocks, and no delay statement. The definition was given in operational semantics and was based on the preliminary version of Ada. The proof rules for the definition were given in linear temporal logic and only for *just* and *fair* computation sequences. The use of temporal logic in proving correctness was demonstrated by examples. The definition explicitly dealt with entry queues and was at a low level.

In order to give some flavor of temporal logic definition, a simplified example is provided by this author. Traditionally, a state in temporal logic contains two kinds of variables *control* variables which indicate the locations during the execution of programs (usually represented by ***at***, ***after*** etc.) and normal ***program*** variables. Following [PD82], a state is defined as: $s = <(T_1 \textbf{ at } S_1) \wedge \ldots \wedge (T_m \textbf{ at } S_m); \vec{\eta}>$

where $T_1, \ldots, T_m$ are m tasks, S_i is a sequence of statements, $\vec{\eta}$ contains the current values of program variables. Let

$s_k = \ <\ldots(T_i \textbf{ at } \mathrm{e}(\vec{u};\, \vec{v}\,);S_i) \wedge \ldots \wedge (T_j \textbf{ at accept } \mathrm{e}(\vec{f}\textbf{:in};\vec{g}\textbf{:out}\,);\mathrm{B}\textbf{ end } \mathrm{e};S_j) \wedge \ldots;\vec{\eta}>$,

$s_l = \ <\ldots(T_i \textbf{ at rendezvous } \mathrm{e};\, S_i\,) \wedge \ldots \wedge (T_j \textbf{ at } \vec{f}\!:=\, \vec{u}\,;\, \mathrm{B};\, \vec{v}\!:=\!\vec{g};\textbf{ end } \mathrm{e};\, S_j\,) \wedge \ldots;\, \vec{\eta}>$,

$s_m = \ <\ldots(T_i \textbf{ at rendezvous } \mathrm{e};\, S_i\,) \wedge \ldots \wedge (T_j \textbf{ at end } \mathrm{e};\, S_j) \wedge \ldots;\vec{\eta}>$,

$s_n = \ <\ldots(T_i \textbf{ at } S_i) \wedge \ldots \wedge (T_j \textbf{ at } S_j) \wedge \ldots;\, \vec{\eta}\ >$

then the simplified rendezvous transition between task T_i and T_j is: $s_k \rightarrow s_l$

and the rendezvous termination transition is: $s_m \rightarrow s_n$

Let ϕ_1, ϕ_2, ψ_1, and ψ_2 be temporal formulas describing state properties: s_k , s_m, s_l and s_n respectively, then the correctness of rendezvous between T_i and T_j can be expressed as:

(1) $\phi_1(s_k) \rightarrow \Diamond\ \psi_1(s_l)$

(2) $\Box\ (\ \phi_2(s_m) \rightarrow \psi_2(s_n)\ )$

where the first formula means the entry call in T_i will eventually be accepted by an accept statement in T_j and the second formula expresses that the rendezvous between T_i and T_j will always terminate correctly.

3.4. A Comparison of Formal Methods

Based on our own experience, the extensive literature study and the illustration of various formalisms in the previous sections, we evaluate the formal methods against the requirements provided in Chapter 2. Relative values of 1, 2 and 3 have been assigned to each method in terms of the degree of conformance to each requirement, with 1 implying poor, and 3 for very good. The results of the comparison are summarized in Table 3.1 to Table 3.3.

3.4.1. The Interpretation of Table 3.1

All of the above specification methods except the operational approach have well-founded mathematical bases; actually some of them (Petri nets and temporal logic) are themselves rigorous meta theories. The operational approach is based on the concept of a very simple abstract machine.

While all of the above specification methods are precise, they allow unique interpretations of specifications to be written in any one of them.

Table 3.1 - Foundation

	Petri Nets	Temporal Logic	Algebraic Approach	Denotational Approach	Axiomatic Approach	Operational Approach
Formal	3	3	3	3	3	1
Precise	3	3	3	3	3	3
Expressive	3	2	2	2	1	3

Table 3.2 - Applicability

	Petri Nets	Temporal Logic	Algebraic Approach	Denotational Approach	Axiomatic Approach	Operational Approach
Abstract	3	1	2	2	3	1
Modular/ Compositional	2	1	2	3	1	3
Sequential	3	3	3	3	3	3
Parallel	3	3	2	2	2	3
Executable	3	2	1	2	1	3
Verifiable	2	3	3	2	3	1

Table 3.3 - Comprehensibility

	Petri Nets	Temporal Logic	Algebraic Approach	Denotational Approach	Axiomatic Approach	Operational Approach
Understandable	3	2	1	1	3	2
Learnable	3	2	1	1	2	3
Usable	3	2	1	1	2	3

Petri nets and the operational approach are powerful enough to express almost all computable functions (computable by Turing machine). The axiomatic approach being based on propositional logic is the least powerful among the above specification methods.

3.4.2. The Interpretation of Table 3.2

Petri nets and the axiomatic approach can be used to give very high level specifications without using some sort of internal state components, while the operational approach and temporal logic traditionally used with the operational approach are explicitly involving internal state components.

The compositionality is one of the principles of the denotational and operational approaches, while specifications in either temporal logic or the axiomatic approach is very difficult to be composed.

All of the above specification techniques can handle sequentiality easily.

Petri nets and temporal logic are intrinsically suited to describe parallelism. The operational approach is also capable to deal with parallelism. Other specification methods are not very good at specifying and verifying parallel systems.

Specifications in Petri nets and the operational approach are directly executable, while specifications in the algebraic and axiomatic approaches are not executable.

Specifications in any one of the temporal logic, the algebraic, and the axiomatic approaches can be verified within the same framework, while specifications in the operational approach are very difficult to verify.

3.4.3. The Interpretation of Table 3.3

The principles of Petri nets and the axiomatic approach are easy to understand. The algebraic and denotational approaches are difficult to understand, especially when strange notations are used.

Petri nets with their simple graphical representations are relatively easy to learn and use, so is the operational approach due to its similarity to procedural languages. The algebraic and denotational approaches with their profound mathematical foundations are generally difficult to learn and use.

Chapter 4

A Model for Integrating Formalisms in Software Development

In the previous Chapters, various formal methods were introduced, their applications were reviewed and their merits were compared. In this Chapter, several large projects based on formal methods for software development are overviewed, then a model for integrating various formal methods in software development is created with the following features:

(1) Specifications are viewed both as a set of products and a set of well-defined steps of a process,

(2) Specifications (as a set of products) at different development steps are to be written and verified by different formalisms,

(3) Specification (as a process) spans from the requirement phase to the detailed design phase,

(4) Specification for both concurrent and sequential software is supported, and

(5) Specifications for different logical aspects (concurrent control abstraction, data abstraction and procedural abstraction) of a piece of software are dealt with separately.

The model intends to combine the strengthes of various formal methods and to adopt the successful experience of various projects to software development. The model and the associated methodology constitute a *specification transition paradigm.*

4.1. Basic Elements in Software Development

4.1.1. Software Life Cycle

The Software life cycle is the description of the *process* of software development starting from user requirements through design and implementation, to delivery and maintenance. There are several software life cycle models ([Bal85], [Zav82]), but the most commonly used is the *waterfall* model [Boe76] which captures the main phases during a software development. The waterfall model is presented in Figure 4.1, though many variants can be found.

4.1.2. Software Quality

Software quality is one of the measures applicable to the *product* of software development. The criteria by which to measure software product quality [Ram86] can be divided into three categories:

(1) functionality -- *correctness, reliability, robustness,*

(2) performance -- *efficiency, cost-effectiveness, constraint satisfaction, reusability,* and

(3) comprehensibility -- *maintainability, understandability* and *usability.*

These quality criteria are not necessarilly maintained in any intermediate product, but ideally they have to be assured by the intermediate phases of software development process.

Functional properties can be best described, analyzed and guaranteed by formal methods as proven by many studies. Performance properties such as real-time constraint, though are hard to be formalized, can also be effectively expressed and simulated by timed Petri nets and temporal logic. Comprehensibility properties are very difficult to measure since different software partcipants may have different perceptions of software quality as pointed out in the next section; however, formalisms such as Petri nets are general enough to be easily understood by all partcipants.

4.1.3. Software Development Participants

Software development participants can be divided into different groups according to the phases of software life cycle: *customers* (those providing requirements), *requirements engineers* (those formalizing the requirements), *system designers* (those designing the

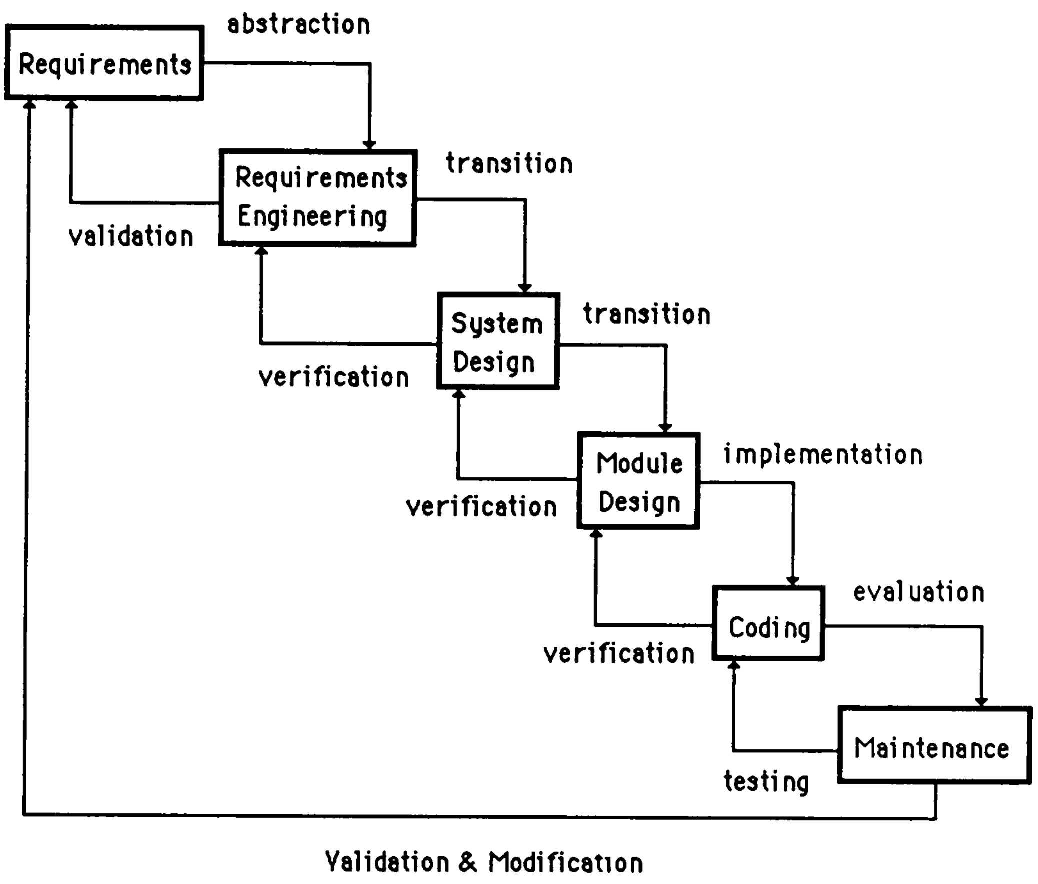


Figure 4.1 - The Waterfall Model of Software Life Cycle

system structure and its interfaces), *module designers* (responsible for designing individual modules), *programmers* (people implementing algorithms and data structures), *maintainers* (tasking with installing and maintaining the software product), and *end users* (those using the software product, who coincidently also may be customers). Persons belonging to different groups may have different perceptions of the software process and software product. Usually, those involved in phase n are responsible for the quality of intermediate product for entry into phase n+1, and are concerned about the quality of intermediate product from phase n−1.

4.2. Large Software Development Projects Based on Formalisms

There are many studies and applications of formal methods as reviewed in previous Chapters, however most of them concern either formalisms themselves (theoretical studies) or applications of formalisms to isolated specific problems (case studies). This is one of the major reasons that while research on formalisms is flourishing, the real progress towards rigorous software development is very slow. To narrow this gap between theoretical results and practical software development, a systematic software development methodology employing formalisms is needed, which should take various elements in software development into account, i.e.

(1) what is a general software development model to be supported by the methodology?

(2) where is a formal method applicable in the model?

(3) who is the user of the formal method? and

(4) how can software qualities be assured?

There are several large software development projects working towards the construction of such a systematic methodology, they are:

- the Automatic Programming Project [Bal85],
- the CIP Project ([CIP85], [CIP87]),

- the Larch Project [GHW85], and
- the RAISE Project ([BDMP85], [MG87]).

These projects are briefly introduced in the following sections.

4.2.1. The Automatic Programming Project

The Automatic Programming Project has been undertaken at Information Sciences Institute of University of Southern California for more than 15 years [Bal85].

With the aim to overcome the fundamental flaws in traditional waterfall model:

(1) lacking technology for managing the knowledge-intensive activities during the software development process and

(2) performing maintenance on low level source code,

an extended automatic programming paradigm Figure 4.2 has been proposed.

The extended automatic programming paradigm was based on operational approach and consisted of three major software development phases:

(1) specification acquisition:

A formal (operational) specification language called Gist was used in this phase to construct formal high-level specifications from informal specifications. The obtained formal specifications were then validated by static paraphrase (paraphrasing them into corresponding natural language specifications to enhance their understandability) and dynamic symbolic execution (executing them to detect misbehaviors). Maintenance was performed on high-level specifications so that the second flaw of waterfall model could be resolved;

(2) interactive translation:

In this phase, high-level (operational) specifications were transformed into low-level (operational) specifications by the interactive process called formal development which generated the history of transformational derivation. A language called Paddle was

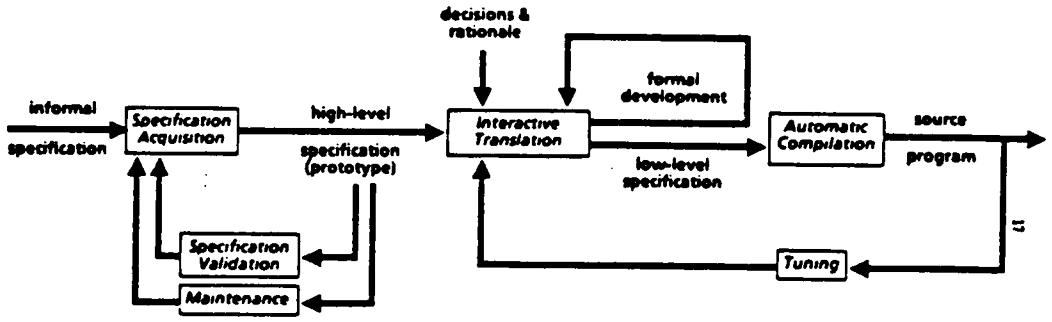


Figure 4.2 - Extended Automatic Programming Paradigm

used to operationally specify the structure of transformation sequences for a system implementation. The execution of such a transformation sequence upon a high-level specification would produce a corresponding (low-level specification) implementation of the high-level specification. A replay mechanism was introduced to repeatedly modify the formal development to generate the next version of low-level specification so that the first flaw in the waterfall model could be eliminated;

(3) automatic compilation:

Low-level specifications were to be automatically translated into implementations during this phase. Some experimental work focused on narrowed application area was being explored.

Although this project is still in its exprerimental stage with many problems to be solved, such as incremental specification, replay and automatic compilation, its effort in formalizing and automating the software process and explicating the development paradigm has largely influenced our own model.

4.2.2. The CIP Project

The CIP (Computer-aided, Intuition-guided Programming) project has lasted for 15 years at the Technical University of Munich and produced fruitful results as summarized in [CIP85] and [CIP87].

The CIP project consisted of two major components for supporting the inferential programming style (the process of constructing, modifying and reasoning programs and their derivations):

(1) A wide spectrum language CIP-L:

CIP-L is a wide spectrum language which supports incremental compilation, localizing correctness consideration and facilitating transformation process based on a common semantics. CIP-L is also an abstract scheme language which supports a complete separation of control structures (by uninterpreted schemes) and data structures (by algebraic types).

CIP-L has a *transformational semantics* in which a kernel contains the essential semantic concepts and the semantics of all other language constructs are provided by mapping them into this kernel by formal transformation rules;

(2) A system CIP-S for program transformation:

CIP-S was designed to support transformational program development. It is responsible for the manipulation of concrete programs, the derivation of new transformation rules within the system, the transformation of algebraic types (schemes), and the verification of applicability conditions. CIP-S also supports interactive input / output, data base facilities for programs, rules, types, and transformation programs, and an advanced documentation components.

The CIP project focused mainly on the design phase of a software life cycle, however its transformational development methodology has significant impact on our specification transformation model.

4.2.3. The Larch Project

The Larch project [GHW85] at MIT is the continuation of research on applications of formal specification started by Guttag [Gut75] and aims at developing both a family of specification languages and a set of tools to support their use, including language-sensitive editors and semantic checkers based on a powerful theorem prover.

The Larch family of specification languages consists of:

(1) the Larch interface languages:

The Larch interface languages are particular to specific programming languages, they are used to specify program modules and to provide information needed to write programs that use these modules, and

(2) the Larch Shared Language:

The Larch Shared Language is independent of any specific programming language and is mainly algebraic, which is used to write equations for defining the relations among operators and giving meaning to equality among terms appearing in interface specifications.

Larch specifications are two-tiered as shown in Figure 4.3. Each Larch specification has a component written in an algebraic language (the Larch Shared Language) and another tailored to a programming language (a Larch interface language).

Although the Larch project focuses on the local specifications for sequential programs, its methodology may well form the basis for the local (module) specification phase of our own model.

4.2.4. The RAISE Project

The RAISE (Rigorous Approach to Industrial Software Engineering) project (started in 1985 and to be finished by 1989) is being carried out by a consortium headed by Dansk Datamatik Center ([BDMP85], [MG87]). It is a part of the ESPRIT project funded by the Commission of the European Communities and has a size of 115 person-years with the aim to construct a mathematics based method for the development of industrial software.

The RAISE project consists of the following components:

(1) the RAISE method:

The RAISE method is developed to enable the construction of reliable software by formalizing the software development process. The rigorous approach as defined in Bjorner [Bjo87] is adopted in the RAISE method. A formal document called *project graph* which describes the overall structure of a software development project is

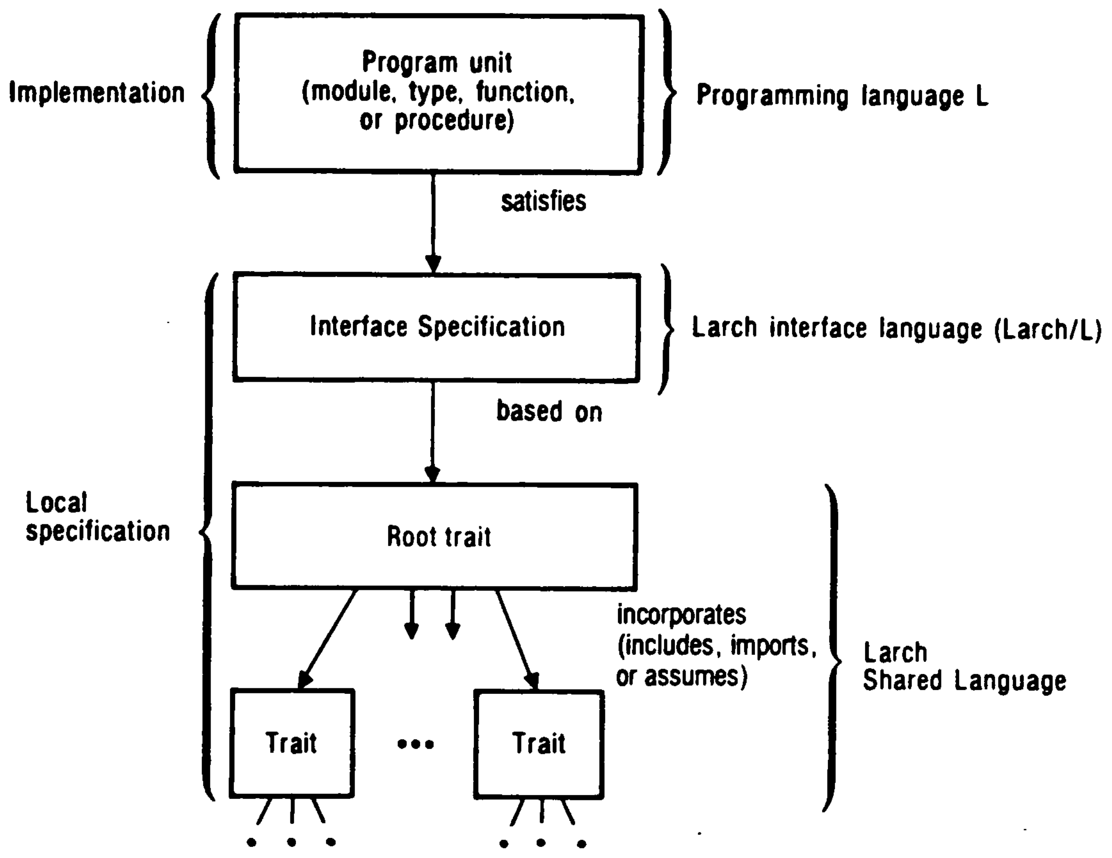


Figure 4.3 - Two-Tiered Specification in Larch

produced to support stepwise development by a process of commitment in which the degrees of freedom or indeterminacy are removed Figure 4.4,

(2) the RAISE Specification Language (RSL):

The RAISE specification language is built on a kernel language in the model-oriented, applicative style of VDM (Vienna Development Method) [BJ82] and is used to express the (specification, design and implementation) phases of software development. The RAISE specification language provides ***structures*** as building blocks and abstraction units, in which the RSL entities such as types, values, variables, operations and processes are defined. Specifications in RAISE are basically model-oriented [LB79] with certain property-oriented (axiomatic or algebraic) concepts,

(3) the mathematical foundation of RAISE:

The semantics of RAISE specification language is mainly defined by denotational semantics [Sto77] so that the same mathematical foundation is employed in both the formal development process and the underlying method,

(4) the RAISE engine:

The RAISE engine will consist of a set of computer-based tools to support the RAISE method and language.

The RAISE project is based on the successful experience of VDM ([VDM87], [VDM88]) and has a large man power investment, therefore its experience will in turn be borrowed in future development of our own model, especially in the module specification phase.

4.3. The Specification Transition Paradigm for Software Development

With the aim to make software development more rigorous and productive, based on our strong interests, beliefs and experience on formalisms, and inspired by the above large formal software development projects, we are developing a *specification transition*

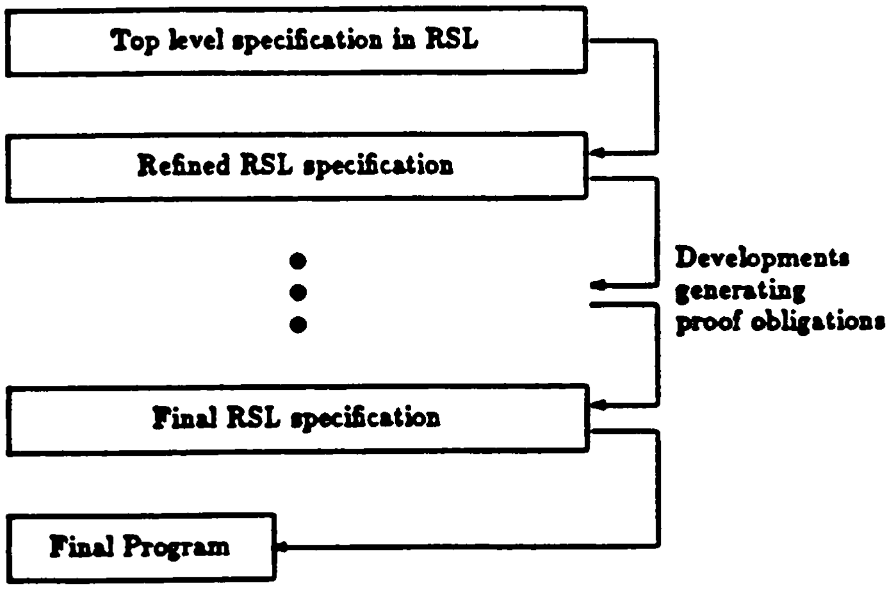

Top level specification in RSL
Refined RSL specification
Developments generating proof obligations
Final RSL specification
Final Program

Figure 4.4 - Stepwise Development in RSL

model and the associated methodology for supporting such a model. Preliminary results of this research are presented in this section.

4.3.1. The Goals and The Strategy

The goals to be achieved in this research are listed below:

(1) Build a software development model containing most advanced formal methods;

(2) Develop a methodology for such a model; and

(3) Design a spectrum specification language consisting of various formalisms.

It is no surprise that our goals are very similar to those in the previous mentioned projects since all aim to introduce formalisms into software development process, to make the process rigorous and to improve the reliability of final software products. However the differences between our model and other projects are:

(1) In our model, specifications are viewed both as a set of products and a set of well-defined steps of a process, furthermore specification (as a process) spans from the requirement phase to the detailed design phase. Therefore our model is *specification-oriented* and *specification-driven.* Although the Automatic Programming project has the same views of specifications, its operational approach based methodology seems inappropriate for high-level specifications and insufficient for error discovery in early software development stages;

(2) In our model, specifications (as a set of products) at different development steps are to be written and verified by different formalisms , and specifications for different logical aspects (concurrent control abstraction, data abstraction and procedural abstraction) of a piece of software are dealt with separately. Our model integrates foremost formalisms (algebraic, axiomatic, denotational and operational approaches, Petri nets and temporal logic) than any other projects, and constitutes a *specification transition paradigm*: *transformations* within a single formalism and *translation* between different

formalisms. Therefore much more difficult and challenging problems are to be solved in our model;

(3) In our model, specifications for both concurrent and sequential software are supported with the emphasis on the formal methods for the specification and verification of concurrent software structures and properties.

The strategy for achieving the above goals are correspondingly divided into three stages:

- Stage 1 - investigate the feasibility and applicability of various formalisms in software development life cycle and study the interfaces between them.

There are numerous applications of a single formalism in software development and most of them almost always focus on a particular phase in the life cycle, usually at a low level. Based on my review of various formalisms in Chapter 3, it is possible to cover the most part of the life cycle by combining various formalisms. The first thing needs to do is to build a taxonomy of the most advanced applications of various formalisms in software development, then to select a subset of most promising formalisms to be employed in our model.

Combining various formalisms together is the hardest part of the model, both theoretical question such as consistency and practical question such as the granularity (abstract level) of the application of each formalism need to be validated. Actually, there are two kinds of interfaces: *vertical interfaces* which determine when a specification in one kind of formalism should be replaced by a more refined specification in another kind of formalism and *horizontal interfaces* which divide a specification in terms of various formalisms employed at the same time. The strategy to be used here is to separate the interfacing problem into subproblems and to attack each interface one at a time.

To summarize, in order to achieve the first goal, three problems need to be solved:

(a) whether it is feasible or not to apply various formalisms in software life cycle;

(b) where in the life cycle should one kind of formalism be applied; and

(c) how can the different formalisms be interfaced.

To solve (a), a review of the applications of various formalisms in the software development was provided in Chapter 3 and Chapter 4.

To solve (b), a model for indicating where to apply various formalisms in software development life cycle is built in Figure 4.5 in a following section.

To solve (c), the interfaces between various formalisms are attacked separately one at a time in a top-down approach; i.e., starting with the study of interface between Petri nets and temporal logic (Chapter 6). Such combined formalisms are then applied to some simple examples in an incremental way. The preliminary model in Figure 4.5 is to be refined and modified as long as necessarily and appropriately.

The originality of this model will shed some lights on the application of formalisms in software development.

- Stage 2 - Search for transition techniques and combine specification with verification.

Once the interfaces between various formalisms in the model have been cleared up, the next thing needs to do is to find various transition techniques which can also be classified into two categories: the first category includes transformation techniques within the same kind of formalism and the second category which is more difficult contains translation techniques between different formalisms. Some methods and heuristics needs to be developed to facilitate automatic and mechanical processing.

In order to guarantee the correctness of the final product, the correctness of intermediate results must be maintained. Parallel to the development of transition techniques, verification techniques need to be developed. For different formalisms, different verification methods are going to be applied and preferablly the same framework of the formalism is going to be used.

To summarize, in order to achieve the second goal; at least the following problems need to be solved:

(a) how to refine a specification in a single framework of one kind of formalism;

(b) how to translate a specification in one kind of formalism into a specification in another kind of formalism; and

(c) how to verify the correctness of such translations.

To solve (a), existing techniques and principles for the formalisms are to be adapted and further developed. Chapter 5 contains a new design methodology for high-level Petri nets.

To solve (b) and (c), some methods and techniques for facilitating automatic and mechanical processing are to be developed. Their usages are to be demonstrated by applying them to simple problems. As a result of the research in these steps, a translation algorithm from predicate transition Petri nets to temporal logic has been designed in Chapter 6, and a special temporal logic verification technique for predicate transition nets has been developed in Chapter 7. Tools for supporting such techniques will not be developed because of the time limitation. Significant theoretical contributions are expected to be made here.

- Stage 3 - Evaluate the model and the methodology and design a spectrum of specification language.

For this model and the methodology to be useful, they need to be applied to some real problems; the results are going to be compared and evaluated with those obtained from a traditional way. Finally, a spectrum of specification language for supporting such a model is going to be designed under the criteria proposed in the previous requirements section.

To summarize, in order to achieve the third goal; some problems need to be solved:

(a) how well can the model and the methodology assure software quality (here the correctness is the main concein); and

(b) how can this model and the methodology be used.

To solve (a), the model and the methodology will be evaluated by applying them to some real problems and comparing the results with those obtained from a traditional way.

To solve (b), the selections of various formalisms among their variants will be made; and a spectrum specification language (based on the selected formalisms and under the requirements proposed in Chapter 2) will be designed in a syntax loosely coupled (salad) fashion to support the application of the model.

This goal has the pragmatic importance, and is a long term one and thus will not be further treated in the dissertation.

4.3.2. The Formal Software Development Model

The applications of various formalisms in software development are abundant, however none of them becomes dominant. Each formalism with its own advantages and disadvantages is applicable to some specific application domains and problems, and is particularly well suited for playing some role in an element of the software development process. A piece of software can logically be divided into three aspects:

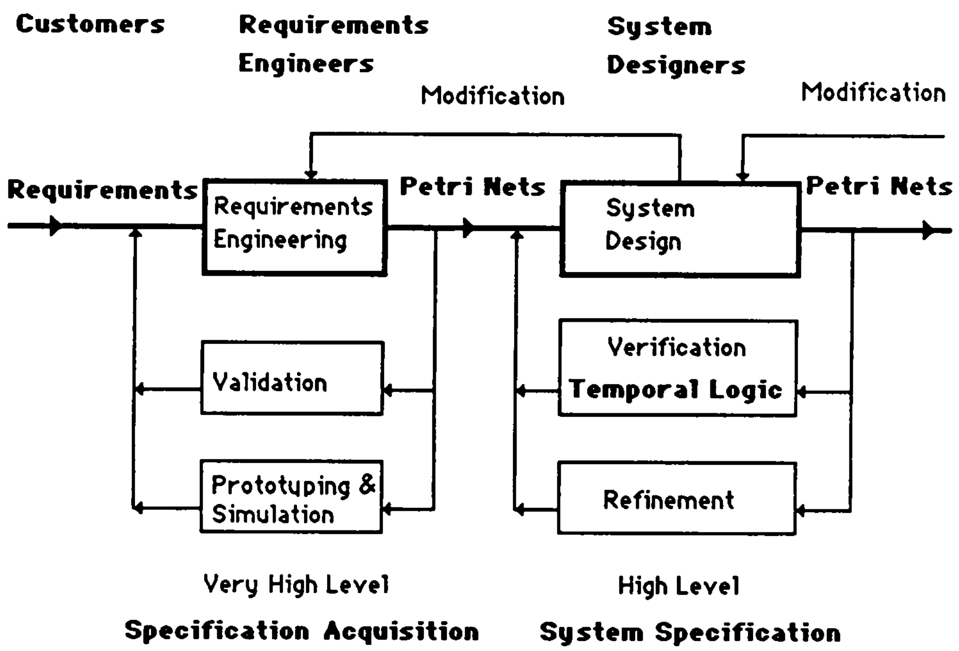
(1) *control abstraction* which specifies the dynamic relationships between subcomponents and can be effectively specified by Petri nets and temporal logic;

(2) *data abstraction* which describes the internal structure of each component and can be naturally described by the algebraic and axiomatic approaches; and

(3) *procedural abstraction* which defines the static functionality of each component and which can be successfully defined by the denotational approach.

While operational approach is capable to deal with all three kinds of abstractions, it is best to be used at a low design level. Based upon the above study, a software development model is presented in Figure 4.5.

It utilizes all the above mentioned formal specification methods and elaborates the waterfall model. At the very high specification level, high level Petri nets can be used to abstract and formalize customers' informal requirements as part of the requirements analysis which results in top-level control flow specifications. Prototyping and simulation can be performed at this stage and the specification can be validated against the user requirements. At the next specification level, the Petri nets can be further refined so that the control structure of the major concurrent and distributed components of the whole system can be constructed. The consistency between different versions of Petri nets can be ensured by the transformation rules of Petri nets, and various properties (liveness and safety) of the specified system can be verified by temporal logic formulas which might be derived partially from the refined Petri nets. At the medium specification level, the modules can be detailed by using the algebraic approach for data abstraction and the denotational approach for functional abstraction. Various transformations and correctness proving can be carried out within the same framework of these formalisms. At the low specification level, the descriptions of modules can be further refined using the operational approach to replace upper level algebraic and denotational definitions, and integrated using operational approach to substitute the Petri nets description of the



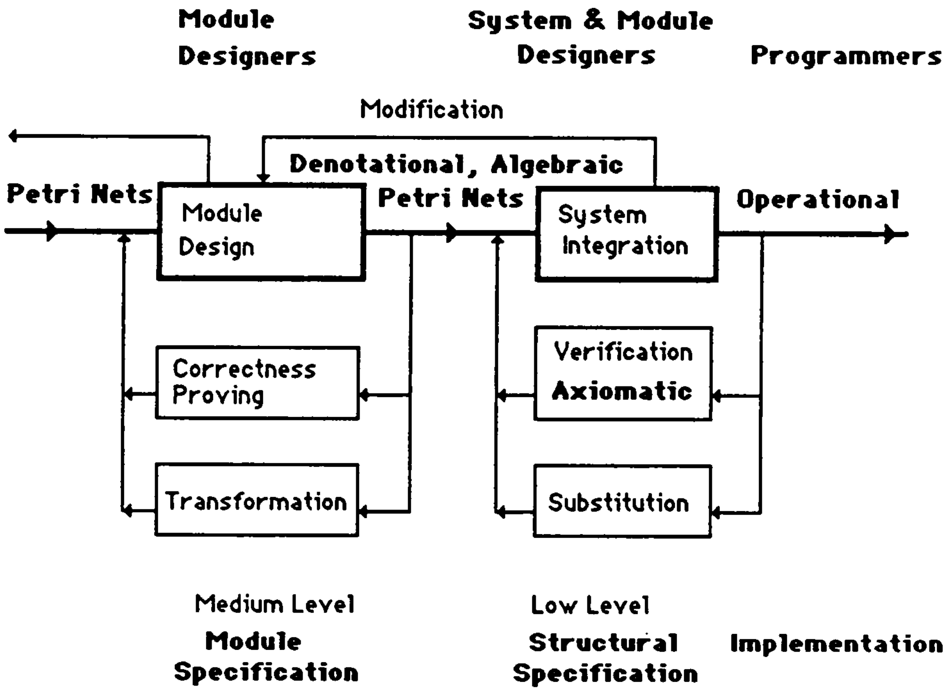


Figure 4.5 - The Specification Transition Paradigm for Software Development

system structure. The axiomatic approach can be used to verify the consistency within each module and the consistency of the interfaces between the modules in the final operational specification of the whole system. At each level verification of the corresponding specification can be completed, provided that the relationships between successive formalisms can be well defined. For example, the uppermost level of control structure must be maintained throughout the development model, and properties developed through (say) algebraic concepts must be reflected in operational specifications. Modification is undertaken only if there is evidence of inconsistency or incompleteness at a lower level as a result of the decision at a immediate higher level.

4.3.3. The Formal Software Development Methodology

The associated methodology contains three major phases in chronological order: *system specification*, *module specification* and *structural specification.*

System specification describes the upper level concurrent and distributed structure of the specified system and the dynamic relationships among the modules. Two kinds of formalisms are employed in this phase: predicate transition Petri nets and first order temporal logic. Petri nets are used to abstract the logic structure of the system and temporal logic is mainly used for verifying the dynamic properties of the system. System specification consists of following steps:

1. Derive an initial predicate relation Petri net specification from informal requirements:

   Initially, such a specification can be very simple and may only have a simple predicate or relation. Techniques such as graphical modification [Mai85] and the distinction between active and passive elements [Rei85a] can be applied;

2. Refine the Petri net specifications structurally according to the users' requirements to the degree so that major concurrent and distributed components can be identified: Principles of abstraction and refinement [Rei85a], and dividing and connecting can be used to transform the initial Petri net specifications into a consistent and more detailed Petri net specifications. A new design methodology for predicate transition Petri nets has been developed in Chapter 5;

3. Mark the Petri net specifications according to users' requirements:
The functionality of each component and relationship between components are defined by constructing corresponding predicate and relation;

4. Validate the specification against the requirements:
A paraphraser similar to that in [Bal85] can be adopted to produce a verbal description of the specification to improve its readability and to uncover those errors causing the specification to be different from users' intent. Simulation and prototyping can be performed by executing the marked Petri nets specification with an intended initial marking [MBT85];

5. Build temporal logic formulas from the Petri net specifications:
The invariant properties of the specified system are traditionally analyzed by linear algebraic techniques [Rei87a] when low level Petri nets are used. It is very difficult either to find place invariants or to solve invariant equations [Jen87] in high level Petri nets. Linear algebraic techniques are also not powerful enough to analyze all desired invariant properties [Bar84]. Several research efforts [DG83] and [QS82] tried to use both Petri nets and temporal logic in specifying and verifying network protocols; however, they provided these two formalisms separately and in very simple forms. Much more research is needed to make such a combination more coherent,

powerful and useful. In Chapter 6, a fundamental relationship between the computation models of predicate transition nets and first order temporal logic has been established, and an algorithm for systematically translating predicate transition nets into equivalent temporal logic formulas has been designed;

6. Prove the various dynamic properties of the Petri nets specification by temporal logic:

   Dynamic properties such as eventuality - total correctness and liveness, and invariance (safety) - partial correctness, mutual exclusion and deadlock freedom can be easily expressed in temporal logic [MP81]. In Chapter 7, a special first order temporal logic verification technique for predicate transition nets has been developed and its application is demonstrated by some well-known examples.

Module specification, called *local specification* in [Hor82], specifies functionality and internal structure of each module. Techniques for module specification have been extensively and intensively studied during the last decade and has reached a mature state ([Jon86], [CIP85]). Two main approaches - denotational and algebraic will be used in this phase. Denotational approach will be used to specify and verify the functionality of a module and algebraic approach will be used to specify and verify the internal data structure of a module. Module specification consists of following steps:

1. Specify and verify the functionality of each component (module) of the refined Petri nets specification using denotational approach:

   The denotational approach adopted here is not the original low level Oxford notation [Sto77] which is very mathematical and can hardly be understood or used in real applications; rather it is VDM-like variation which has been successfully used in many applications [BJ87]. Techniques like structural induction [Jon86] will be used in proving the correctness of specifications;

2. Design algebraic specifications for the internal data structures of each module:
   A more pragmatic algebraic approach like [CIP85] will be used in this step in the specification and verification of data structures instead of more theoretical approaches [GTW78] and [Gut80]. A system for reasoning algebraic specifications such as [LW82] and [CIP87] may also be built for verification purposes, or even term rewriting techniques [Les87] widely used in logic programming can be adopted in verifying the correctness of algebraic specifications.

3. Integrate denotational and algebraic specifications:
   A uniform formal frame which integrates both denotational and algebraic styles will be developed based on [BDMP85], [Jon86] and [CIP85].

Structural specification specifies the structure and functionality of the whole system in a uniform manner. The specification should be correct and ready to be implemented. The operational approach is used to replace and integrate system specifications in Petri nets with the module specifications in denotational and algebraic approaches, the operational approach is powerful enough to capture both parallelism and sequentiality [Plo81], general enough to describe control abstraction, data abstraction and procedural abstraction, practical enough to be directly implementable and machine processable [Bal85], and comprehensive enough to be easily understood, learned and used. The axiomatic approach is used to verify the correctness of such a replacement and the consistency of the static relationship among modules. This phase includes following steps:

1. Translate the denotational specification and the algebraic specification of each module into operational approach:
   A denotational specification such as [Jon86] can easily be translated into an equivalent operational specification. An algebraic specification like [CIP85] is also directly translatable into an equivalent operational specification;

2. Verify the correctness of such an operational specification replacement by axiomatic approach:
   The invariant properties and structural properties should be carried over from the denotational specification and algebraic specification to the operational specification; the axiomatic approach is used to verify such invariant properties [CIP85];

3. Integrate the system structure using operational approach:
   Techniques such as [NH83] may be used to cast Petri net specifications into operational specifications;

4. Prove the correctness of the static relationships among modules using axiomatic approach:
   Properties such as parameter agreement (number and type) need to be checked.

4.4. Discussion

Formal software quality assurance can be partially carried out during the whole software development process in this model. Correctness as the first priority is always to be validated, verified and proven during each stage down to the implementation level. Performance criteria can also be simulated by Petri nets and may be carried on throughout the whole development process. Comprehensibility can be better achieved by the high level structural module and interface design. The interfacing of various specification techniques and automatic tool designing (structural editor, term rewriting system and transformer, theorem prover) are remained to be investigated. The material in this Chapter was published in [HL88].

As pointed out in previous sections, this dissertation only contains partial and preliminary results of this research project which may last for 10 to 15 years based on the experience of similar projects, and mainly concentrates on the system specification level, i.e. integrating predicate transition Petri nets with first order temporal logic in the specification and verification of high level concurrent control structures and properties of software systems. Part II of this dissertation - Chapters 5, 6 and 7, presents some theoretical results of the research on the system specification level.

Chapter 5

A Design Methodology for Predicate Transition Nets

5.1. Introduction

Despite their wide applications [Pet77] and [BRR87], Petri nets have the distinct disadvantage of producing very large and unstructured specifications for the systems being modelled [Bar84]. The introduction of predicate transition nets (abbreviated as PrT nets in the rest of this Chapter) by Genrich and Lautenbach [GL81] has drastically enhanced the expressive power of Petri nets and greatly alleviated the above problems, however to specify a very large system in terms of a predicate transition net is still a formidable task and can result in a huge net too complicated to be readily understood. In order to overcome these problems, a design methodology is needed. Hierarchical (stepwise) and modular development techniques constitute such a design methodology.

The hierarchical (stepwise) and modular design of programs is a well-known programming methodology ([DDH72], [Wir71]) and is widely applied in practical software development. The same methodology is also sought by software specification community in order to make writing specifications of large systems feasible and understandable. Stepwise refinement and abstraction techniques for Petri nets have been explored by several authors: Suzuki & Murata [SM83] and Reisig ([Rei87a], [Rei87b]). In [SM83], a design method for place transition nets (a class of lower level Petri nets) was presented, but unfortunately it could not be directly applied to predicate transition nets. In [Rei87a] and [Rei87b], several heuristic principles were identified and illustrated in developing large PrT net specifications, however a validated methodology was not given. This research work is based on [Rei87a] and further elaborates the transformation techniques for PrT nets. In this Chapter, a new design methodology for writing predicate transition net specifications is presented. The methodology consists of powerful refinement and abstraction, and synthesis and decomposition techniques. A new systematic notational convention is also proposed for such transformation techniques. The concept of separating the specification and body parts of a program unit as used in Ada is adopted in the refinement and abstraction techniques and the state decomposition technique in Statecharts [Har88] is employed in designing various label constructing operators. The methodology will be shown to significantly reduce the complexity in designing large PrT net specifications and enhance the comprehensibility of such specifications.

5.2. Predicate Transition Petri Nets

In this section, the basic concepts and examples of predicate transition nets are informally introduced to motivate the development of the design methodology. Since the structural aspects (syntax and static semantics) of Petri nets are the main concern of the design methodology, the dynamic semantics (firing of transitions) of Petri nets is not discussed. Formal definitions of various concepts of predicate transition Petri nets can be found in Chapter 6. Two well-known problems, the five dining philosophers problem and the reader-writer problem, are first specified by place transition nets and then respecified by predicate transition nets. The expressive power enhanced by predicate transition nets over place transition nets should be clear from these examples.

A net structure is a directed bipartite graph (S,T;F) where $S \bigcup T$ is the set of nodes and $S \bigcap T = \phi$, and F is the set of directed edges. In Petri nets terminology, S is called the set of *places* which are represented by circles; T is called the set of *transitions* which are represented by bars (or boxes); and $F \subseteq (S \times T) \bigcup (T \times S)$ is called the set of flow relations which are represented by directed arcs.

A place transition net is defined as a 5-tulpe $N = (S,T;F,W,M_0)$ where

(1) (S,T;F) is a finite net structure,

(2) W: F $\rightarrow \mathbb{N} \setminus \{0\}$ gives weight to each arc of the net, i.e., the flow capacity of each arc ($\mathbb{N}$ is the set of natural numbers),

(3) M_0: S $\rightarrow \mathbb{N}$ is the initial *marking* which distributes some dots in various places.

The five dining philosophers problem is specified by a place transition net in Figure 5.1 where p_1 to p_5 stand for five philosophers; p_6 to p_{10} stand for five chopsticks; p_{11} to p_{15} stand for eating places; t_1 to t_5 stand for the picking actions; and t_6 to t_{10} stand for the putting actions. The weight for each arc is marked on its side in Figure 5.1. The initial marking is: $\forall$ p $\in \{p_1,\ldots,p_{10}\}$: M_0 (p) = 1.

The reader-writer problem is described in Figure 5.2 (assuming there are five readers and one writer) where p_1 stands for the waiting position of the writer, p_2 stands for the writing position, p_3 stands for resources, p_4 stands for the reading position, and p_5 stands for the waiting position of readers. The weight of each arc is as inscribed in the Figure 5.2 and the initial marking is:

$M_0(p_1) = 1$, $M_0(p_3) = M_0(p_5) = 5$.

From the above examples, we see even for simple problems the resulting place transition nets specifications are large, there are 15 places and 10 transitions in the specification of the five philosopher problem. One can imagine that there will be hundreds or even thousands of places and transitions in a specification of a medium sized problem. Huge, complicated and unstructured descriptions of systems are the major drawback of using place transition nets.

The second drawback of place transition nets is their structural inflexibility, it is very hard to modify an existing place transition net. For example, once a new philosopher is added to the dining philosophers problem, the graphical specification needs to be changed, i.e. an additional part similar to the structure of p_1, p_6, p_{10} , p_{11}, t_1 and t_6 needs to be added into Figure 5.1. Sometimes, the structure of a whole net has to be changed.

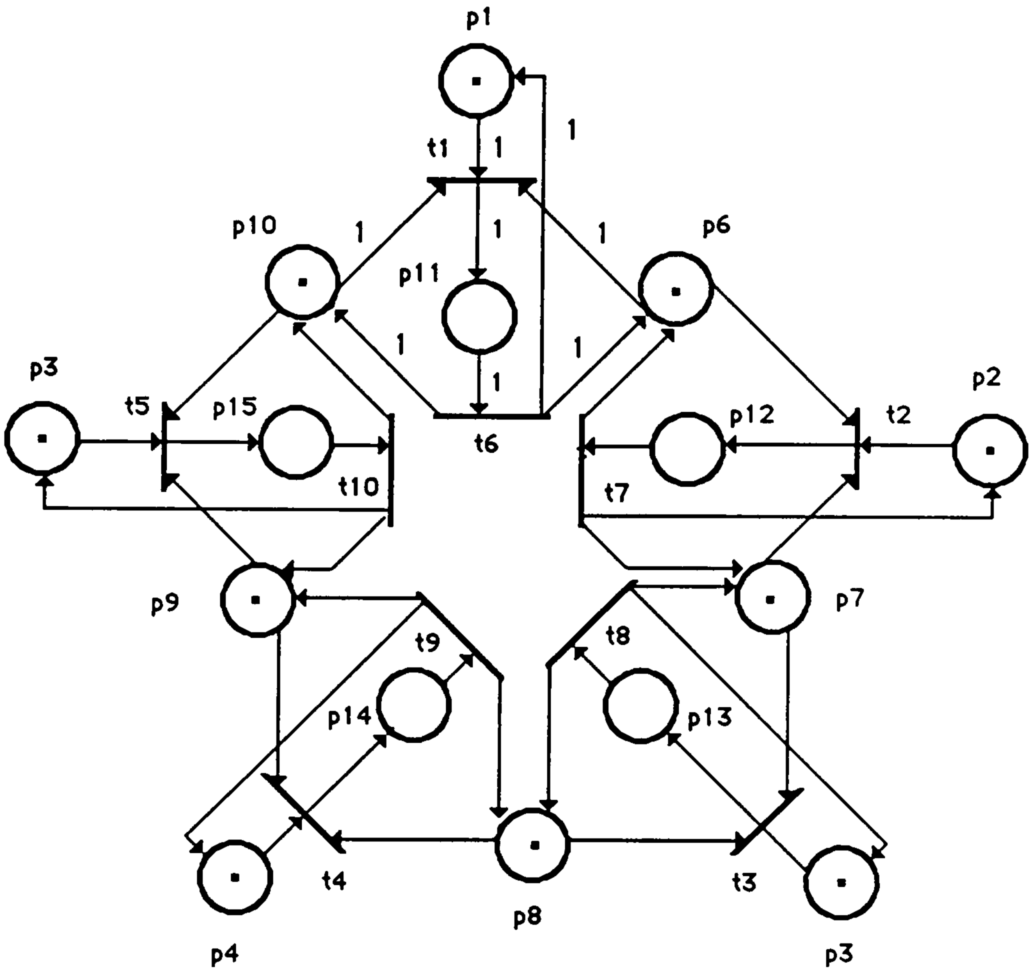


Figure 5.1 - A PT Net Specification of the Five Dining Philosophers Problem

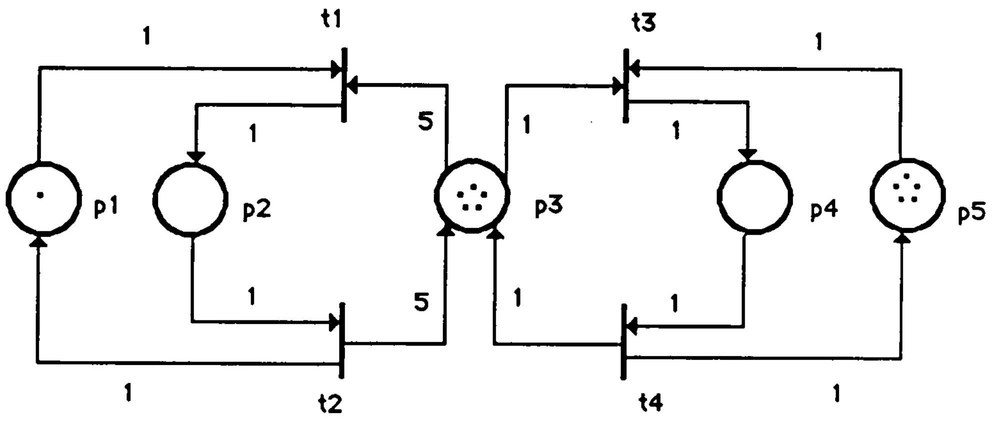


Figure 5.2 - A PT Net Specification of the Reader-Writer Problem

The third drawback of place transition nets is their inability to identify the different tokens, i.e., the simple tokens (dots) in place transition nets lack information. For example, there is no way to distinguish a reader from others in the specification Figure 5.2 of the reader-writer problem.

To overcome the above drawbacks, predicate transition nets [GL81] and colored Petri nets [Jen87] have been developed. Intuition tells us that the equivalent structures such as five repeated philosophers and chopstick nodes can be collapsed into single predicates; this not only reduces the number of places and transitions significantly, but also improves the structural flexibility; therefore effectively overcomes the first two drawbacks. Individualizing tokens, i.e., let tokens carry their own information such as identification number, can get rid of the third drawback.

A *predicate transition net* (PrT) has a net structure (P,T;F) where P is the set of predicates which are first order (or parameterized) places, T is the set of transitions which may contain relational expressions, and F is the set of arcs (flow relations) which may have structural labels (set of tuples) instead of the simple integer numbers as in place transition nets (formal definition of predicate transition nets can be found in Chapter 6).

The five dining philosophers problem is re-specified by the predicate transition net shown in Figure 5.3 where $\oplus$ is modulo 5 addition operator, p_1 is the predicate representing the philosophers identified by $<w,x>$, p_2 is the predicate containing the chopsticks identified by $<x>$, p_3 stands for the eating predicate, and transitions t_1 and t_2 stand for the actions of picking up and putting down chopsticks respectively. The formula $y = x \oplus 1$ specifies the relationship between philosophers and the chopsticks which they can use. For example, philosopher $<w,2>$ can only use chopsticks $<2>$ and $<3>$, otherwise, the transition t_1 is not enabled. Not only the number of elements is drastically reduced from 15 places and 10 transitions in Figure 5.1 to 3 predicates and

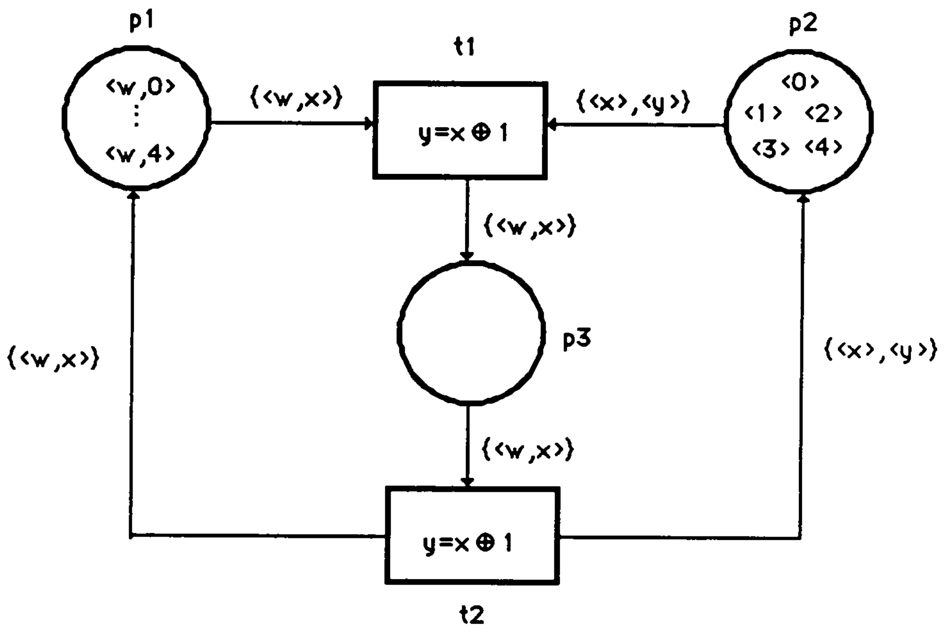


Figure 5.3 - A PrT Net Specification of the Five Dining Philosophers Problem

2 transitions in Figure 5.3, but also the structure of such a specification is stable with respect to either increasing or decreasing the number of philosophers and chopsticks, i.e. when more philosophers and chopsticks are added, only the initial marking needs to be changed while the net structure remains essentially the same.

In order to identify the different reader or to add more writers of the reader-writer problem in the place transition net specification given in Figure 5.2, we have to unfold the right hand side part of Figure 5.2 or to add more identical parts to the left hand side of Figure 5.2 respectively [GL81]. However, this is not necessary when a predicate transition net is used and the re-specification of this problem in predicate transition nets is shown in Figure 5.4, where p_1 is the predicate for waiting writers, p_2 is the predicate for writing, p_3 stands for resources, p_4 is the predicate for reading, and p_5 is the predicate for waiting readers.

5.3. Transformation Techniques for Predicate Transition Nets

From place transition Petri nets to predicate transition Petri nets, great progress has been made within the formalism itself, i.e. by enhancing the foundation of Petri nets. However, it is very difficult or even impossible for users to construct a complete predicate transition net for a system as a single design, especially when the given system is large and very complicated. Transformation techniques serve as a methodology to overcome such a difficulty by providing principles and mechanisms to obtain net specifications.

Transformation in this context means several things. First, it is a structural modification of an unmarked net; although it is very desirable that we also modify marked nets, it is very difficult to achieve that and no such techniques for PrT nets are known. Second, transformation includes both *refinement* and *abstraction*; refinement is to add detail to an existing net while abstraction is to shrink an existing net. Third, transformation

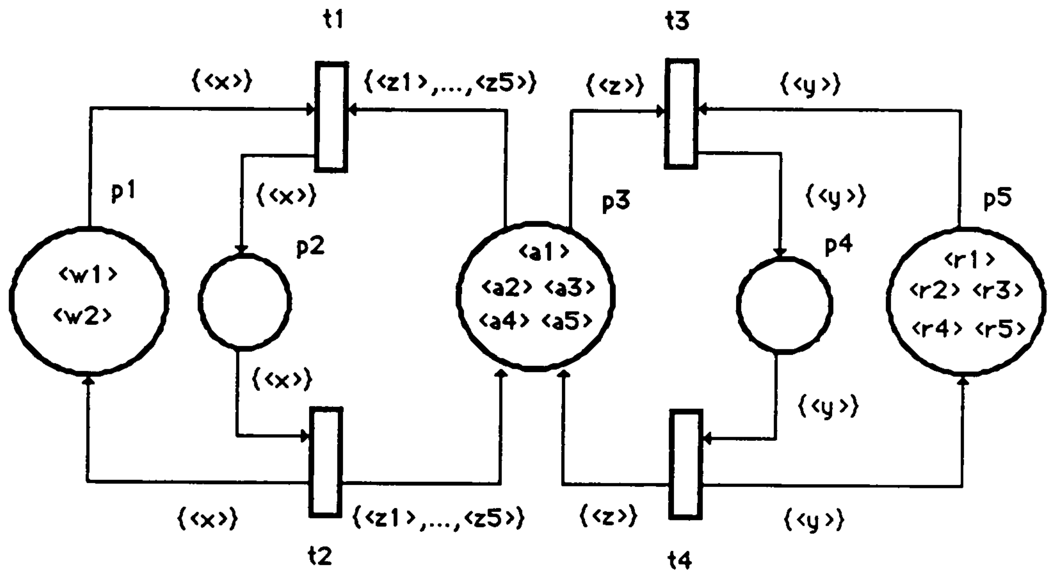


Figure 5.4 - A PrT Net Specification of the Reader-Writer Problem

includes both *synthesis* and *decomposition* ; synthesis introduces additional components into a given net while decomposition separates a net into several subnets.

It is clear from the definitions of place transition nets and predicate transition nets that predicates are the structural abstractions of places. The relationship between place transition nets and predicate transition nets is very similar to that between propositional logic and first order predicate logic. Therefore the advancement from place transition nets to predicate transition nets can be viewed as *place-oriented* abstraction. Another dimension of abstraction , which may be called the *flow relation-oriented* abstraction, is conceivable from the complexity of a general network structure caused by various inter-connections between network nodes (instead of the nodes themselves). Based on the above observation and inspired by Harel's Statechart [Har88] design technique, various label construction operators are designed, which constitute a label algebra. This new flow relation-oriented abstraction by introducing various new label operators may result in a new and even more powerful class of Petri nets - known herein as *clause transition Petri nets,* however this new discovery is temporarily treated as a special refinement and abstraction technique in this dissertation, and its impact on Petri net theory will be studied in a separate project.

5.3.1. Initial Development of Predicate Transition Net Specifications

There are two kinds of components explicitly represented in a net: place/transition or predicate/transition in predicate transition nets. Predicates and transitions are treated equally in nets where predicates are *passive* elements and transitions are *active* elements. This is a major difference between nets and finite state automata where only passive elements (states) are explicitly presented while active elements (state transitions) are implicit functions. Predicates and transitions are connected by a flow relation which has no counterpart in other conventional models (such as state machine).

Realizing this fact, the first thing to do is to distinguish the major passive and active components; for predicate transition nets one only needs to consider one representative of similar elements; then to express them with a suitable net.

5.3.2. Hierarchical Development of Predicate Transition Net Specifications

The hierarchical development method supports both top-down and bottom-up design paradigms which correspond to *refinement* and *abstraction* concepts resectively. Refinement substitutes single elements of a net by another net of arbitrary size; the result of such a refinement should be a more detailed net. Abstraction is the inverse process of refinement and replaces a subnet with a single element (predicate or transition); the result of such an abstraction should be a more concise net.

In order to keep the result of such a transformation again being a net, we define the following concept [Rei87a]:

Given a net $N=(P,T;F)$, let $S=(P_S,T_S;F_S)$ be a subnet of N without isolated elements. Then the *border* (the set of elements of the subnet S which connects to the elements in the net N but not in S) of S in N is defined as

$bd(S) = \{s \in S | \exists\, x \in N\backslash S \wedge ((s,x) \in F \vee (x,s) \in F)\}$

S is called a P-subnet iff bd(S) &inc. P; and S is called a T-subnet iff bd(S) &inc. T.

- Abstraction Rule

Given a net $N=(P,T;F)$, a P-subnet $S=(P_S,T_S;\ F_S\ )$ in N can be replaced by a single predicate p. The result of such an abstraction is a new net $N'=(P',T';F')$ where

$P' = (P - P_S) \bigcup \{p\}$;

$T' = T - T_S$; and

$\forall\, x,y \in (P - P_S) \bigcup (T - T_S)$: $(x,y) \in F \Rightarrow (x,y) \in F'$;

$\forall\, x \in (T - T_S)$: $(\exists\, y \in P_S \wedge ((x,y) \in F \Rightarrow (x,p) \in F' \vee (y,x) \in F \Rightarrow (p,x) \in F'))$.

Similarly, we can define the rule for a T-subnet.

- Refinement Rule

Given a net N, a single predicate p of N can be substituted with a P-subnet of N and the resulting net $N' = (P',T';F')$ is defined as

$P' = (P - \{p\}) \bigcup P_S$;

$T' = T \bigcup T_S$; and

$\forall\, x,y \in (P - \{p\}) \bigcup T$: $((x,y) \in F \Rightarrow (x,y) \in F')$;

$\forall\, x,y \in P_S \bigcup T_S$: $((x,y) \in F_S \Rightarrow (x,y) \in F')$;

$\forall\, x \in (P - \{p\})$: $(\forall\, y \in bd(S) \Rightarrow ((x,p) \in F \Rightarrow (x,y) \in F' \vee (p,x) \in F \Rightarrow (y,x) \in F'))$.

Similarly, we can define the refinement rule for T-subnet when a substitution is made for a transition t of N.

In order to provide mechanisms for refinement and abstraction, the following systematic notations are developed:

- Predicates can be represented by either solid or dotted circles, while a solid circle means an elementary predicate at the current specification level, a dotted (*super*) circle stands for either the abstraction of an existing P-subnet or a P-subnet to be further refined;

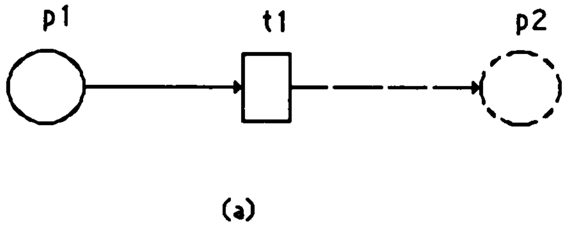
- Transitions can be represented by either solid or dotted boxes, while a solid box means an elementary transition at the current specification level, a dotted (*super*) box stands for either the abstraction of an existing T-subnet or a T-subnet to be further refined;

- Flow relations can be represented by either solid or dotted directed arcs, while a solid arc means an elementary flow relation at the current specification level, a dotted (*super*) arc stands for either the abstraction of a collection of existing flow relations or a collection of flow relations to be further refined or simply a flow relation to a super component.

For each dotted circle or box, there is a corresponding concrete subnet corresponding to it to complete a design at the current specification level. Two situations are allowed:

(1) A subnet is enclosed by a dotted counterpart while still connected to the rest of a complete net. This is called a *united* refinement or abstraction and permits users to focus on the part of the net of current interest, and is the traditional refinement concept of Petri nets;

(2) A subnet is enclosed by a dotted counterpart and isolated from the rest of a net, and the dotted counterpart is connected to the rest of the net by dotted arc(s). This is called a *separated* refinement or abstraction Figure 5.5. The entity encompassed by a dotted component represents a refinement of the dotted component and is a sub-specification at a lower level which can be studied separately. It is the introduction of this new technique which reduces the design complexity of net specifications greatly. The technique is analog to the task specification and task body concepts in the programming language Ada [Ada83]. Nonterminating arcs with different labels may occur in the subnet to keep the correct flow relations of the original or intended net specification and they represent the relations between the subnet and its outside environment (parent net). Default flow relations (without nonterminating arcs) can be used when all of the same kind components (either circles or boxes) in the subnet have the same label to or from the environment Figure 5.6. The idea is similar to that in Statechart [Har87].



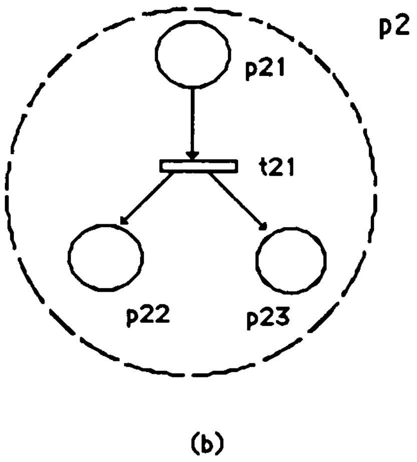


Figure 5.5 - An Example of Refinement

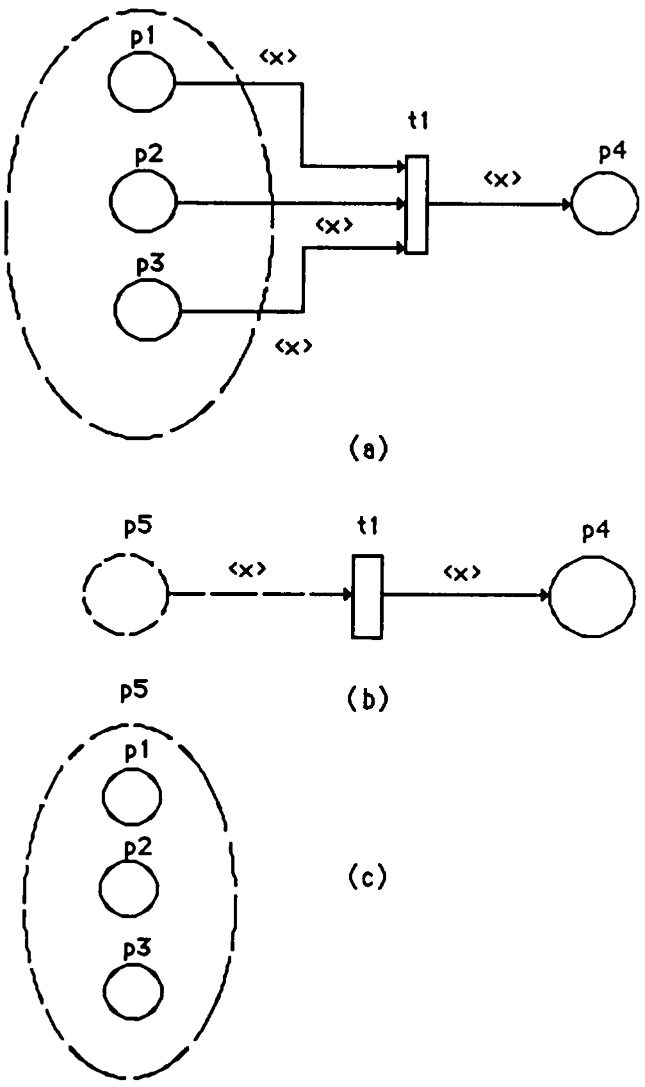


Figure 5.6 - An Example of Abstraction

Labels of the dotted arcs and nonterminating arcs in a subnet are constructed according to the following different situations:

(1) For optional flow relations, the label of the dotted arc is the concatenation of the labels of all these optional flow relations by the operator |. This is shown in Figure 5.7;

(2) For concurrent flow relations, the label of the dotted arc is the concatenation of the labels of all these concurrent flow relations by the operator ||. This is shown in Figure 5.8;

(3) For independent flow relations, the label of the dotted arc is the concatenation of these independent flow relations by the operator |||. This is shown in Figure 5.9.

Parenthesis may be needed to give priority when some combination of the above situations occurs.

It is easy to see that abstraction and refinement are both transitive. Therefore, we can perform such transformations in a stepwise and hierarchical fashion and focus on the most important part of current interest.

5.3.3. Modular Development of Predicate Transition Net Specifications

Synthesis connects two existing nets into a new net (or adds additional components to an existing net). *Decomposition* divides an existing net into two subnets. Synthesis and decomposition are general useful techniques akin to the *modular programming* and *divide and conquer* techniques. When a net grows too big, we can always divide it into several subnets of controllable size and only study the subnet of current interest locally. Synthesis is an most important way to construct a net specification for a large system, it can be used as a modular net specification technique and it can be used when we want to study the global properties of specified systems. It should be noted when this technique is used together with the refinement and abstraction technique, consistency must be maintained for the same predicate or transition scattered in different subnets.

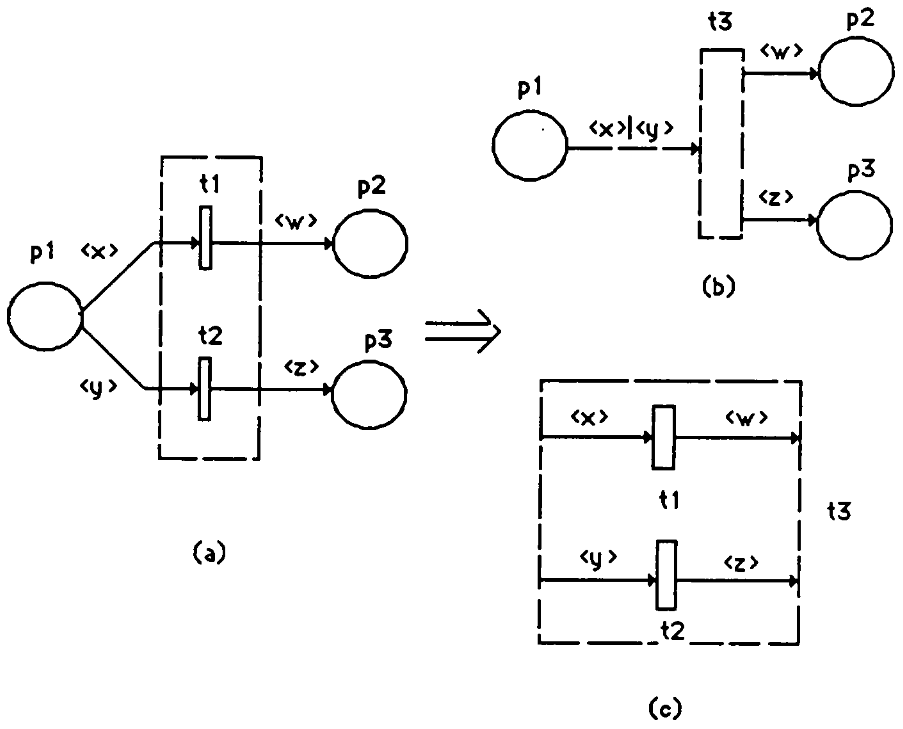


Figure 5.7 - An Example of Optional Flow Relation

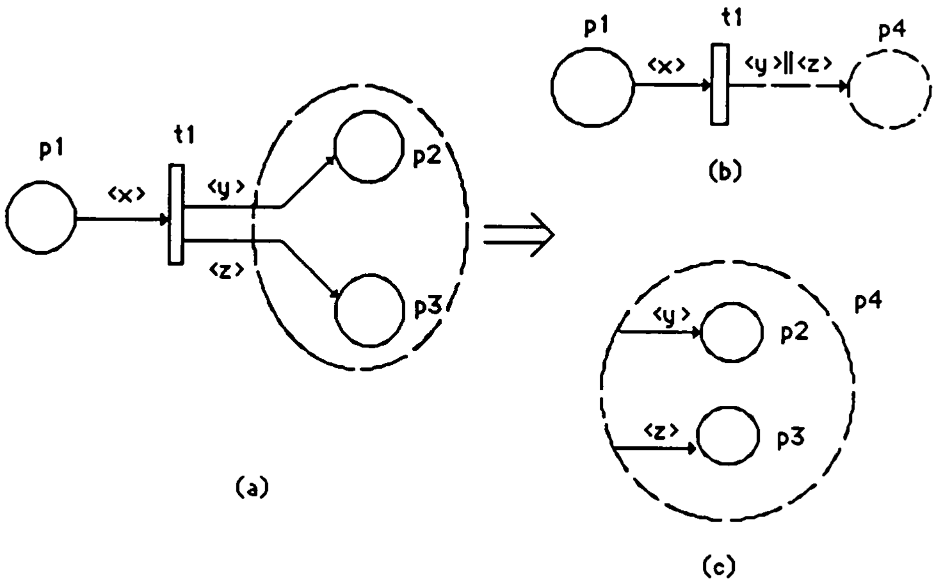


Figure 5.8 - An Example of Concurrent Flow Relation

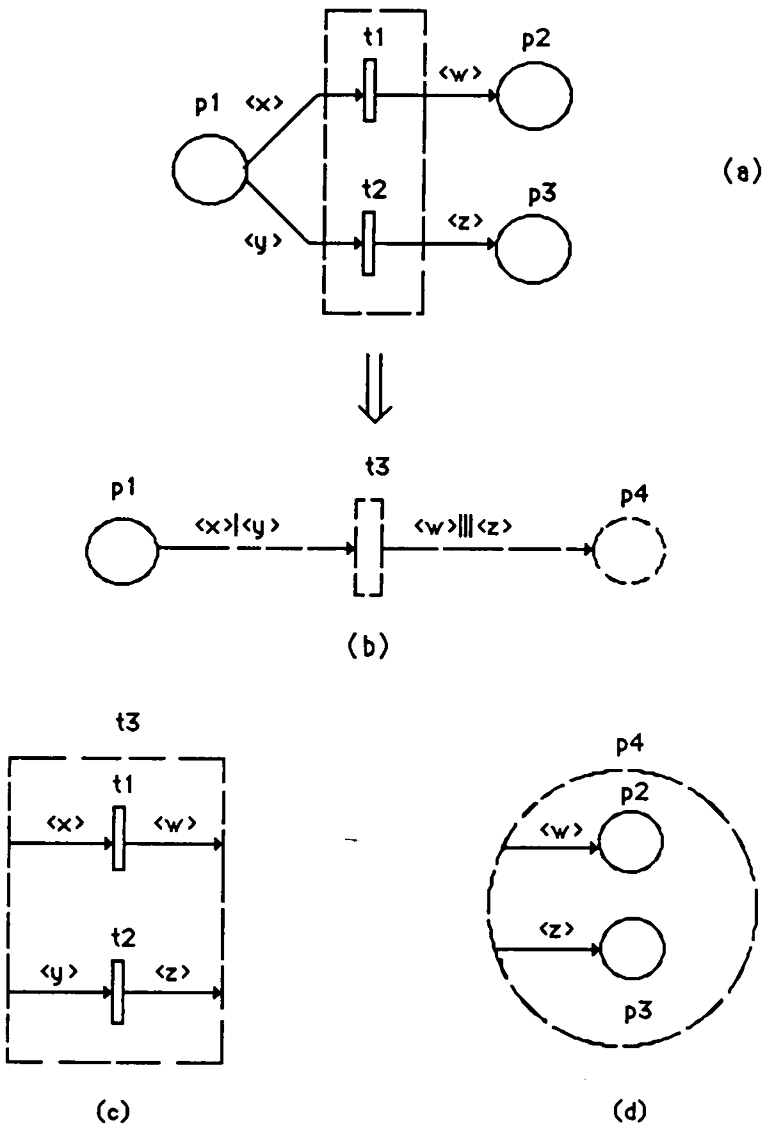


Figure 5.9 - An Example of Independent Flow Relation

- Decomposition Rule

Let N=(P,T;F) be a net, S=$(P_S,T_S;F_S)$ be its subnet and bd(S) be its border in N, a *bd(S)-division* of N results two subnets: S and N′=(P′,T′;F′) where

$P' = (P - P_S) \cup \{p \in P \wedge p \in bd(S)\};$

$T' = (T - T_S) \cup \{t \in T \wedge t \in bd(S)\};$

$F' = F - F_S.$

- Synthesis Rule

Let $N_1=(P_1,T_1;F_1)$ and $N_2=(P_2,T_2;F_2)$ be two nets.

(a) N_1 and N_2 were divided along border bd(S), the result of the synthesis is a new net N=(P,T;F) where

$P = (P_1 \cup P_2) - bd(S);$

$T = (T_1 \cup T_2) - bd(S);$ and

$F = F_1 \cup F_2.$

(b) Either N_1 or N_2 is a new net, the result of the synthesis is a new net N=(P,T;F) where

$P = P_1 \cup P_2;\ T = T_1 \cup T_2;$

$F = F_1 \cup F_2 \cup$ {new connected arcs}.

Synthesizing of two existing nets has the following different situations which are treated differently:

(1) Synthesis of two solid components (nets):

Solid arcs are created with the new labels to reflect new flow relations;

(2) Synthesis of one solid component (net) and one dotted component (net):

(2.1) Separated refinement or abstraction:

Dotted arcs are created with the new labels to reflect new connections; either non-terminating arcs in the dotted subnet are created or new labels are created by using the label operators to reflect the new flow relations with the changed outside environment Figure 5.10;

(2.2) United refinement or abstraction:

Solid arcs are created with the new labels to reflect the new flow relations and these connections extend to the subcomponents of the dotted component (net);

(3) Synthesis of two dotted components (nets):

(3.1) Both being either united refinement or abstraction:

Solid arcs are created with new labels to reflect the new flow relations and the connections extend to the subcomponents of the dotted components (nets) Figure 5.11(a);

(3.2) At least one of them being a separated refinement or abstraction:

Dotted arcs are created with new labels to reflect the new flow relations and either individual non-terminating arcs are created or new labels are created by using the label operator to reflect the changed outside environment in the separated subnet(s) Figure 5.11(b) and (c).

Decomposition is the reverse process of synthesis and therefore deletes connections and/or labels. The decomposition takes the following steps:

(1) Delete flow relations (arcs) to reflect changes and discard the uninterested part(s) of the net;

(2) Delete arcs or discard some part of the labels with the corresponding label operators to reflect the changed outside environment for separated dotted subnet.

A chain modifications for nested refinements might be needed for both synthesis and decomposition techniques, i.e. adding arcs and/or labels while in synthesis and deleting arcs and/or labels while in decomposition must be carried to the elementary (solid) component level.

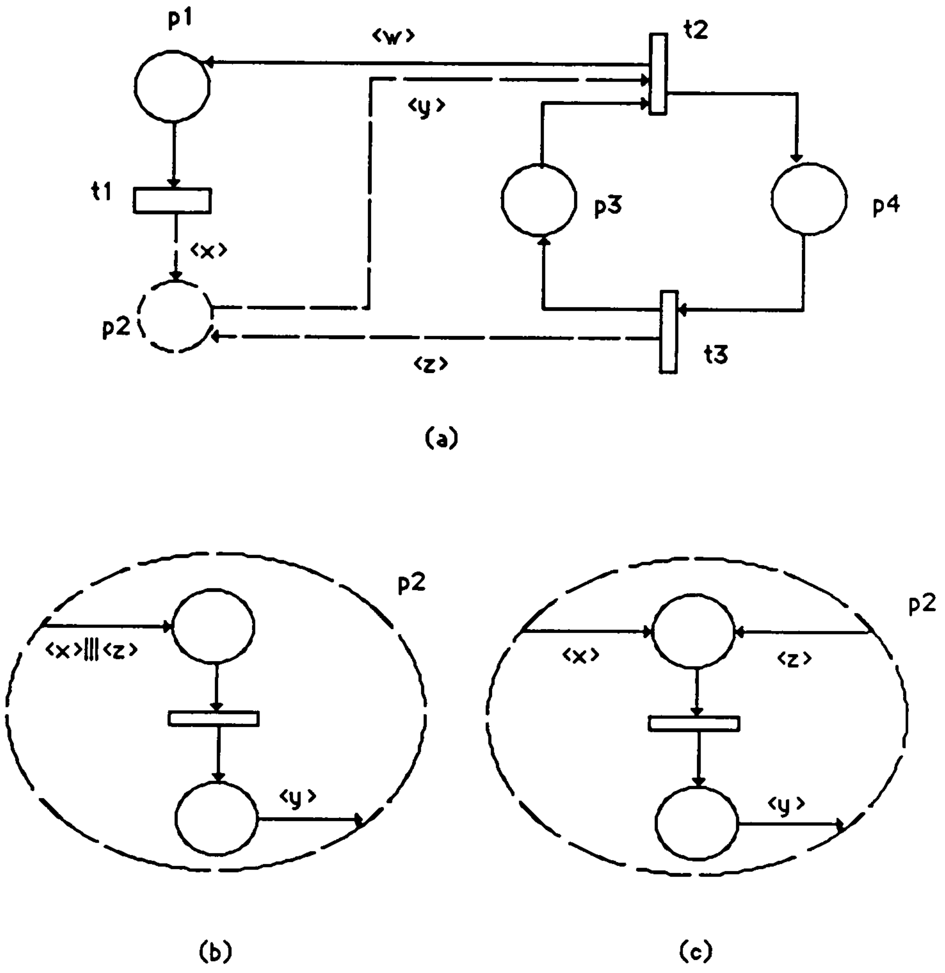


Figure 5.10 - An Example of Synthesis

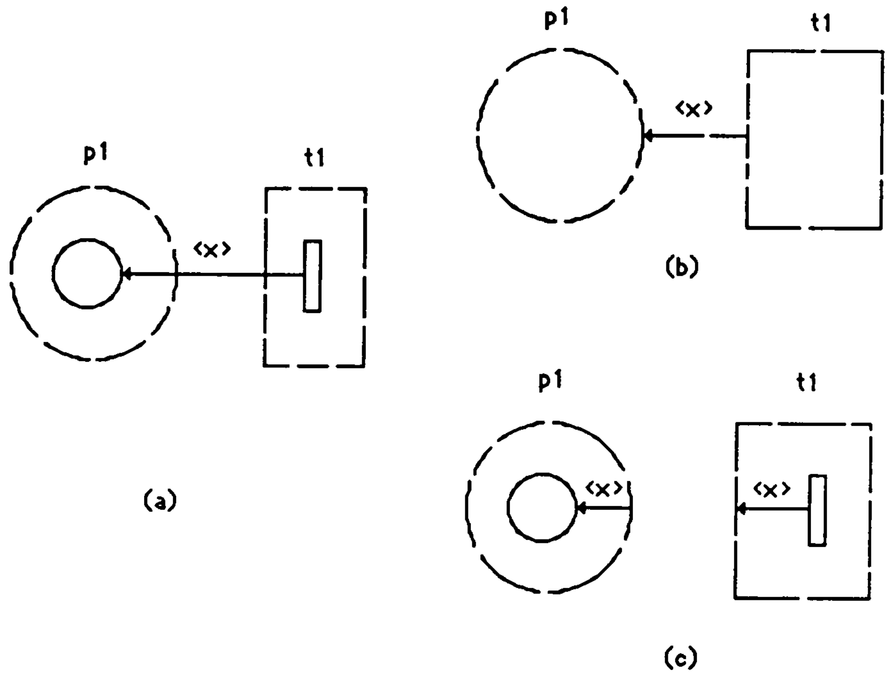


Figure 5.11 - Another Example of Synthesis

5.4. The Consistency of the Transformation Techniques

In the above sections, transformation techniques about Petri net specifications are introduced. While various transformation rules ensure the consistency between the resulting net and its immediate predecessor, the consistency of a whole design still needs to be checked. In [Rei87a], a design *schedule* for refinement and embedding (equivalent to synthesis in this paper) was used to detect several kinds of possible inconsistencies after a serials of refinements and syntheses.

Let single-lined arrows stand for refinements and double-lined arrows stand for syntheses, then the patterns of transformations in Figure 5.12 are inconsistent.

Figure 5.12(a) shows the result of refining a net equals to the result of synthesizing the net, it is obviously inconsistent since refining does not change the scope of the described object while synthesizing enlarges the scope of the considered object.

Figure 5.12(b) is also clearly inconsistent since a net cannot remain the same after a sequence of refinements and syntheses.

Figure 5.12(c) is inconsistent since different refinements should represent the same real object, one of them cannot be the proper subcomponent of the other.

Figure 5.12(d) is the symmetrical counter part of Figure 5.12(c), thus it is inconsistent.

Figure 5.12(e) represents the situation that a net and its refinement unified to one net by adding additional components to both of them. This is certainly inconsistent since the new net contains a refinement of a section of itself.

The above inconsistent transformation patterns cannot occur in our transformation methodology:

Pattern (a) cannot occur since refinement creates new dotted components while synthesis does not.

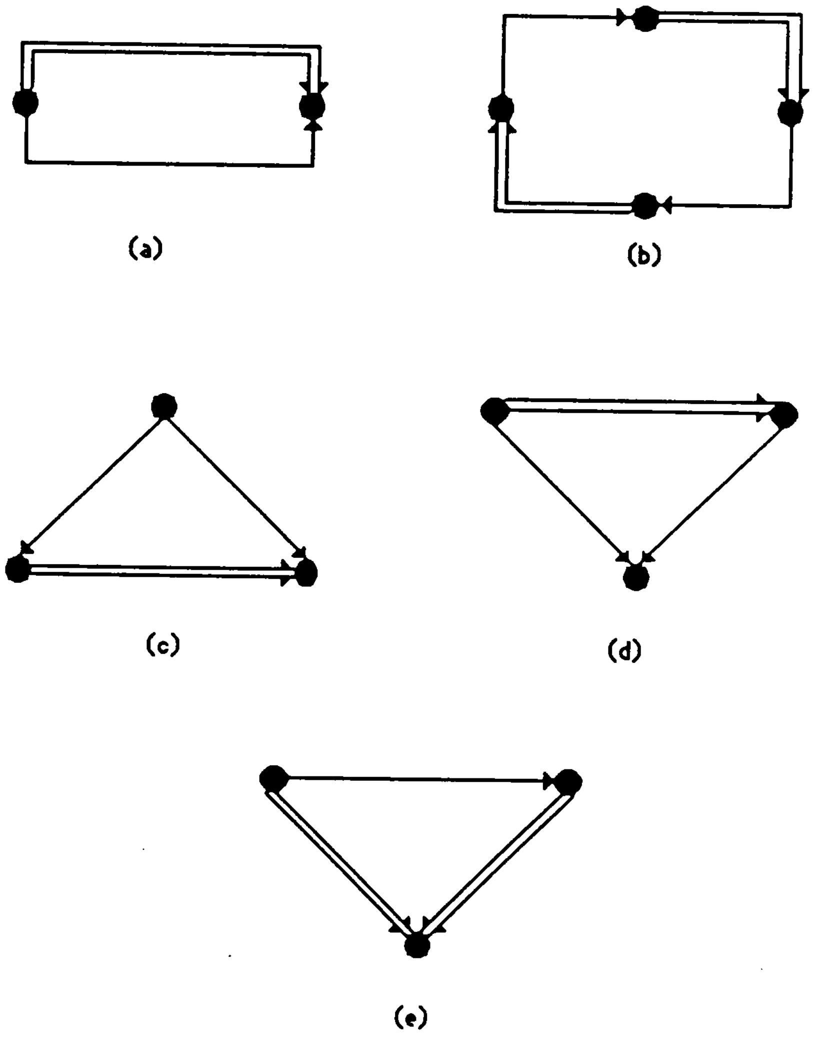


Figure 5.12 - Inconsistent Patterns of Transformations

Pattern (b) cannot occur since refinement creates new dotted components and synthesis does not remove them, and synthesis adds new components to the existing net and refinement does not delete them.

Pattern (c) and (d) cannot occur since synthesis increases the number of elementary (solid) components and refinement does not decrease them.

Pattern (e) cannot occur since refinement creates new dotted components and synthesis does not remove them.

Therefore the transformation techniques are consistent in the sense that the above inconsistent patterns will not be resulted during the transformation process.

5.5. A PrT Net Specification of the Lift Problem

A revised form of the Lift Problem proposed by Davis [Dav87] is specified by using the transformation techniques in this section. The Lift Problem is as follows:

A lift system is to be installed in a building with m floors. The lift and the control mechanisms are supplied by the manufacturer. The internal mechanisms of these are assumed (given). The problem concerns the logic to move the lift between floors according to the following constraints:

(1) The lift has a set of buttons, one for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is cancelled when the corresponding floor is visited by the lift;

(2) Each floor has two buttons (except ground and top floor), one to request an up-lift and one to cause down-lift. These buttons illuminate when pressed. The illumination is cancelled when a lift visits the floor and is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be cancelled;

(3) When a lift has no requests to service, it should remain at its final destination with its door closed and await for further requests;

(4) All requsets for lifts from the floors must be serviced eventually with all floors given equal priority;

(5) All requests for floors within the lift must be serviced eventually, with floors being serviced sequentially in the direction of travel.

At the upper level, the lift system is abstracted as consisting of a Cabin, Floors and a Control System as shown in Figure 5.13. The flow relations between these components are labeled by <x> and <x,y>, where x stands for the floor number requested and y stands for the direction requested.

The Cabin and Floors are further refined as in Figure 5.14 and 15,
where buttons are abstracted as tokens instead of predicates which greatly reduced the size and stablized the structure of the resulting specification since no matter how many buttons (floors) there are, the only change needed is the value of m. The predicates within the Cabin and the Floors contain all the illuminating buttons. When two buttons on the same floor are both pressed, only one of them is illuminating.

The Control System is refined as in Figure 5.16, which contains a request pool of unique requests, a current status indicating where the lift is stopped and its future moving direction, a next floor signal for deciding the lift's next stop, a light off signal for cancelling the illuminating buttons once the corresponding floor is visited, door close and open signals, and three subsystems to be further refined.

The Motor Control System and Door Control System are further refined as in Figure 5.17 and 5.18 respectively. The lift starts moving only when the next destination <x,y> has been selected and the door is closed <w> (the constraint (3) is satisfied). When the lift is stopped, it updates the current status <w,y> and cancels corresponding illumi-

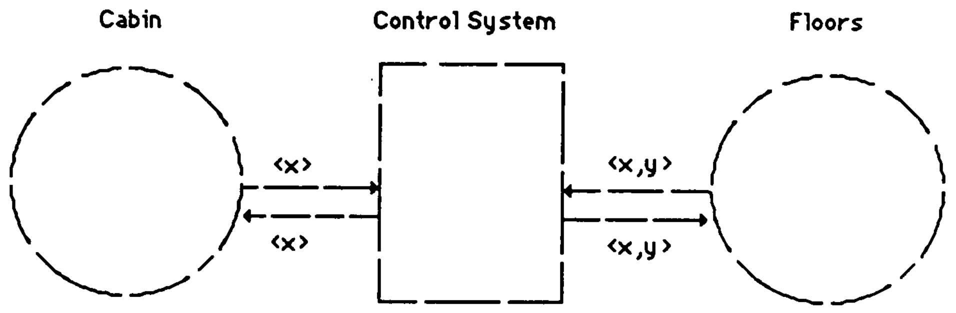


Figure 5.13 - A High Level PrT Net Specification of the Lift Problem

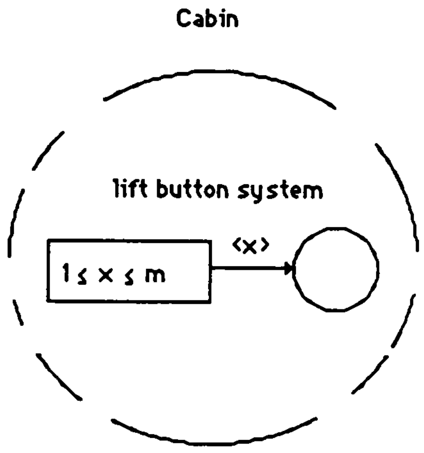


Figure 5.14 - A Refinement of Cabin

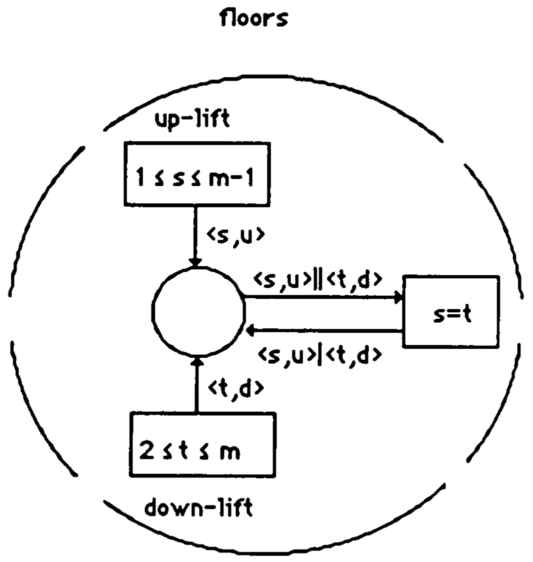


Figure 5.15 - A Refinement of Floors

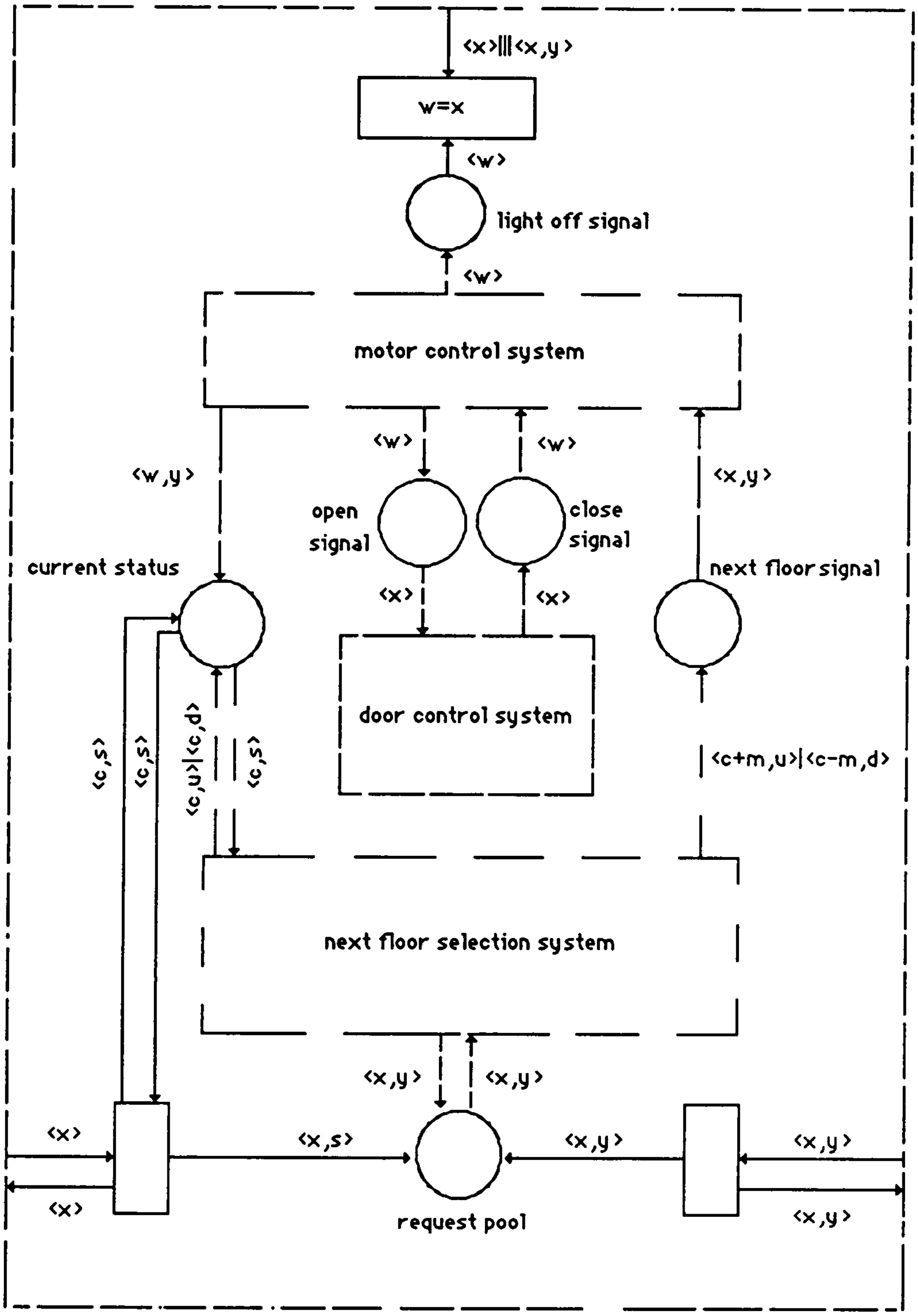


Figure 5.16 - A Refinement of Control System

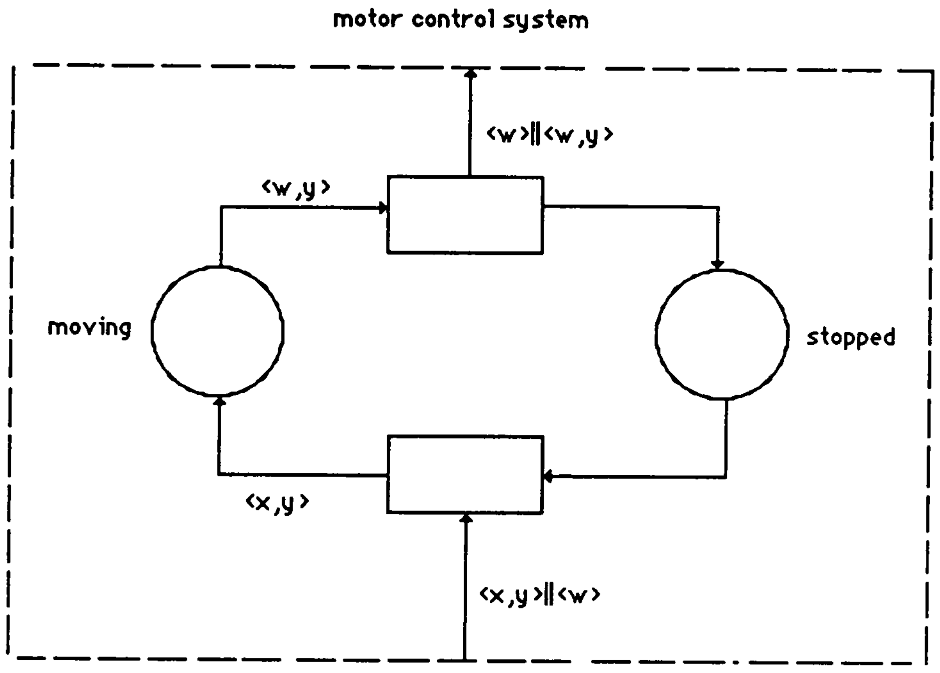


Figure 5.17 - A Refinement of Motor Control System

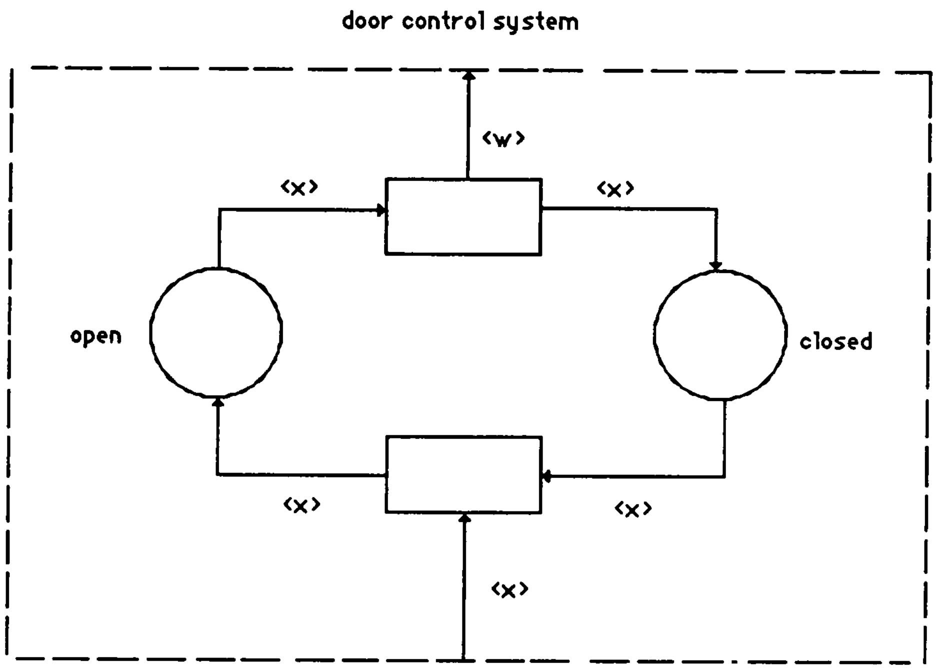


Figure 5.18 - A Refinement of Door Control System

nating button(s) <w> and <w,y> by consuming the corresponding token(s) from the predicates in Cabin and/or Floors (the constraints (2) and (3) are thus assured).

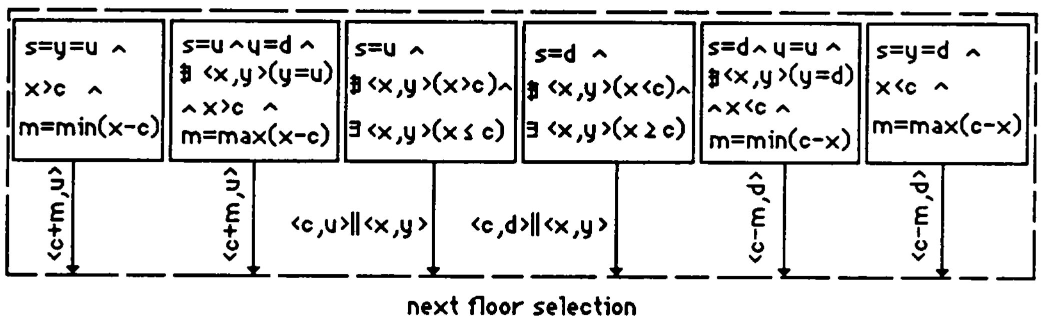
The Next Floor Selection System is further refined as in Figure 5.19. The requests <x,y> from the Request Pool are compared with the current status of the lift <c,s>. Six different situations can be identified (with requests for the current floor ignored) and priority are given to the request closest to the current floor with the same direction as that of the elevator in the common sense. The selection strategy guarantees every request will be eventually served (constraints (4) and (5)).

An integration of the whole system can be easily done by using the synthesis technique.

5.6. Discussion

In this Chapter, a design methodology for predicate transition Petri net specifications is developed. The methodology is so powerful that a huge net specification can be designed or broken into many small manageable sized subnet specifications. For example, a net of one thousand components (predicates or transitions) can be transformed into either 50 subnets each of size 20 or 10 subnets each of size 10 and 10 sub-subnets each of size 10. The design freedom provided by this methodology also facilitates the users to study a logically related subsystem isolately without regards to the activities in the rest of the complete net. The material in this Chapter is published in [HL89b].

Compared to the techniques proposed by Harel [Har88] for Statechart, the techniques proposed in this Chapter for Petri nets are more powerful: while in Statechart, two different situations for XOR (exclusive) and AND decompositions are distinguished by a dashed line in a super state for concurrency; different label operators (| and ||) for forming a new label of a dotted arc is used to differentiate the above situations in our methodology, moreover a third label operator ||| gives the power to define the independent relations (OR relation) of the subcomponents of a super component.



s=y=u ∧ x>c ∧ m=min(x−c)
s=u ∧ y=d ∧ ∄ <x,y>(y=u) ∧ x>c ∧ m=max(x−c)
s=u ∧ ∄ <x,y>(x>c) ∧ ∃ <x,y>(x ≤ c)
s=d ∧ ∄ <x,y>(x<c) ∧ ∃ <x,y>(x ≥ c)
s=d ∧ y=u ∧ ∄ <x,y>(y=d) ∧ x<c ∧ m=min(c−x)
s=y=d ∧ x<c ∧ m=max(c−x)
<c+m,u>
<c+m,u>
<c,u>||<x,y>
<c,d>||<x,y>
<c−m,d>
<c−m,d>
next floor selection

Figure 5.19 - A Refinement of Next Door Selection System

How this methodology affects the dynamic semantics and the analysis techniques of predicate transition Petri nets needs further study.

Chapter 6

Deriving Temporal Logic Formulae from Predicate Transition Nets

6.1. Introduction

While many theoretical and practical formalisms, and methodologies based on formalisms have been developed in dealing successfully with sequential computation models [Bar84], formalisms for studying concurrent and distributed systems have not yet reached a mature stage and few have been used in practice. However, there are two most significant formalisms -- Petri nets and temporal logic -- which have been studied extensively due to their power in specifying and verifying concurrent systems.

Petri nets, designed originally by Petri [Pe62], have gone through a series of extensions. One of the most significant developments is the contribution by Genrich and Lautenbach [GL81] that advanced place transition nets to predicate transition nets and greatly enhanced the expressive power of Petri nets. Petri nets have a simple graphical representation and therefore are very easy to comprehend. Petri nets are well suited for specifying the concurrent structures of software and hardware systems and have been applied widely [BRR87]. The dynamic attribute of Petri nets (marking, firing transitions) makes them also very attractive for simulating the executions of concurrent software systems. However, the only effective method for analyzing the invariant properties of Petri net specifications is the linear algebraic technique ([Lau87], [MV87]). While the linear algebraic technique has been successfully applied to simple classes of Petri nets (condition event nets and place transition nets), it becomes very complicated in dealing with predicate transition nets [Gen87]. In order to study the liveness properties of Petri net specifications, a different method based on the reachability set is needed [HJJJ85].

Temporal logic was first developed by Pnueli [Pnu77] in specifying and verifying concurrent programs. Temporal logic is well suited to specifying and verifying both the invariant (safety) and liveness properties of concurrent systems ([MP81], [Pnu86]), however it is very difficult to be applied to specify the whole system, especially the structural aspect of the system.

It would be a significant progress if the strengths of these two formalisms could be combined so as to use Petri nets as the specification tool and temporal logic as the verification tool. Few authors ([AEI83], [DG83], [GLT80], [QS82]) studied Petri nets and temporal logic together. In [AEI83], representing condition event Petri nets (the simplest class of Petri nets) in propositional temporal logic was exemplified without concern for the verification issues. In [DG83] and [QS82] both Petri nets and temporal logic were used in specifying software problems with the emphasis of using temporal logic for verification purposes. However specifications in the two different formalisms were created separately. In [GLT80], some basic equivalent relationships between the dead transitions (never enabled) of Petri nets and temporal logic formulae were studied without any consideration of specification and verification issues.

In this Chapter, a general algorithm which systematically translates predicate transition net specifications into their equivalent linear time temporal logic specifications is presented. This is the first step towards the integration of high level Petri nets and temporal logic in the specification and verification of software systems; the second step of verifying the Petri net specifications by using temporal logic appears in Chapter 7. The rest of the Chapter is organized as follows. Sections 2 and 3 introduce the basic concepts of Petri nets and temporal logic, respectively, necessary to the understanding of the subsequent translation algorithm. Section 4 introduces the algorithm by which predicate transtion nets are translated into temporal logic formulae. Section 5 summarizes what has been achieved in this method and the necessary subsequent steps.

6.2. Predicate Transition Petri Nets

A net is called *pure* iff it does not have any self-loops, i.e.

$\forall\ (s,t) \in S \times T\ (((s,t) \notin F \wedge (t,s) \notin F))$

A net is called *simple* iff its distinct elements do not have the same pre- and postset, i.e.

$\forall\ x,y \in S \bigcup T\ ((\cdot x = \cdot y \wedge x\cdot = y\cdot) \supset x = y\ )$

In the rest of the Chapter, only pure and simple nets are considered since non-pure and non-simple nets can easily be transformed into pure and simple nets [Lau87].

Let $D = C \bigcup V$ where C is a finite domain of constants and V is a finite set of variables with a range over C and let $\Phi = \{f_1, \dots, f_n\}$ be a finite set of functions over C, i.e. for any function $f \in \Phi$, $f : C \times \dots \times C \rightarrow C$. The set of *terms* is defined recursively as follows:

(1) every $d \in D$ is a term,

(2) if $f^{(n)}$ is a n-ary function $\in \Phi$ and $v_1, \dots, v_n$ are terms, then $f(v_1, \dots, v_n)$ is a term,

(3) no other expression is a term.

The *labeling set* of tuples are defined as follows:

$\forall\ n \geq 0$ and D^n (the set of tuples of length n);

(1) if $v_1,\dots,v_n$ are terms, then $< v_1,\dots,v_n > \ \in D^n$,

(2) if $d_1, d_2 \subset D^n$, then $d_1 \bigcup d_2 \subset D^n$,

(3) no other expression belongs to labeling set.

The set of *relational expressions* is defined as follows:

(1) if v_1 and v_2 are terms, then $v_1 = v_2$, $v_1 \neq v_2$, $v_1 > v_2$, $v_1 \geq v_2$, $v_1 < v_2$ and $v_1 \leq v_2$ are relational expressions,

(2) if r is a relational expression, then $\neg r$ is a relational expression,

(3) if r_1 and r_2 are two relational expressions, then $r_1 \vee r_2$, $r_1 \wedge r_2$, $r_1 \supset r_2$ and $r_1 \equiv r_2$ are relational expressions,

(4) if r is a relational expression and x is a free variable occurred in r, then $\forall_x$ r and $\exists_x$ r are relational expressions,

(5) no other expressions are relational expressions.

The arity of an element $x \in D^n$ is said to be n and denoted by $Ar(x) = n$. We use C^n to denote the set of all constant tuples of arity n (C^0 denotes the empty set).

A *predicate transition net* (abbreviated as PrT nets in the rest of the Chapter) is a 7-tuple $N = (P,T;F,A,W,R,M_0)$ where

(1) (P,T;F) is a net structure, P is called the set of predicates, T is called the set of transitions, both P and T are finite, and F is the set of arcs (flow relations) between P and T;

(2) $A = (C,\Phi)$ is an algebra;

(3) $W: F \rightarrow D^+_{fin}$ is a function from the set of arcs F into the labeling set of tuples D^+_{fin} where $D^+_{fin} = D^1 \cup \ldots \cup D^{max}$ (max is the maximum length of tuples). The function W also has to satisfy the following *consistent condition*:

$\forall\ p \in P\ (\forall\ x,y \in \Sigma_{p \cap \{a,b\} \neq \phi} W(a,b) \supset Ar(x) = Ar(y))$,

i.e. the arity of labels of all arcs incident with a predicate p must be the same;

(4) $R: T \rightarrow \{\lambda\} \cup RE$ is a function from the transition set to the set of relational expressions,

where λ denotes the empty relational expression and RE is the set of relational expressions. The function R has to satisfy the following *relevant condition*:

$\forall\ t \in T$ ({x | x is a free variable in R(t)} $\subset$ {y | y is a free variable in $\forall\ p \in P$ (W(p,t)) or in $\forall\ p' \in P$ (W(t,p'))}),

i.e. all free variables occurred in a relational expression corresponding to a transition t must be in some label of some arc incident to t;

(5) M_0: $P \rightarrow C_{fin}{}^*$ is an initial marking, which is a mapping from the set of predicates to the finite closure of constant tuples ($C_{fin}{}^* = C^0 \cup C^1 \cup \ldots \cup C^{max}$, elements in $C_{fin}{}^*$ are usually called colors [Jen87]). The mapping M_0. has to satisfy the following *strict con-*

dition: there are no multiple occurences of the same color in any predicate, and the *consistent condition*: the arity of colors assigned to any predicate p must be Ar(p).

The above concepts can be illustrated in Figure 6.1 where

$P = \{p_1, p_2\}$; $T = \{t_1\}$;

$F = \{(p_1, t_1), (t_1, p_2)\}$;

$A = (\{a, c\} \bigcup \{true, false\}, \{>\})$;

$W(p_1, t_1) = \{<x>, <y>\}$, $W(t_1, p_2) = \{<x>\}$;

$R(t_1) = x > y$;

$M_0(p_1) = \{<a>, <c>\}$.

The arity of a predicate $p \in P$ is defined to be the arity of its incident arc labels. A marking M of N is a mapping: $M: P \rightarrow C_{fin}{}^*$.

Let E be an expression, $x_1, \ldots, x_n$ be variables and $v_1, \ldots, v_n$ be terms. A *substitution* is defined as $\alpha = \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}$, and the result of the simultaneously substitution of each free ocurrence of x_i in E by v_i is denoted as $E{:}\alpha$. A substitution α is said to be *feasible* for a label W(x,y) of arity n iff $W(x,y){:}\alpha \in C^n \wedge \forall\, w_1, w_2 \in W(x,y)\ ((w_1 \neq w_2) \supset (w_1{:}\alpha \neq w_2{:}\alpha))$, i.e. all tuples in a label should be instantialized to constant labels and different tuples should be instantialized to different constant labels.

Let M be a marking of N, $t \in T$ be a transition, α be a feasible substitution for t, the transition t is called *M-enabled* with α iff

$\forall\, p \in {\cdot t}, (W(p,t){:}\alpha \subseteq M(p)) \wedge R(t){:}\alpha \wedge \forall\, p' \in t{\cdot}\ (W(t,p'){:}\alpha \bigcap M(p') = \phi)$.

The *firing rule* of N is defined as follows: a transition $t \in T$ which is M-enabled with α (after firing) yields a follower marking M′:

$$M'(p) = \begin{cases} M(p) - W(p,t){:}\alpha & \text{iff } p \in {\cdot t} \\ M(p) + W(t,p){:}\alpha & \text{iff } p \in t{\cdot} \\ M(p) & \text{otherwise} \end{cases}$$

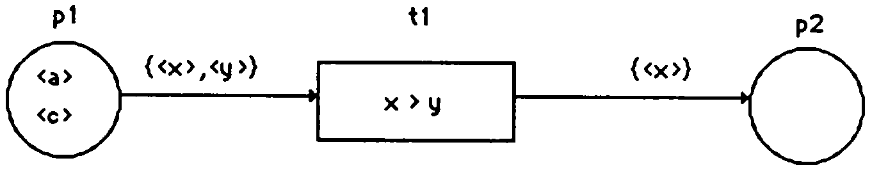


Figure 6.1 - A Predicate Transition Net

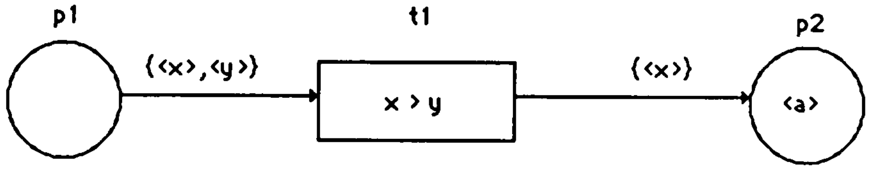


Figure 6.2 - The Marking after Firing of t1

the firing of the transition t with α is denoted by M $[t>_\alpha$ M$'$ and the set of all markings reachable from M_0 is written as $[M_0>$. Two transitions t_1 and t_2 are said to be in *conflicts* if the firing of one of them disables the other. The firing of transitions in conflicts is made nondeterministically. The enabling condition is therefore the necessary condition (not sufficient condition) for the firing of a transition. Two transitions t and t$'$ may occur concurrently with substitutions α and α' respectively (including $t=t'$ or $\alpha=\alpha'$) when they are not in conflicts.

The transition t_1 in Figure 6.1 is enabled with the substitution $\{x \leftarrow a, y \leftarrow c\}$, the result of firing t_1 is shown in Figure 6.2.

An *execution (marking) sequence* E = $M_0 \ldots M_i\, M_{i+1} \ldots$ is a member of $[M_0>^*$ such that E starts with M_0 and for any two adjacent markings M_i and M_{i+1} in E, there is a non-empty set of transitions $T_s \subseteq$ T and a set of feasible substitutions A (one for each transition in T_s) such that $M_i\ [T_s>_A M_{i+1}$. The set of all execution sequences of a PrT net N is denoted as $M_0\ [T^*>$. The semantics of N is defined operationally as the set of all execution sequences $M_0\ [T^*>$ (the execution sequences are equivalent to the step sequences in [Rei85b] in the sense that the same set of marking sequences are characterised, however step sequences are transition sequences instead of marking sequences).

Examples of PrT net specifications of the five dining philosophers problem and the reader-writer problem can be found in Chapter 5.

6.3. First Order Temporal Logic

First order *temporal logic* is a kind of *modal logic* [RU71] which deals with time concepts and is based on first order logic [And86] and usually contains the following *temporal operators*: *always* represented by $\Box$, *sometimes* represented by $\Diamond$ and *next* represented by $\bigcirc$. Their meanings will be given in a following section.

Temporal terms includes ordinary *first order logic terms* - constants, individual variables and function applications such as $f(x_1, x_2, \dots, x_n)$ and $\bigcirc t$ where t is a term. The meaning of $\bigcirc t$ is the next value of t.

Temporal formulae are syntactically defined as follows:

(1) propositions are temporal formulae,

(2) if p is a n-ary predicate and $t_1, \dots, t_n$ are terms, then $p(t_1, \dots, t_n)$ is a temporal formula,

(3) if p is a temporal formula, then $\neg$ p is a temporal formula,

(4) if p_1 and p_2 are temporal formulae, then $p_1 \vee p_2$, $p_1 \wedge p_2$, $p_1 \supset p_2$ and $p_1 \equiv p_2$ are temporal formulae,

(5) if p is a temporal formula and x is an individual variable, then $\forall_x$ p and $\exists_x$ p are temporal formulae,

(6) if p is a temporal formula, then $\Box p$, $\Diamond p$ and $\bigcirc p$ are temporal formulae,

(7) no other expressions are temporal formulae.

A formula without temporal operators is said to be *ordinary* (*static, classical*).

Temporal logic formulae are evaluated under various computation models. A *computation model* for temporal logic is a tuple (I, α, σ) where I is an interpretation which specifies the domains under consideration and assigns meaning to various constants, function and predicate symbols, α is an assignment (substitution) which assigns a value from the appropriate domain to each of the global free individual variables and propositions, and $\sigma = s_0\, s_1 \dots$ is an infinite state sequence. Each state s_i is a mapping from individual variables to values in the appropriate domains specified by I. This definition of the computation model is consistent with that in [MP81].

A temporal logic formula is interpreted over the state sequence σ, therefore it captures the dynamic properties of a computation process while a ordinary first order logic formula is evaluated upon a single state s_i. The behavior (operational semantics) of a specified system is taken as the set Σ of all its computation sequences σ's, therefore the temporal logic used in this dissertation is linear time [MP81].

Let $\sigma^{(k)}$ be a k-shifted sequence: $s_k \; s_{(k+1)} \ldots$ and p be a temporal formula and $w|^{\alpha}_{\sigma}$ be the value of p under α and σ with I being implicitly assumed. The formal semantics of temporal terms and formulae is defined as follows:

(1) y is a local individual variable or local proposition: $y|^{\alpha}_{\sigma} = y s_0$,

(2) x is a global individual variable or global proposition: $x|^{\alpha}_{\sigma} = \alpha[x]$,

(3) c is an individual constant: $c|^{\alpha}_{\sigma} = I[c]$,

(4) f is a n-ary function and $t_1, \ldots, t_n$ are temporal terms:

$f(t_1, \ldots, t_n)|^{\alpha}_{\sigma} = I[f](t_1|^{\alpha}_{\sigma}, \ldots, t_n|^{\alpha}_{\sigma})$

i.e. the value is obtained by applying the interpreted function I[f] to the values of $t_1, \ldots,$ t_n evaluated in the model (I, α, σ),

(5) t is a temporal term: $\bigcirc t|^{\alpha}_{\sigma} = t|^{\alpha}_{\sigma(1)}$,

(6) p is a n-ary predicate and $t_1, \ldots, t_n$ are temporal terms:

$p(t_1, \ldots, t_n)|^{\alpha}_{\sigma} = I[p](t_1|^{\alpha}_{\sigma}, \ldots, t_n|^{\alpha}_{\sigma})$,

(7) p is a temporal formula: $(\neg p)|^{\alpha}_{\sigma} =$ true iff $p|^{\alpha}_{\sigma} =$ false,

(8) p_1 and p_2 are temporal formulae:

$(p_1 \vee p_2)|^{\alpha}_{\sigma} =$ true iff $p_1|^{\alpha}_{\sigma} =$ true or $p_2|^{\alpha}_{\sigma} =$ true,

the semantics of $\wedge$, $\supset$ and $\equiv$ can be derived from $\vee$ and $\neg$,

(9) p is a temporal formula and x is a local individual variable:

$\exists_x\, p|^{\alpha}_{\sigma} =$ true iff for some constant $c \in D$, $p|^{\alpha}_{\sigma} =$ true

where D is the given domain over which x ranges and $\alpha' = \alpha\,[x \leftarrow c]$, the semantics of $\forall$ can be derived from the counterpart $\exists$,

(10) p is a temporal formula: $\bigcirc p|^{\bullet}_{\sigma} = p|^{\bullet}_{\sigma(1)}$

thus $\bigcirc$p means p will be true in next instant,

(11) p is a temporal formula: $\Box p|^{\bullet}_{\sigma} =$ true iff $\forall\ k \geq 0 \supset p|^{\bullet}_{\sigma(k)} =$ true

thus $\Box$p means p is true for all future instants,

(12) p is a temporal formula: $\Diamond p|^{\bullet}_{\sigma} =$ true iff $\exists\ k \geq 0 \supset p|^{\bullet}_{\sigma(k)} =$ true

thus $\Diamond$p means p will be true for some future instant.

For different programming domains (systems, languages), there may be different models. When a formula p is true under a model (I, α, σ), p is said to be *satisfied* by (I, α, σ) or (I, α, σ) *satisfies* p, and this fact is usually denoted by: (I, α, σ) $|=$ p. A formula p is *satisfible* if there is a model satisfying it. A formula p is *valid* in a system with semantics Σ iff every model satisfies it ($\forall\ \sigma \in \Sigma \Rightarrow$ (I,α,σ) $|=$ p) and this is denoted by: (I,α,Σ) $|=$ p. p is called a system dependent theorem.

6.4. Deriving Temporal Logic Formulae from Predicate Transition Nets

Two approaches can be recognized in representing Petri nets in terms of temporal logic -- state-based and event-based approaches. In [AEI83], both approaches were exemplified for condition event Petri nets (abbreviated as CE nets in the rest of this Chapter). As Anttila et al [AEI83] point out, event-based representation does not consider markings and therefore lacks information to distinguish between different cases of a CE net. The above problem of event-based approach becomes more serious when PrT nets are the objects of study since predicates and individual tokens are the vital components of PrT nets. Therefore, the state-based approach is used in this Chapter.

6.4.1. Relating the Computation Models of the Two Formalisms

Unlike in first order logic where the validity of a formula is determined in a static world, in temporal logic the validity of a formula is decided upon in a dynamic world or a series of *changing worlds.* Therefore, a formula can be true in one world and can become false in the next world. The concepts of *world* and *changing world* are two fundamental underlying elements in studying temporal logic. In computer science, *states* and *state transitions* are used to abstract worlds and changing worlds; a computation is abstracted as a sequence of state transitions $\sigma = s_0\ s_1 \ldots$.

A computation is simulated in a marked PrT net by executing the net, i.e. by continuously firing various enabled transitions (possibly concurrently). Each firing of some transition results in a new marking -- a re-distribution of tokens. Therefore, it is natural to define the following relationships:

1. states are markings in PrT nets; one distinguishable state is the initial state s_0 which is the initial marking M_0 of the PrT nets;

2. states transitions are the firings of the enabled transitions of PrT nets; state transition functions are the firing rules of PrT nets.

In PrT nets [GLT80], the term 'predicate' is not used in the mathematical logic sense. Rather predicates are the abstractions of 'first order' places in the sense that a place is not necessarily the atomic unit since it can have its own internal structures. This concept is a simile to that between propositions in propositional logic to predicates in first order predicate logic. In [GLT80], the relationship between PrT nets and first order logic was studied. It has been shown [GLT80] that every first order logic formula can be equivalently represented by a set of *dead transitions* (never enabled) of a PrT net with all predicates having the capacity one (at most one color is allowed at any time). It is clear that the relations are made between logic formulae and transitions (instead of predicates). However, in order to express PrT nets in temporal logic, we need to link predicates in PrT nets with logical predicates. For each predicate p in a PrT net of arity n, an n-ry logical predicate $p(x_1,...,x_n)$ is defined as: $p(c_1,...,c_n)$ = true in a state s iff

$\exists M \in [M_0> \ (<c_1,...,c_n> \in M(p) \wedge s=M)$,

i.e. the meaning of the predicate is defined upon the color set with which p will be marked with during the execution of the PrT net N.

Transitions in PrT nets may contain relational expressions (whose evaluations return true or false), therefore, they are first order logic formulae too and can be translated straightforwardly into temporal logic formulae.

The set of models (I,α,Σ) is obtained from $N = (P,T,F;A,W,R,M_0)$ as follows:

(1) $I = C \cup \Phi \cup \{p \mid$ as defined above$\}$;

(2) α: {the set of free variables occurred in all labels and relational expressions} $\rightarrow C$;

(3) $s_0 = M_0$ and

$\sigma = ... s_i ... \in \Sigma$ iff there is an infinite execution sequence $E = ... M_i ... \in M_0\,[T^*>$ such that $s_i = M_i$ for all $i \geq 0$;

$\sigma = ... s_k\, s_k ... \in \Sigma$ iff there is a finite execution sequence $E = ... M_k \in M_0\,[T^*>$ such that $s_i = M_i$ for $0 \leq i \leq k$ and $s_k = s_{k+j}$ for all $j \geq 0$ (called stuttering in [Lam83]).

6.4.2. Translating Predicate Transition Nets into Temporal Logic Formulae

The equivalent relationship between the two formalisms given in the last section is conceptually elegant, however it is impractical to achieve since both execution sequences and the number of execution sequences can be infinite. Instead of translating the set of marking sequences $M_0[T^*>$ into the temporal logic paradigm, the firing rule of a PrT net which generates the execution sequence set $M_0[T^*>$ can be expressed as the following temporal logic formula:

$\forall$ M $\in [M_0>$ ($\exists$M$' \in [M_0>$ ($\exists$ t $\in$ T ($\forall\beta \in$ {the set of all feasible substitutions}
($\forall$p $\in \cdot t$ (W(p,t):$\beta \subseteq$ M(p)) $\wedge$ R(t):β $\wedge$ $\forall$p$' \in t\cdot$ (W(t,p$'$):$\beta \cap$ M(p$'$)$=\phi$) $\supset$
($\forall$q $\in \cdot t$ (M$'$(q)$=$M(q)$-$W(q,t):β) $\wedge$ $\forall$ q$' \in t\cdot$ (M$'$(q$'$)$=$M(q$'$)$+$W(t,q$'$):β) $\wedge$
$\forall$r $\notin \cdot t \cup t\cdot$ (M$'$(r)$=$M(r)))))) (*)

Though the above translation is straightforward, the obtained formula is complicated and involves many obscure and unessential things such as markings M and M$'$ and the feasible substitution β and many kinds of set operations. A further analysis shows that (1) $\forall$ M $\in [M_0>$ can be represented by introducing the temporal operator always $\square$, (2) the follower marking M$'$ can be implied by using the temporal operator next $\bigcirc$, and (3) the feasible substitution $\forall$ β can be replaced by universally quantifying all free variables occured in the labels of all arcs incident with a transition t and in the relational expression R(t) over the current state s. The following translation algorithm gives a systematic means to obtain a set of temporal formulae (one for initial marking M_0 and one for every transition t in a PrT net N) which are equivalent to (*).

A Translation Algorithm:

(1) $\forall$ p $\in$ P (M_0(p)$=\{<c_1^1,\ldots, c_n^1>,\ldots,<c_1^k,\ldots,c_n^k>\}$), a following sub-formula is created:

p($c_1^1,\ldots, c_n^1$) $\wedge \ldots \wedge$ p($c_1^k,\ldots,c_n^k$)

A formula F corresponding to the initial marking M_0 is obtained by making the conjunctions of all such sub-formula created in step (1);

(2) For every *internal* transition t $\in$ T (both $\cdot t$ and $t\cdot$ are not empty) in a PrT net, a temporal logic formula is generated as follows:

(2.1) $\forall$ p $\in \cdot t$ (for any predicate belongs to the preset of the transition t): two sets of sub-formulae are created:

(2.1.1) { p($x_1,\ldots,x_n$) | $\forall$ $<x_1,\ldots,x_n> \in$ W(p,t) },

(2.1.2) {($x_1,\ldots,x_n$) $\neq$ ($y_1,\ldots,y_n$) | $\forall <x_1,\ldots, x_n>, <y_1,\ldots,y_n> \in$
W(p,t)($<x_1,\ldots,x_n> \neq <y_1,\ldots,y_n>$)}.

where the operation $\neq$ is defined as the pairwise comparison of the terms in the tuples.

A sub-formula A corresponding to $\forall\ p \in {}^{\cdot}t\ (W(p,t) \subseteq M(p))$is obtained by making the conjunctions of all sub-formulae in (2.1.1) and all sub-formulae in (2.1.2);

A sub-formula $\overline{A}$ corresponding to $\forall\ q \in {}^{\cdot}t\ (M'(q)=M(q)-W(q,t))$ is obtained by making the conjunctions of the negation of all sub-formulae in set (2.1.1);

(2.2) $\forall\ q \in t^{\cdot}$ (for any predicate belongs to the postset of the transition t): two sets of sub-formulae are created:

(2.2.1) $\{\ q(x_1,\ldots,x_n) \mid \forall\ <x_1,\ldots,x_n> \in W(t,q)\ \}$,

(2.2.2) $\{(x_1,\ldots,x_n) \neq (y_1,\ldots,y_n) \mid \forall <x_1,\ldots, x_n>, <y_1,\ldots,y_n> \in$

$W(t,q)(<x_1,\ldots,x_n> \neq <y_1,\ldots,y_n>)\}$.

A sub-formula B corresponding to $\forall\ q' \in t^{\cdot}\ (M'(q')=M(q')+W(t,q'))$ is obtained by making the conjunctions of all sub-formulae in (2.2.1);

A sub-formula $\overline{B}$ corresponding to $\forall\ p' \in t^{\cdot}\ (W(t,p') \cap M(p')\ =\ \phi)$ is obtained by making the conjunctions of the negation of all sub-formulae in (2.2.1) and the conjunctions of all sub-formulae in (2.2.2);

(2.3) A sub-formula C is obtained from the relational expression R(t).

The temporal logic formula corresponding to the transition t is:

$\square\ \forall_{\vec{x}}\ (A \wedge \overline{B} \wedge C \supset \bigcirc\ (B \wedge \overline{A}))$

where $\vec{x}$ is the set of all variables occurred in A, B and C (defined as above).

(3) For every *external* transition $t \in T$ (either ${}^{\cdot}t$ or $t^{\cdot}$ is empty) in a PrT net, a temporal logic formula is generated as follows:

(3.1) ${}^{\cdot}t$ is empty: $\square\ \forall_{\vec{x}}\ (\overline{B} \wedge C \supset \bigcirc\ B)$

where B, $\overline{B}$ and C are constructed as in the steps (2.2) and (2.3);

(3.2) $t^{\cdot}$ is empty: $\square\ \forall_{\vec{x}}\ (A \wedge C \supset \bigcirc\ \overline{A})$

where A, $\overline{A}$ and C are constructed as in the steps (2.1) and (2.3).

While the translation algorithm generates a syntactic and semantic correspondence between a PrT net and a set of temporal logic formulae which captures the behavior of each single transition as shown in Figure 6.3, the casual relationship between transition sharing a common input predicate (transitions might be in conflict) as shown in Figure 6.4 is not reflected. Furthermore, the conflicts in a predicate transition net might be caused not only by the above alternative input situation, but also by different simultaneous overlapping subsitutions for the variables of the same predicate to enable the same transition. There are two approaches to solve this problem:

(1) To create new temporal logic formulae to prohibit such conflict situations.

While it is possible to do so for alternative input situation regardless of the resulting very complicated formulae (one formula for each alternative input predicate), it would be very difficult to construct formulae for some simultaneous overlapping substitutions since the number of different combinations of such substitutions is too large. Therefore, this approach is impractical;

(2) To incorporate a new substitution inference rule into the resulting proof system which is adopted in this dissertation as the following uniqueness substitution rule is incorporated into the derived proof system.

The Uniqueness Substitution Rule:

At any time, only one (with a unique substitution including renaming) of the system dependent axioms (obtained from the translation algorithm) capturing the conflict situation can be applied in the proving process.

This inference rule can be viewed as a restricted form of the general substituation and renaming rules in first order logic. The consequence and resulting proof technique of this inference rule is discussed in Chapter 7.

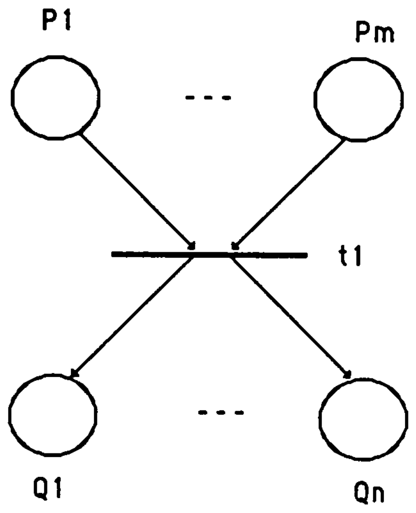


Figure 6.3 - Concurrent Input Pattern

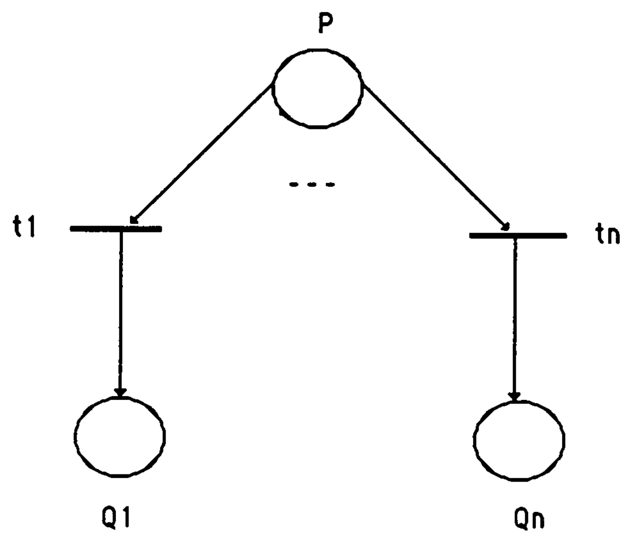


Figure 6.4 - Alternative Input Pattern

Therefore a semantic equivalence between the PrT net specification and the derived temporal logic specification is established since both specifications define the same set of computation sequences. The result is stated in the following theorem:

Theorem For any given PrT net specification, there is an equivalent linear time temporal logic specification.

Proof: The semantics of a given PrT net N = (P,T;F,A,W,R,M_0) is the marking set of M_0[T*> from the definition of N.

In section 6.4.1, a set of models (I, α, Σ) can be effectively constructed from N where $\Sigma = M_0$[T*>.

Based on the structure of the given PrT net N and the firing rule, a set of temporal logic formulae and a uniqueness substitution rule capturing the same causal relations are obtained from the translation algorithm.

The derived temporal logic formulae and the uniqueness substitution rule characterize exactly the set of models (I, α, Σ) derived from the PrT net N, i.e. (I, α, Σ) is the largest set in which the derived temporal formulae are valid. ■

The relationship between these two formalisms is summarized in Figure 6.5.

For example, the temporal logic formulae obtained from the translation of the PrT specification of five dining philosophers problem in Figure 5.3 are as follows:

$$p_2(0) \wedge \ldots \wedge p_2(4) \wedge p_1(w,0) \wedge \ldots \wedge p_1(w,4) \qquad (1)$$

$$\Box \forall_{x,y} (p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge \neg p_3(w,x) \wedge (y = x \oplus 1)$$
$$\supset \bigcirc(p_3(w,x) \wedge \neg p_1(w,x) \wedge \neg p_2(x) \wedge \neg p_2(y))) \qquad (2)$$

$$\Box \forall_{x,y} (p_3(w,x) \wedge \neg p_1(w,x) \wedge \neg p_2(x) \wedge \neg p_2(y) \wedge (y = x \oplus 1)$$
$$\supset \bigcirc (p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge \neg p_3(w,x))) \qquad (3)$$

Formula (1) states that logical predicates p_1 and p_2 are satisfied initially in the world $\{0,\ldots,4\} \cup \{(w,0),\ldots,(w,4)\}$ which reflects the fact that predicates p_1 and p_2 of PrT are marked initially with $\{<w,0>,\ldots,<w,4>\}$ and $\{<0>,\ldots,<4>\}$ respectively.

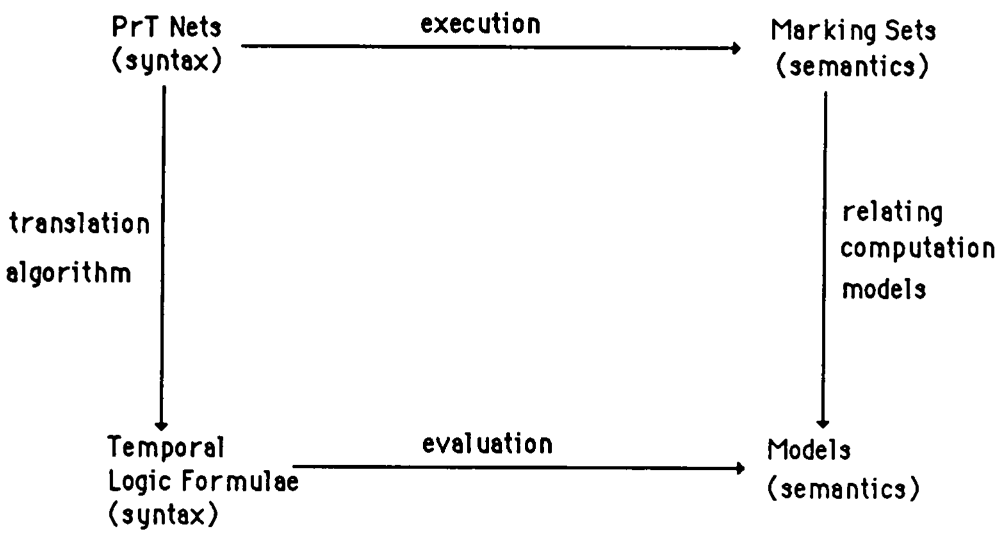


Figure 6.5 - The Relationship between The Two Formalisms

Formula (2) means that once logical predicate p_1 and p_2 are evaluated true under some appropriate substitutions for x and y and y = x $\oplus$ 1, and the logical predicate is not true under the current state, the logical predicate p_3 will become true in the next state under the same substitution and in that state the logical predicates p_1 and p_2 are not true any more under the same substitution. Therefore, the meaning of formula (2) captures exactly the momentum of the firing of transition t_1 in Figure 5.3.

The meanings of other formulae can be understood in the same way.

The temporal logic formulae obtained from the translation of the PrT net specification of the reader-writer problem in Figure 5.4 are as follows:

$$p_1(w_1) \wedge p_1(w_2) \wedge p_3(a_1) \wedge \ldots \wedge p_3(a_5) \wedge p_5(r_1) \wedge \ldots \wedge p_5(r_5) \qquad (1)$$

$$\Box\ \forall_{\vec{x}}\ (p_1(\mathrm{x}) \wedge p_3(z_1) \wedge \ldots \wedge p_3(z_5)) \wedge \neg p_2(\mathrm{x}) \wedge \Lambda_{1 \le i < j \le 5}(z_i \neq z_j)$$
$$\supset \bigcirc\ (p_2(\mathrm{x}) \wedge \neg p_1(\mathrm{x}) \wedge \neg p_3(z_1) \wedge \ldots \wedge \neg p_3(z_5))) \qquad (2)$$

$$\Box\ \forall_{\vec{x}}\ (p_2(\mathrm{x}) \wedge \neg p_1(\mathrm{x}) \wedge \neg p_3(z_1) \wedge \ldots \wedge \neg p_3(z_5) \wedge \Lambda_{1 \le i < j \le 5}(z_i \neq z_j)$$
$$\supset \bigcirc\ (p_1(\mathrm{x}) \wedge p_3(z_1) \wedge \ldots \wedge p_3(z_5) \wedge \neg p_2(\mathrm{x}))) \qquad (3)$$

$$\Box\ \forall_{y,z}\ (p_3(\mathrm{z}) \wedge p_5(\mathrm{y}) \wedge \neg p_4(\mathrm{y}) \supset \bigcirc\ (p_4(\mathrm{y}) \wedge \neg p_3(\mathrm{z}) \wedge \neg p_5(\mathrm{y}))) \qquad (4)$$

$$\Box\ \forall_{y,z}\ (p_4(\mathrm{y}) \wedge \neg p_3(\mathrm{z}) \wedge \neg p_5(\mathrm{y}) \supset \bigcirc\ (p_3(\mathrm{z}) \wedge p_5(\mathrm{y}) \wedge \neg p_4(\mathrm{y}))) \qquad (5)$$

6.5. Discussion

This Chapter relates two very powerful formalisms for specifying concurrent systems - predicate transition Petri nets and linear time temporal logic, a theoretical relationship between the computation models of these two formalisms is established and an algorithm for systematically translating PrT nets into temporal logic formulae is presented, which is much more powerful and general than that proposed in [AEI83]. The translation process can be mechanized. The material in this Chapter is published in [HL89a].

Chapter 7

Verifying Predicate Transition Nets by Temporal Logic

7.1. Introduction

In the previous chapter, how to derive temporal logic formulae and a uniqueness substitution rule from a given predicate transition Petri net was presented. Therefore, the problem of verifying the system properties in the original predicate transition nets is effectively converted into the problem of proving theorems (stating those system properties) in the temporal logic framework. It is usually very easy to express various system properties in succinct and understandable temporal logic formulae [MP81] as will be seen in the following sections, however it is difficult to prove temporal logic formulae in general. Several proof techniques for first order temporal logic have been developed. The traditional tableaux proof technique [Fit72] was not able to handle the modal operator next $\bigcirc$ which plays a vital role in our derived temporal logic specifications. Abadi and Manna's temporal resolution technique [AM86] is too restricted to be only applied in the sound modal context and too complicated to be used in proving various system properties. Ohlbach's resolution technique requires an additional costly syntactical transformation from temporal formulae to ordinary first order logic formulae before the resolution can be performed. In this Chapter, an extended Hilbert-style proof technique [And86] is used in proving various system properties.

7.2. Proof Techniques for First Order Temporal Logic

In Chapter 2, proof techniques for propositional temporal logic were briefly mentioned. Since the satisfiability problem of propositional temporal logic is decidable, the research in propositional temporal logic is mainly focusing on the search for efficient proof techniques. For first order temporal logic, there are both the computability (decidability) problem and the complexity (efficiency) problem since the satisfiability problem of first order temporal logic is undecidable [HIR88]. In the following sections, several proof techniques for first order temporal logic are briefly introduced.

7.2.1. The Semantic Tableau Proof Technique

The semantic tableau proof technique first studied by Beth [And86] for proving first order classical logic formulae is a kind of *refutation* proof technique, i.e. proof by contradiction through the negation of the goal of a given formula to be proven. In the tableau technique, a downward binary tree, called *dyadic* tree, is created by applying a set of inference rules in a given logic. Each node in the tree is labeled with a formula which is either a hypothesis or can be deduced from its ancestors (formulae associated with the nodes on the branch from the root to the current node) by using one of the inference rules. A branch is called *closed* iff there is a formula associated with some node in the branch complementing the formula of the leaf (identical but with different sign). A dyadic tree is closed iff every branch of it is *closed.* Such a tree labeled with formulae created by applying inference rules in a given logic is called a *semantic tableau* [And86]. The following theorem about semantic tableau is well-known [And86]:

Theorem: Let S be a set of formulae of a logic L. S has no model (i.e. unsatisfiable) iff S has a closed semantic tableau.

Therefore the semantic tableaux provides a sound and complete refutation technique for first order classical logic. However there may be no finite semantic tableau for a given set of formulae since the satisfiability problem for first order logic is undecidable.

The semantic tableau proof technique was extended by Fitting ([Fit72], [Fit83]) to deal with first order temporal logic. Theorems similar to that of first order classical logic were obtained for various modal logics [Fit72].

The following example illustrates the proof process of the semantic tableau technique. Figure 7.1 contains a set of inference rules and Figure 7.2 is a semantic tableau proof of the formula: $(\Box\forall_x A(x) \wedge \Box Y) \supset \Box(A(c) \wedge Y)$

7.2.2. The Resolution Proof Technique

The resolution technique pioneered by Robinson [CL73] is also a kind of refutation proof techniques, it is an extension of the following inference rule (called *one-literal* rule):

$P \;\&\; \neg P \vee Q \mid\!- Q$

The *resolution principle* for *clause* formulae (disjunction of *literals* - atomic formula or its negation) can be stated as follows:

Let C_1 and C_2 be two clauses, if there is a literal L_1 in C_1 that is complementary to a literal L_2 in C_2, then the result (*resolvent*) from the resolution of C_1 and C_2 is obtained by deleting L_1 and L_2 from C_1 and C_2 respectively and then disjuncting the remaining clauses together.

The soundness of the resolution principle is stated in the following theorem:

Theorem (Soundness of the Resolution principle): A resolvent C of C_1 and C_2 is a logical consequence of C_1 and C_2.

Given a set S of clauses, a *resolution* (deduction) of C from S is a finite sequence C_1 , C_2, ..., C_k of clauses such that each C_i is either a clause in S or a resolvent of clauses preceding C_i, and C = C_k. The deduction of an empty clause from S is a (refutation) proof of S. The following completeness of resolution principle is well-known [CL73]:

Theorem (Completeness of the Resolution Principle): A set S of clauses is unsatisfiable iff there is a deduction (resolution) of the empty clause from S.

In order to make literals complementary in first order logic, a *substitution* is needed, which is usually expressed as a finite set { t_1/v_1, ... , t_n/v_n } where every v_i is a variable, every t_i is a term different from v_i, and $v_i \neq v_j$ (i ≠ j). A substitution θ is called a *unifier*

α	α_1	α_2
$(X \wedge Y)$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \supset Y)$	X	$\neg Y$
$\neg\neg X$	X	X

β	β_1	β_2
$(X \vee Y)$	X	Y
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$(X \supset Y)$	$\neg X$	Y

ν	ν_0
$\Box X$	X
$\neg \Diamond X$	$\neg X$

π	π_0
$\Diamond X$	X
$\neg \Box X$	$\neg X$

γ	γ_0
$\forall_x A(x)$	$A(c)$
$\neg \exists_x A(x)$	$\neg A(c)$

δ	δ_0
$\exists_x A(x)$	$A(c)$
$\neg \forall_x A(x)$	$\neg A(c)$

α -rule: $\dfrac{\alpha}{\begin{matrix}\alpha_1\\ \alpha_2\end{matrix}}$ where $\dfrac{\alpha}{\begin{matrix}\alpha_1\\ \alpha_2\end{matrix}}$ means that α_1 and α_2 (derived from α) are on the same branch

β -rule: $\dfrac{\beta}{\beta_1 | \beta_2}$ where $\dfrac{\beta}{\beta_1 | \beta_2}$ means the creation of a new branch

ν -rule: $\dfrac{\nu}{\nu_0}$ for any time moment

π -rule: $\dfrac{\pi}{\pi_0}$ for any unrestricted time moment

γ -rule: $\dfrac{\gamma}{\gamma_0}$ for any parameter c

δ -rule: $\dfrac{\delta}{\delta_0}$ for any new parameter c (i.e. not yet used on the branch)

Figure 7.1 - A Set of Inference Rules

(1) $\neg[(\Box\forall_x A(x) \wedge \Box Y) \supset \Box(A(c) \wedge Y)]$			hypothesis
(2) $\Box\forall_x A(x) \wedge \Box Y$			α-rule on (1)
(3) $\neg\Box(A(c) \wedge Y)$			
(4) $\Box\forall_x A(x)$			α-rule on (2)
(5) $\Box Y$			
(6) $\neg(A(c) \wedge Y)$			π-rule on (3)
(7) $\neg A(c)$		(8) $\neg Y$	β-rule on (6)
(9) $\forall_x A(x)$	ν-rule on (4)	(11) Y	ν-rule on (5)
(10) $A(c)$	γ-rule on (9)		
closed (7) & (10)		closed (8) & (11)	

Figure 7.2 - A Tableau Proof of $(\Box\forall_x A(x) \wedge \Box Y) \supset \Box(A(c) \wedge Y)$

for a set $\{ E_1, \dots , E_k \}$ iff $E_1\theta = \dots = E_k\theta$ (where $E_i\,\theta$ stands for the result by substituting those variables in E_i occurring in θ with the corresponding terms in θ). A unifier σ for a set $\{E_1, \dots , E_k\}$ is a *most general unifier* iff for each unifier θ for the set there is a substitution λ such that $\theta = \sigma \circ \lambda$, i.e. θ is a *composition* of σ and λ. For example, the most general unifier for the set:

{ p(a,y), p(x,f(b)) } is $\theta = \{ a/x, f(b)/y \}$

The process for finding a most general unifier for a given set is called *unification.* The following unification theorem is well-known [CL73]:

Theorem (Unification): If S is a finite nonempty unifiable set of expressions, then there is a general unification algorithm which will always terminate and return a most general unifier for S.

Unfortunately, unification in first order logic is undecidable, i.e. the assumption in the above theorem cannot be determined in advance. He et al [HIR88] had a comprehensive overview of various unification problems.

Abadi and Manna [AM86] extended the resolution technique for first order classical logic to first order temporal logic. The classical unification algorithm was extended with the following rules:

(1) Quantifier extension: Let Q be a quantifier and x′ a new variable,

unifier($Qx_1.u_1[x_1], \dots , Qx_m.u_m[x_m]$) is

unifier($u_1[x'], \dots , u_m[x']$)	if the unifier exists and does not bind x′
fail	otherwise

(2) Modality extension: Let M be any of $\bigcirc$, $\Box$ or $\Diamond$,

unifier($Mu_1, \dots , Mu_m$) is

unifier($u_1, \dots ,u_m$)	if the unifier exists
fail	otherwise

The modal operators $\bigcirc$, $\Box$ and $\Diamond$ were treated as unary connectives.

The new nonclausal resolution rule was [AM86]:

$Q_1 x_1 \ldots Q_h x_h .\ A < v_1, \ldots , v_n > ,\ R_1 y_1 \ldots R_k y_k .\ B < v_{n+1}, \ldots , v_m >$

$|- S_1 z_1 \ldots S_{h+k} z_{h+k} .\ [\ A\theta < \text{true} > \ \vee\ B\theta < \text{false} >\ ]$

where θ is a most general unifier of v_i ($1 \leq i \leq m$), Q_j ($1 \leq j \leq h$), R_l ($1 \leq l \leq k$) and S_n ($1 \leq n \leq h+k$) are quantifiers (the notation u<v> indicates that v occurs in u and exactly one occurrence of v is replaced during a substitution, $A\theta$<true> means that each v_i in A after the unification θ are replaced by the truth value true), with the following restrictions:

(1) The same-time restriction: if any flexible symbol (changing over time) occurs in $v\theta$ then the replaced occurrences of $v\theta$ are all in the scope of the same number of O's and are not in the scope of any other modal operator in either $A\theta$ or $B\theta$;

(2) The replaced occurrences of $v\theta$ are not in the scope of any quantifier in either $A\theta$ or $B\theta$;

(3) x_i ($1 \leq i \leq h$) and y_j ($1 \leq j \leq k$) are all different variables;

(4) $S_l z_l$ ($1 \leq l \leq h+k$) is a merge of $Q_i\, x_i$ ($1 \leq i \leq h$) and $R_j y_j$ ($1 \leq j \leq k$);

(5) If $t/x \in \theta$ then for some i ($1 \leq i \leq h+k$), $S_1 = \forall$, $z_i = x$, and no variable in t occurs bound in $\forall z_i\ S_{i+1}\ z_{i+1} \ldots S_{h+k} z_{h+k}$. ($A \wedge B$), i.e. θ only instantiates universally quantified variables, no free variables is captured when θ is applied and t does not depend on x implicitly;

(6) Suppose the replaced occurrences of $v\theta$ are all in the scope of n O's and are not in the scope of any other modal operators in either $A\theta$ or $B\theta$. If $t/x \in \theta$ and a flexible symbol occurs in t, then all occurrences of x in A and B are in the scope of n O's and are not in the scope of any other modal operators in either A or B. This restriction guarantees that if θ indicates x should be equal to t, then θ refers to the value of t in n time units and is only applied in contexts where this would be clear.

The above resolution rule with the restrictions is sound, however it is incomplete [AM86]. The following example from [AM86] illustrates how the resolution technique works. The given set of formulae is:

$\forall x_1 \exists x_2 (\neg\, p(x_1,x_2,a) \vee \bigcirc q(x_1)) \;\&\; \exists\, y_1 \forall y_2 (p(y_1,y_2,a) \vee \Diamond r(f(b)))$

Select

$A = (\neg\, p(x_1,x_2,a) \vee \bigcirc q(x_1)),$

$B = (p(y_1,y_2,a) \vee \Diamond r(f(b))),$

$v_1 = p(x_1,x_2,a),$

$v_2 = p(y_1,y_2,a),$

$\theta = \{ y_1/x_1,\ x_2/y_2 \}$

restrictions (1), (2) and (3) are satisfied.

Therefore the resolvent is

$\exists y_1 \forall x_1 \exists x_2 \forall y_2 ((\neg\, \text{true} \vee \bigcirc q(y_1)) \vee (\text{false} \vee \Diamond r(f(b))))$

which satisfies restrictions (4), (5) and (6) and simplifies to

$\exists y_1 (\bigcirc q(y_1) \vee r(f(b)))$

7.2.3. The Hilbert-Style Proof Technique

A Hilbert-style proof system consists of a set of *axioms schemata* such as: $A \vee A \supset A$ and a set of *inference rules* such as the *modus ponens* rule: $A \;\&\; \neg A \vee B \mid\!- B$. Let $\mathscr{H}$ be a set of wffs (well-formed formulae as defined conventionally [And86]), a *proof of a wff A from the set $\mathscr{H}$ of hypotheses* is a finite sequence $A_1, \dots, A_n$ of wffs such that $A = A_n$ and for each i ($1 \leq i \leq n$) at least one of the following conditions is satisfied:

(1) A_i is an axiom,

(2) A_i is a member of $\mathscr{H}$,

(3) A_i is inferred from some preceding A_k ($1 \leq k < i$) by using one of the inference rules.

A *proof* of a wff A is a proof of A from the empty set of hypotheses. A *theorem* is a wff which has a proof. The fact that A is provable from $\mathscr{H}$ is denoted by $\mathscr{H} \vdash A$ and is called A is *derivable* from $\mathscr{H}$. Therefore derivation $\vdash$ is syntactic whereas satisfaction $\models$ is semantic. Let L be a logic, M be a model of L, $L\vdash$ denote the set of theorems derivable from L, and $(L,M)\models$ denote the set of valid formulae of L under M. The soundness and completeness of L can be described by following relations:

L is sound:	$L\vdash \subseteq (L,M)\models$
L is complete:	$L\vdash \supseteq (L,M)\models$
L is sound and complete:	$L\vdash = (L,M)\models$

The following logic system F has been proven sound (every theorem of F is valid) and complete (every valid wff of F is a theorem - Godel's Complete Theorem) [And86]:

Axiom Schemata:

(1) $A \vee A \supset A$

(2) $A \supset B \vee A$

(3) $(A \supset B) \supset (C \vee A \supset B \vee C)$

(4) $\forall_x A \supset S^x_t A$ where $S^x_t A$ stands for the substitution of x by t in A, and t is a term which is free for individual variable x in A.

(5) $\forall_x (A \vee B) \supset A \vee \forall_x B$ provided that x is not free in A

Inference Rules:

Modus Ponens (MP): $A, A \supset B \vdash B$

Generalization (Gen): $A \vdash \forall_x A$, where x is any individual variable.

The system F is not only sound and complete, but also *independent* (no axiom or inference rule is redundant). Extra axioms and inference rules, though do not increase the expressive power of F, do facilitate the proof processes of wffs and improve the readability of such proofs. Therefore various derived theorems and inference rules based on

F will be used as meta axioms and inference rules in the examples in the following sections.

In Manna and Pnueli [MP81], various valid temporal formulae were presented in Hilbert-style. Some of the temporal axioms and inference rules which are to be employed later are listed below:

Axioms concerning temporal operators:

$\Box A \equiv \Box\Box A$

$\Diamond A \equiv \Diamond\Diamond A$

$\bigcirc A \supset \Diamond A$

Axioms concerning connectives and temporal operators:

$\Box(A \wedge B) \equiv \Box A \wedge \Box B$

$\bigcirc(A \vee B) \equiv \bigcirc A \vee \bigcirc B$

$\Box(A \supset B) \supset (\Box A \supset \Box B)$

$\Box(A \supset B) \supset (\Diamond A \supset \Diamond B)$

$\Box(A \supset \bigcirc A) \supset (A \supset \Box A)$

$\bigcirc(A \supset B) \equiv (\bigcirc A \supset \bigcirc B)$

Axioms concerning quantifiers and temporal operators:

$\Box\forall_x A \equiv \forall_x\Box A$ (Barcan's formula)

$\exists_x\Box A \supset \Box\exists_x A$

Temporal Inference Rules:

$\Box A \vdash A$	$\Box$-instantiation
$\Diamond A \vdash A$	$\Diamond$-instantiation for some new moment m
$A \supset B \vdash \Box A \supset \Box B$	$\Box\Box$-introduction

It is easy to check the validity of above axioms and inference rules from the definition in Chapter 6. Therefore any sound first order classical logic system such as F, extended with the above temporal axioms and inference rules is sound, which is also powerful enough for expressing and proving various system properties of predicate transition nets discussed in the following sections.

7.3. Proving System Properties of Predicate Transition Nets

There are two major classes of properties of a concurrent system - *safety properties* and *liveness properties* (a comprehensive introduction can be found in Lamport [Lam89]). Safety properties are invariance and can be expressed by the temporal formulae of the following general form [MP81]: $\Box$ P, where P stands for some invariant property. Liveness properties state that good things will eventually happen and can be expressed by the temporal formula of the form [MP81]: C $\supset$ $\Diamond$P, where C is the condition initially true and P is a kind of the desired properties. In this Chapter, two kinds of safety properties - deadlock freedom and mutual exclusion and one kind of liveness properties - livelock (starvation) freedom property are studied and illustrated by the five dining philosophers problem.

7.3.1. Proving Safety Properties

A predicate transition net is deadlock free iff either there is at least one transition (with an appropriate substitution) being enabled at any time or a normal terminating state has been reached. Therefore the following temporal formula which expresses the deadlock freedom property for a given PrT net is obtained:

$$\Box\ \exists_{\vec{x}}(F_1 \vee \ldots \vee F_n) \qquad (*)$$

where $\vec{x}$ is the set of all individual variables occurred in all F_i ($1 \leq i \leq n$), each F_i corresponds to the precendent of one of the derived axioms for transitions and is of the form $A \wedge \overline{B} \wedge C$ as explained in Chapter 6, and n is the number of transitions in the PrT net.

The uniqueness substitution rule does not affect the proof of deadlock freedom property since the enabling conditions for all transitions F_i in (*) are in a **or** situation, and therefore can be neglected in the proving process.

Theorem If the derived axioms for all transitions of a given PrT net imply a cyclic formula $\Box(F \supset \bigcirc F)$ and the axiom derived from the initial marking M_0 of the PrT net satisfies F, then the PrT net is free from deadlock.

Proof: from the conditions of theorem, the following valid formulae (1) and (2) are established:

(1) F'	F' is F instantiated under M_0
(2) $\Box(F' \supset \bigcirc F')$	
(3) $\Box(F' \supset \bigcirc F') \supset (F' \supset \Box F')$	Temporal axiom
(4) $F' \supset \bigcirc F'$	MP rule on (2) and (3)
(5) $\Box F'$	MP rule on (1) and (4)
(6) $\exists_{\bar{x}} \Box F$	$\exists$-generalization rule on (5)
(7) $\exists_{\bar{x}} \Box F \supset \Box \exists_{\bar{x}} F$	Temporal axiom
(8) $\Box \exists_{\bar{x}} F$	MP rule on (6) and (7)

Formula (8) implies the general formula (*) for deadlock freedom ∎

The deadlock freedom property for the five dining philosophers problem specified in Figure 5.3 is:

$$\Box\ \exists_{x,y}\ (p_3(w,x) \vee p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge (y=x\oplus 1)) \qquad (*)$$

The formula (*) states that there is at least one transition (with an appropriate substitution) being enabled at any time. The proof for (*) is sketched as follows (some obvious inference steps are skipped):

(1) From the derived axiom (2) of the specification:

$$\Box\ \forall_{x,y}\ (p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge \neg p_3(w,x) \wedge (y=x\oplus 1) \supset$$
$$\bigcirc(p_3(w,x) \wedge \neg p_1(w,x) \wedge \neg p_2(x) \wedge \neg p_2(y)))$$

(2) From $\Box\forall_x A \equiv \forall_x \Box A$ (Barcan's formula):

$\forall_{x,y} \Box\, (p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge \neg p_3(w,x) \wedge (y=x\oplus 1) \supset$

$\bigcirc(p_3(w,x) \wedge \neg p_1(w,x) \wedge \neg p_2(x) \wedge \neg p_2(y)))$

(3) By $\forall$-instantiation of (2):

$\Box(p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0) \supset \bigcirc\, (p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1)))$

(4) Similarly, from the derived axiom (3) and steps (2) and (3):

$\Box(p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1) \supset \bigcirc\, (p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0)))$

(5) By applying $(A\supset B)\wedge(C\supset D)\supset(A\vee C\supset B\vee D)$ and $\Box\Box$-introduction rule to (3) and (4):

$\Box((p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0) \vee p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1)) \supset$

$(\bigcirc(p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1)) \vee \bigcirc(p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0)))$

(6) From $\bigcirc A \vee \bigcirc B \equiv \bigcirc(A \vee B)$:

$\Box((p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0) \vee p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1)) \supset$

$\bigcirc(p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1) \vee p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0)))$

(7) From the derived axiom (1) of the specification:

$(p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0) \vee p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1))$

(8) By applying A and $\Box(A \supset \bigcirc A) \supset (A \supset \Box A)$ to (6) and (7):

$\Box(p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge \neg p_3(w,0) \vee p_3(w,0) \wedge \neg p_1(w,0) \wedge \neg p_2(0) \wedge \neg p_2(1))$

(9) By weakening (8):

$\Box(p_1(w,0) \wedge p_2(0) \wedge p_2(1) \wedge (1=0\oplus 1) \vee p_3(w,0))$

(10) By applying $\exists$-generalization rule to (9):

$\exists_{x,y} \Box\, (p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge (y=x\oplus 1) \vee p_3(w,x))$

(11) By applying $\exists_x \Box A \supset \Box\exists_x A$ and modus ponens rule to (10):

$\Box\, \exists_{x,y}(p_1(w,x) \wedge p_2(x) \wedge p_2(y) \wedge (y=x\oplus 1) \vee p_3(w,x))$

Mutual exclusion property states that only one of the competing processes can enter the critical region at any time, and thus reflects the fact that some markings of the predicate(s) cannot be resulted for a given PrT.

Unlike deadlock freedom property which is global in the sense the whole net (system) is considered, the mutual exclusion property is local to some part (subnet) of a PrT net and concerns only those predicates and transitions involved. Therefore it is impossible to obtain a general formula from the derived axioms as for the deadlock freedom property. However, mutual exclusion property can be expressed by the temporal formula of the form:

$$\Box \forall_{\vec{x}} \neg (F_1 \wedge \ldots \wedge F_n) \qquad (**)$$

Where $\vec{x}$ is the set of individual variables occurred in all F_i ($1 \leq i \leq n$), and each F_i is the precedent of some derived axioms for transitions.

The uniqueness substitution rule plays a vital role in the proving of mutual exclusion property since it prohibits the simultaneous applications of those conflicting (either from sharing a common input predicate or from different overlapping substitutions) derived axioms.

Theorem A subnet of a given PrT net has the mutual exclusion property iff the formula $\Box \forall_{\vec{x}} \neg (F_1 \wedge \ldots \wedge F_n)$ capturing the property can be derived.

Proof: by the definition of the formula $\Box \forall_{\vec{x}} \neg (F_1 \wedge \ldots \wedge F_n)$. ∎

The proving technique for the mutual exclusion property is illustrated by the following example. The mutual exclusion property for the five dining philosophers problem means that no adjacent philosophers can eat at the same time, which can be expressed by the formula:

$$\Box \forall_x \neg (p_3(\mathrm{w,x}) \wedge p_3(\mathrm{w,x} \oplus 1)) \qquad (**)$$

Instead of proving the formula (**) directly, which is quite difficult to achieve, we prove its negation by refutation:

(1) Assuming the negation of (**):

$$\Diamond \exists_x (p_3(\mathrm{w,x}) \wedge p_3(\mathrm{w,x} \oplus 1))$$

(2) $\Diamond$-instantiation of (1) to a special moment n:

$\exists_x (p_3(w,x) \wedge p_3(w,x \oplus 1))$

(3) $\exists$-instantiation of (2):

$p_3(w,a) \wedge p_3(w,a \oplus 1)$

(4) From the derived axioms (a detailed proof is ommitted):

$\Box\forall_x(p_3(w,x) \supset \neg p_2(x) \wedge \neg p_2(x \oplus 1) \wedge \neg p_1(w,x))$

(5) $\Box$-instantiation, $\forall$-instantiation of (4):

$p_3(w,a) \supset \neg p_2(a) \wedge \neg p_2(a \oplus 1) \wedge \neg p_1(w,a))$

(6) $\Box$-instantiation, $\forall$-instantiation of (4):

$p_3(w,a \oplus 1) \supset \neg p_2(a \oplus 1) \wedge \neg p_2(a \oplus 2) \wedge \neg p_1(w,a \oplus 1)$

(7) Weakening of (3), modus ponens with (5), then strenthening:

$\neg p_2(a) \wedge \neg p_2(a \oplus 1) \wedge \neg p_1(w,a) \wedge p_3(w,x)$

(8) Weakening of (3), modus ponens with (6), then strenthening:

$\neg p_2(a \oplus 1) \wedge \neg p_2(a \oplus 2) \wedge \neg p_1(w,a \oplus 1) \wedge p_3(w,x \oplus 1)$

(9) By applying uniqueness substitution rule and derived axiom:

$\bigcirc(p_1(w,a) \wedge p_2(a) \wedge p_2(a \oplus 1) \wedge \neg p_3(w,a)) \wedge \bigcirc p_3(w,a \oplus 1)$

(10) By weakening (9):

$\bigcirc p_2(a \oplus 1)$

(11) By weakening (9):

$\bigcirc p_3(w,a \oplus 1)$

(12) $\Box$-instantiation, $\forall$-instantiation of (4):

$\bigcirc(p_3(w,a \oplus 1) \supset \neg p_2(a \oplus 1) \wedge \neg p_2(a \oplus 2) \wedge \neg p_1(w,a \oplus 1))$

(13) Applying $\bigcirc(A \supset B) \equiv \bigcirc A \supset \bigcirc B$ rules and modus ponens rule to (12) and (11):

$\bigcirc(\neg p_2(a \oplus 1) \wedge \neg p_2(a \oplus 2) \wedge \neg p_1(w,a \oplus 1))$

(14) Weakening (13): $\bigcirc \neg p_2(a \oplus 1)$

Formula (**) is proven by the contradiction of (10) and (14).

7.3.2. Proving Liveness Properties

A predicate transition net is livelock free iff for any infinitely often enabled transition (with an appropriate substitution), the same transition with the substitution will fire infinitely often. In other words, a process (represented by a color tuple) must have a fair chance to enter the critical region and must progress eventually.

The livelock freedom property concerns each individual transition with a special substitution and can be captured by the temporal formula:

$$\forall_{\vec{x}}(\Box\Diamond F \supset \Box\Diamond \overline{F}) \qquad (***)$$

where $\vec{x}$ is the set of all individual variables occurred in F and $\overline{F}$, F is the precedent of one of the derived axioms for transitions and is of the form $A \wedge \overline{B} \wedge C$ (as defined in Chapter 6) and $\overline{F}$ is B (as defined in Chapter 6). For each transition, there is one formula like (***) corresponding to it.

In order to prove livelock freedom property, a *fair* firing rule for enabled transitions need to be added to the definition of PrT nets so that only fair execution sequences are allowed ([Pnu86] and [HRY88]). Therefore a **fair uniqueness substitution rule** (instead of the old uniqueness substitution rule) is derived from the translation, which not only prohibit the firing of conflicting transitions, but also allows a specific (infinitely often enabled) transition to fire.

Theorem The derived axioms and the fair uniqueness substitution rule for a given PrT net guarantees the livelock freedom of the PrT net.

Proof: by the definition of the fair uniqueness substitution rule and definition of the livelock freedom property ■

One of the liveness (starvation free) property for the five dining philosophers problem is that every individual philosopher will eventually get a piece of meal, which can be expressed as:

$$\forall_x(\Box\Diamond(p_1(w,x) \wedge p_2(x) \wedge p_2(x \oplus 1) \wedge \neg p_3(w,x)) \supset \Box\Diamond p_3(w,x)) \qquad (***)$$

The technique for proving livelock freedom is illustrated below:

(1) From the derived axiom (2):

$\Box\forall_{x,y} (p_1(w,x) \land p_2(x) \land p_2(y) \land \neg p_3(w,x) \land (y=x\oplus 1) \supset$

$\bigcirc(p_3(w,x) \land \neg p_2(x) \land \neg p_2(y) \neg p_1(w,x)))$

(2) By applying $\Box\forall_x A \equiv \forall_x \Box A$ to (1):

$\forall_{x,y} \Box (p_1(w,x) \land p_2(x) \land p_2(y) \land \neg p_3(w,x) \land (y=x\oplus 1) \supset$

$\bigcirc(p_3(w,x) \land \neg p_2(x) \land \neg p_2(y) \neg p_1(w,x)))$

(3) By applying $\forall$-instantiation rule to (2)

$\Box(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x) \supset$

$\bigcirc(p_3(w,x) \land \neg p_2(x) \land \neg p_2(x\oplus 1) \neg p_1(w,x)))$

(4) By applying weakening and modus ponens rules to (3):

$\Box(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x) \supset \bigcirc p_3(w,x))$

(5) From $\Box A \equiv \Box\Box A$:

$\Box\Box(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x) \supset \bigcirc p_3(w,x))$

(6) By applying $\Box(A\supset B) \supset (\Diamond A \supset \Diamond B)$ and modus ponens rule to (5):

$\Box(\Diamond(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x)) \supset \Diamond\bigcirc p_3(w,x))$

(7) By applying $\Box(A\supset B) \supset (\Box A \supset \Box B)$ and modus ponens rule to (6):

$\Box\Diamond(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x)) \supset \Box\Diamond\bigcirc p_3(w,x)$

(8) By applying $\bigcirc A \supset \Diamond A$ and modus ponens rule to (7):

$\Box\Diamond(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x)) \supset \Box\Diamond\Diamond p_3(w,x)$

(9) By applying $\Diamond A \equiv \Diamond\Diamond A$ and modus ponens rule to (8):

$\Box\Diamond(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x)) \supset \Box\Diamond p_3(w,x)$

(10) By applying $\forall$-generalization rule to (9):

$\forall_x (\Box\Diamond(p_1(w,x) \land p_2(x) \land p_2(x\oplus 1) \land \neg p_3(w,x)) \supset \Box\Diamond p_3(w,x))$

7.4. Discussion

In this Chapter, the temporal logic formulae obtained from the translation are served as system dependent axioms which are used together with the uniqueness substitution rule and pure temporal logic axioms and inference rules to prove various safety and liveness properties in the original PrT net specification. A special proving technique based on Hilbert-style logic system is proposed and illustrated through the five dining philosophers problem. Therefore, temporal logic is used as an effective verification tool for PrT net specifications. This provides another prospective analysis technique other than the current workhorse linear algebraic technique for analyzing invariant properties and reachability set method for analyzing liveness properties for PrT nets. One advantage of using logic formalism is that the same framework can be used in both specifying and verifying system properties. Another advantage of using logic proving systems is the potential that such proof processes could be mechanized and automatically carried out. The material in this Chapter is to be published in [HL89c].

There are other properties of predicate transition nets such as *accessibility* and *termination* that can be studied in this framework. The Hilbert-style proof technique, though is straight forward and quite easy to understand, is very inefficient by generating search spaces that contain too much redundancies. More effective proof techniques such as Matrix Proof Technique [Wal87] need to be investigated in order to make this approach practical.

Chapter 8

Conclusion

This dissertation contains the preliminary results of an intended long-term research project for integrating formal specification and verification methods in software development, and consists of two major parts:

(1) the design of the specification transition paradigm for software development (Chapter 2 to Chapter 4), and

(2) the theoretical study of the system specification phase in the paradigm (Chapter 5 to Chapter 7).

The development of the specification transition paradigm has been greatly inspired and influenced by several well-known software development projects - the Automatic Programming Project [Bal85], the CIP Project ([CIP85], [CIP87]), the Larch Project [GHW85] and the RAISE Project ([BDMP85], [MG87]), with the aim to increase software productivity, to enhance software reliability and ultimately to ensure software correctness. More specifically, this paradigm has adopted and adapted the new software development life cycle model from the Automatic Programming Project, the specification classification from the Larch Project, and the integration of formal methods from the CIP Project and the RAISE Project. Despite its similarities to the above mentioned projects, this paradigm contains foremost formalisms (predicate transition Petri nets, first order temporal logic, the algebraic, the axiomatic, the denotational and the operational approaches) than any other known models, and has the following distinctive features:

(1) Specifications are viewed both as a set of products and a set of well-defined steps of a process,

(2) Specifications (as a set of products) at different development steps are to be written and verified by different formalisms,

(3) Specification (as a process) spans from the requirement phase to the detailed design phase,

(4) Specification for both concurrent and sequential software is supported, and

(5) Specifications for different aspects (concurrent control abstraction, data abstraction and procedural abstraction) of a piece of software are dealt with separatedly.

The theoretical study of the system specification phase has been motivated by the analysis that system specification is still not well understood [Hor82], there is still a lack of mature formal methods for concurrency [Bar84], and there is a need to combine different formalisms in software specification and verification [Hara88]; and has generated the following results:

(1) a design methodology for predicate transition nets, which incorporates the separate definition technique in Ada [Ada83] and state decomposition technique in Statechart [Har88] into the traditional transformation techniques for Petri nets, and therefore will significantly reduce the design complexity and enhance the comprehensibility of large predicate transition net specifications;

(2) the establishment of a fundamental relationship between predicate transition nets and first order temporal logic and the design of an algorithm for systematically translating predicate transition nets into equivalent temporal logic formulae. Therefore the goal to combine the strengths of both formalisms, i.e. to use predicate transition nets as a specification method and to use temporal logic as a verification method is achieved; and

(3) the discovery of a special temporal logic proof technique based on Hilbert-style logic system to verify various properties of predicate transition nets and the associated theo-

rems. Thus temporal logic is effectively used as an analysis method for both safety and liveness properties of predicate transition nets.

This research has so far directly produced the following publications: [HL88] and [HL89a - e], and has also resulted in other related publications: [HX85], [HRD87], [HIR88] and [LH90].

Since it is just the beginning of an intended long-term research project, this dissertation research has opened many future research problems. In terms of the long-term goals of this research project, i.e.

(1) build a software development model employing most advanced formal methods,

(2) develop a methodology to support such a model,

(3) design a spectrum of specification language, and

(4) construct computer aided tools (software) to facilitate the use of such a paradigm,

the following major research and development stages and associated problems can be envisaged:

- Stage 1 - investigate the feasibility and applicability of various formalisms in software development life cycle and study the interfaces between them.

Three obvious problems of this stage are:

(1) whether it is feasible or not to apply various formalisms in software development life cycle,

(2) where in the life cycle should one kind of formalism be employed, and

(3) how can the different formalisms be interfaced.

Problem (3) is the hardest one, both theoretical questions such as the consistency within a single formalism and between different formalisms and practical questions such as the granularity (abstract level) of the application of each single formalism need to be solved. Actually, there are three kinds of interfaces:

(a) *vertical interfaces* which determine when a specification in one kind of formalism should be replaced by a more refined one within the same formalism (*transformation*),

(b) *horizontal interfaces* which decides where a specification in one formalism should be substituted by an equivalent specification in another formalism (*translation*), and

(c) *mixed interfaces* which divides how a specification in different formalisms coexists.

While problems (1) and (2) have been quite satisfactorily solved, problem (3) needs much more investigation. Techniques for integrating formalisms beyond the system specification phase in the paradigm need to be developed.

- Stage 2 - Search for transition techniques and combine specification and verification.

Some problems of this stage are:

(1) how to refine a specification within a single formalism (transformation),

(2) how to convert a specification in one formalism into an equivalent specification in another formalism (translation),

(3) how to verify the consistency and correctness of such transitions - transformations and translations, and

(4) what kind of computer tools (software) can be developed to automate such transitions.

Although a design (transformation) methodology for predicate transition nets and a translation technique from predicate transition nets to first order temporal logic have been developed, there are much more research efforts needed to make the whole paradigm work.

- Stage 3 - Evaluate the model and the methodology and design a spectrum of specification language.

For this model and the methodology (paradigm) to be practical, the following problems concerning this stage have to be solved:
(1) how well can this paradigm assure software quality (reliability and correctness are the major concerns of this research project) and increase software productivity,
(2) how can this paradigm be used, and
(3) how can a spectrum of specification language be designed to provide mechanisms to the users of this paradigm.

Bibliography

[Ada79] Ada79: "Preliminary Ada Reference Manual", ACM SIGPLAN Notices, Vol.14, No.6 (Part A), June 1979.

[Ada81] Ada81: *The Programming Language Ada Reference Manual*, LNCS [1] 106, Springer-Verlag, 1981.

[Ada83] Ada83: *Reference Manual for the Ada Programming Language* , ANSI/MIL-STD-1815A, American National Standards, 1983.

[Aba88] Abadi, M.: "The Power of Temporal Proofs", RR#30, Systems Research Center, August 15, 1988.

[ABB78] Anderson, E.R., F.C. Belz and E.K. Blum: "Issues in the Formal Specification of Programming Languages", in *Formal Description of Programming Concepts* (ed. E.J. Neuhold), North-Holland, 1978, pp.1-30.

[AEI83] Anttila, M., H. Eriksson and J. Ikonen: "Tools and Studies of Formal Techniques -- Petri Nets and Temporal Logic", in *Protocol Specification, Testing, and Verification, III* (eds. H. Rudin and C.H. West), Elsevier Science Publishers, IFIP, 1983, pp.139-148.

[AFD80] Apt. K.R., N. Francez and W.P. De Roever: "A Proof System for Communicating Sequential Processes", ACM TOPLAS, Vol.2, No.3, July 1980, pp.359-385.

[AM86] Abadi, M. and Z. Manna: "A Timely Resolution", in Proc. of Symposium on Logic in Computer Science, 1986, pp.176-186.

[And86] Andrews, P.B.: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[Apt81] Apt, K.R.: "Ten years of Hoare's logic: a survey - Part I", ACM TOPLAS, Vol.3, No.4, 1981, pp.431-483.

[Ast86] Astesiano, E. et al.: "The Ada Challenge for New Formal Semantics Techniques", Proc. of the Ada - Europe International Conference (ed. P.L. Wallis), Edinburgh, 1986, pp.239-248.

[Bae87] Baer, J.L.: "Modeling Architectural Features with Petri Nets", in LNCS 254, Springer-Verlag, 1987, pp.258-278.

[Bak80] de Bakker, J.W.: *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.

[Bal85] Balzer, R.: "A 15 Year Perspective on Automatic Programming", IEEE Trans. on SE, SE-11, 11, 1985, pp.1257-1267.

[1] LNCS: Lecture Notes in Computer Science.

[Bar84] Barringer, H.: "Formal Specification Techniques for Parallel and Distributed Systems - A Short Review", Proc. of the 3rd Joint Ada Europe / AdaTEC Conference (ed. J. Teller), Brussels, 1984, pp.281-294.

[Bar85] Barringer, H.: *A Survey of Verification Techniques for Parallel Programs*, LNCS 191, Springer-Verlag, 1985.

[Bare81] Barendregt, H.P.: *The Lambda Calculus - Its Syntax and Semantics*, North-Holland, 1981.

[Bau85] Bauer, F.L.: "Where Does Computer Science Come From and Where Is It Going? ", in *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North-Holland, 1985, pp.31-44.

[BB85] de Bruin, A. and W. Bohm: "The Denotational Semantics of Dynamic Networks of Processes", ACM TOPLAS, Vol.7, No.4, Oct. 1985, pp.656-679.

[BBFM82] Berg, H.K., W.E. Boebert, W.R. Franta and T.G. Moher: *Formal Methods of Program Verification and Specification*, Prentice-Hall, 1982.

[BBH80] Belz, F.C., E.K. Blum and D. Heimbigner: "A Multi-Processing Implementation -Oriented Formal Definition of Ada in SEMANOL", SIGPLAN Notices, Vol.15, No.11, 1980, pp.202-212.

[BDMP85] Bjorner, D., T. Denvir, E. Meiling and J.S. Pederson: "The RAISE Project: Fundamental Issues and Requirements", RAISE/DDC/EM/1/V6, December, 1985.

[Ber85] Berry, D.M.: "A Denotational Semantics for Shared-Memory Parallelism and Nondeterminism", Acta Informatica, Vol.21, 1985, pp.599-627.

[BG79] Balzer, R. and N. Goldman: "Principles of Good Software Specification and Their Implications for Specification Language", IEEE Specification of Reliable Software, 1979, pp.58-67.

[BJ78] Bjorner, D. and C.B. Jones(eds.): *The Vienna Development Method: The Meta Language*, LNCS 61, Springer-Verlag, 1978.

[BJ82] Bjorner, D. and C.B. Jones: *Formal Specification and Software Development*, Prentice-Hall, 1982.

[BJ87] Bjorner, D. and C.B. Jones(eds.) *VDM'87: VDM-A Formal Method at Work*, LNCS 252, Springer-Verlag, 1987.

[Bjo87] Bjorner, D.: "On the Use of Formal Methods in Software Development", 9th Software Engineering Conference, California, USA, 1987, pp.17-29.

[BK84] Bergstra, J.A. and J.W. Klop: "Process Algebra for Synchronous Communication", Information and Control, Vol.60, 1984, pp.109-137.

[BKP84] Barringer, H., R. Kuiper and A. Pnueli: "Now You May Compose Temporal Logic Specifications", Proc. of 11th Annual ACM Symposium on POPL, 1984, pp.51-63.

[BO80] Bjorner, D. and O.N. Oest(eds.): *Towards a Formal Definition of Ada*, LNCS 98, Springer-Verlag, 1980.

[Boe76] Boehm, B.W.: "Software Engineering", IEEE Trans. Computer, C-25, 1976, pp.1226-1241.

[Boe81] Boehm, B.W. *Software Engineering Economics*, Prentice-Hall, 1981.

[BRR87] Brauer, W., W. Reisig and G. Rozenber(eds.): *Petri Nets: I -- Central Models and Their Properties and II -- Applications and Relationships to Other Models of Concurrency*, LNCS 254 and 255, Springer-Verlag, 1987.

[Bro85] Brookes, S.D.: "An Axiomatic Treatment of a Parallel Programming Language", in LNCS 193, 1985, pp.41-60.

[Brow86] Browne, M.C.: "An Improved Algorithm for the Automatic Verification of Finite State Systems Using Temporal Logic", Proc. of IEEE 1986 Symposium on Logic in Computer Science, 1986, pp260-266

[Bur85] Burns, A.: *Concurrent Programming in Ada*, Cambridge University Press, 1985.

[BWP87] Broy, M., M. Wirsing and P. Pepper: "On the Algebraci Definition of Programming Languages", ACM TOPLAS, Vol.9, No.1, 1987, pp.54-99.

[BZ83a] de Bakker, J.W. and J.I. Zucker: "Processes and the Denotational Semantics of Concurrency", Information and Control, Vol.54, No.1-2, 1983, pp.70-120.

[BZ83b] de Bakker, J.W. and J.I. Zucker: "Processes And A Fair Semantics for the Ada Rendezvous", LNCS 154, Springer-Verlag, 1983, pp.52-66.

[CE81] Clarke, E.M. and E.A. Emerson: "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic", in LNCS 131, Springer-Verlag, 1981, pp.52-71.

[CES83] Clarke, E.M., E.A. Emerson and A.P. Sistla: "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", Proc. of 10th Annual ACM Symposium on POPL, 1983, pp.117-126.

[Che84] Cherry, G.W.: *Parallel Programming in ANSI Standard Ada*, Reston Publishing, 1984.

[CIP85] CIP Language Group: *The Munich Project CIP: Volume I: The Wide Spectrum Language-CIPL*, LNCS 183, Springer-Verlag, 1985.

[CIP87] CIP Language Group: *The Munich Project CIP: Volume II: The Program Transformation System CIP-S*, LNCS 292, Springer-Verlag, 1987.

[CL73] Chang, C.L. and R.C.T. Lee: *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[Coh86] Cohen, N.H.: "Ada Axiomatic Semantics: Problems & Solutions", Proc. of the Ada - Europe International Conference (ed. P.L. Wallis), Edinburgh, 1986.

[Dav87] Davis: "The Lift Problem", in Proc. of the Fourth International Workshop on Software Specification and Design, Monterey, California, 1987.

[DDH72] Dahl, O.J., W. Dijkstra and C.A.R. Hoare: *Structured Programming*, Academic Press, N.Y., 1972.

[Deu82] Deutsch, M.S.: *Software Verification and Validation*, Prentice-Hall, 1982.

[DG83] Diaz, M. and G. Guidacci Da Silverira: " On the Specification and Validation of Protocals by Temporal Logic and Nets", in Information Processing 83, Proceeding of IFIP83 Congress, Paris, Sept. 1983, pp.47-52.

[Dia87] Diaz, M.: "Petri Net Based Models in the Specification and Verification of Protocols", in LNCS 254, Springer-Verlag, 1987, pp.135-170.

[Dij75] Dijkstra, E.W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", CACM, Vol.18, No.8, 1975.

[DKS83] Dewar, R., P. Kruchten and E. Schonberg: "What Should Be In A Formal Definition of Ada", 8th Meeting of the Ada-Europe Formal Semantics Working Group, Bruxelles, 1983.

[EM85] Ehrig, H. and B. Mahr: *Fundamentals of Algebraic Specification: Equations and Initial Semantics*, Springer-Verlag, 1985.

[EW85] Ehrig, H. and H. Weber: "Algebraic Specification of Modules", in Formal Models in Programming (eds. E.J. Neuhold and G. Chroust), North-Holland, IFIP, 1985, pp.231-258.

[Fit72] Fitting, M.C.: "Tableau Methods of Proof for Modal Logics", Norte Dame Journal of Formal Logic, Vol.XIII, No.2, 1972. pp.237-247.

[Fit83] Fitting, M.C.: "Proof Methods for Modal and Intuitionistic Logics", *Synthese Library*, Vol.169, Dordrecht, Holland, 1983.

[Fla79] Flan, L.: "A Unified Approach to the Specification and Verification of Abstract Data Types", IEEE Specification of Reliable Software, 1979, pp.162-169.

[Gen87] Genrich, H.J.: "Predicate Transition Nets", in LNCS 254, Springer-Verlag, 1987, pp.207-247.

[Ger82] Gerth, R.: "A Sound and Complete Hoare Axiomatization of Ada Rendezvous", in LNCS 140, Springer-Verlag, 1982, pp.252-264.

[GG75] Goodenough, J.B. and S.L. Gerhart: "Toward a Theory of Test Set Selection", IEEE Trans. on Software Engineering, SE-1, No.2, 1975, pp.157--173.

[GHM78] Guttag, J.V., E. Horowitz and D.R. Musser: "Abstract Data Types and Software Validation", CACM, Vol.21, No.12, 1978, pp.1048-1064.

[GHW85] Guttag, J.V., J.J. Horning and J.M. Wing: "LARCH in Five Easy Pieces", Technical Report #5, Systems Research Center, July, 1985.

[GKL80] Gouge, V.D., G. Kahn and B. Lang: "On the Formal Definition of Ada", in LNCS 94, Springer-Verlag, 1980.

[GL81] Genrich, H.J. and K. Lautenbach: "System Modelling with High Level Petri Nets", Theoretical Computer Science, Vol.13, 1981, pp.109-136.

[GLT80] Genrich, H.L., K. Lautenbach and P.S. Thiagarajan: "Elements of Net Theory", LNCS 84, Springer-Verlag, 1980, pp.21-164.

[GM86] Gehani, N. and A.D. McGettrick (eds.): *Software Specification Techniques*, Addison-Wesley, 1986.

[Gog81] Goguen, J.A.: "More Thoughts on Specification and Verification", ACM SIGSOFT Software Engineering Notes, Vol.6, No.3, 1981.

[Gor79] Gordon, M.J.C.: *The Denotational Decsription of Programming Languages*, Springer-Verlag, 1979.

[GPSS80] Gabbay, D., A. Pnueli, S. Shlab and J. Stavi: "The Temporal Analysis of Fairness", Seventh ACM Symposium on Principles of Programming Languages, 1980, pp.163-173.

[GTWW79] Goguen, J.A., J.W. Thatcher, E.G. Wagner and J.B. Wright: "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in *Current Trends in Programming Methodology IV* (ed. R. Yeh), Prentice-Hall, 1979, pp.80-149.

[Gri81] Gries, D.: *The Science of Programming*, Springer-Verlag, 1981.

[Gue81] Guessarian, I.: *Algebraic Semantics*, LNCS 99, Springer-Verlag, 1981.

[Gut80] Guttag, J.: "Notes on Type Abstraction", IEEE TSE, Vol.SE-6, No.1, 1980.

[Hai82] Hailpern, L.: *Verifying Concurrent Processes Using Temporal Logic* , Springer-Verlag, 1982.

[Har87] Harel, D.: "Statechart: A Visual Formalism for Complex Systems", Science of Computer Programming, Vol.8, 1987, pp.231-274.

[Har88] Harel, D.: "On Visual Formalisms", CACM, Vol.31, No.5, 1988, pp.514-530.

[Hara88] Harandi, M.T.: "Report on the Fourth International Workshop on Software Specification and Design", Software Engineering Notes, Vol.13, No.1, 1988, pp.20-45.

[Hau84] Hausen, H.L.(ed.): *Software Validation: Inspection, Testing, Verification, Alternatives*, North-Holland, 1984.

[HIR88] He, X., R. Iyengar and J. Roach: "A Systematic Study of Unification", Proc. of 3rd International Symposium on Knowledge Engineering, Madrid, Spain, 1988.

[HJJJ85] Huber, P., A.M. Jensen, L.O. Jepsen and K. Jensen: "Towards Reachability Trees for High-Level Petri Nets", in LNCS 188, Springer-Verlag, 1985, pp.215-233.

[HL88] He, X. and J.A.N. Lee: "A Strategy for Integrating Formalisms in Software Development", Proc. of 6th CIPS Edmonton Computer Conference, Edmonton, Canada, 1988, pp.33-42.

[HL89a] He, X. and J.A.N. Lee: "Deriving Temporal Logic Specifications from Predicate Transition Petri Net Specifications", Proc. of Int'l Conf. on Software Engineering and Knowledge Engineering, Chicago, June 15-17, 1989.

[HL89b] He, X. and J.A.N. Lee: "A New Methodology for Writing Predicate Transition Net Specifications", Proc. of First Annual Symposium on Parallel and Distributed Processing, Dallas, Texas, May 22-23, 1989.

[HL89c] He, X. and J.A.N. Lee: "Verifying Predicate Transition Petri Net Specifications by Using Temporal Logic', submitted for publication.

[HL89d] He, X. and J.A.N. Lee: "A Design Methodology for Predicate Transition Petri Nets", submitted for publication.

[HL89e] He, X. and J.A.N. Lee: "A Comparison of Formal Definitions of Ada Tasking", submitted for publication, also as TR 89-4, Department of Computer Science, VPI&SU.

[HLi83] Hennessy, M.C.B. and W. Li: "Translating a Subset of Ada into CCS", in *Formal Description of Programming Concepts - II* (ed. D. Bjorner), North-Holland, IFIP, 1983, pp.227-248.

[Hoa69] Hoare, C.A.R.: "An Axiomatic Basis for Computer Programming", CACM, Vol.12, No.10, 1969, pp.576-583.

[Hoa78] Hoare, C.A.R.: "Communicating Sequential Processes", CACM, Vol.21, No.8, Aug. 1978.

[Hoa87] Hoare, C.A.R. et al: "Laws of Programming", CACM, Vol.30, No.8, Aug. 1987, pp.672-687.

[Hor82] Horning, J.J.: "Program Specification: Issues and Observations", in LNCS 134, Springer-Verlag, 1982.

[How87] Howden, W.E.: "A Functional Approach to Program Testing and Analysis", IEEE Trans. on Software Engineering, SE-12, No.10, 1986, pp.997-1005.

[HRD87] He, X., R. Russell and J. Dickey: "Workload Expert System and Optimizer", Proc. of 7th International Congress on Cybernetics and Systems, London, United Kingdom, 1987, pp.68-73.

[HRY88] Howell, R.R., L.E. Rosier and H.C. Yen: "A Taxonomy of Fairness and Temporal Logic Problems for Petri Nets", LNCS 324, Springer-Verlag, 1988, pp.351-359.

[HW73] Hoare, C.A.R. and N. Wirth: "An Axiomatic Definition of Programming Language PASCAL', Acta Informatica, Vol.2, 1973, pp.335-355.

[HX85] He, X. and J. Xu: "The Design of the Static Denotational Semantics of Programming Language XCY", J. of Computer Research and Development (Chinese), Vol.7, 1985, pp.50-55.

[Jen87] Jensen, K.: "Colored Petri Nets", in LNCS 254, Springer-Verlag, 1987, pp.248-299.

[Jon86] Jones, C.B.: *Syetmatic Development Using the VDM Approach*, Prentice-Hall, 1986.

[Jou85] Jouannaud, J.P.(ed.): *Rewriting Techniques and Applications* LNCS 202, Springer-Verlag, 1985.

[Lam80] Lamport, L.: "The 'Hoare Logic' of Concurrent Programs", Acta Informatica, Vol.14, 1980, pp.21-37.

[Lam83] Lamport, L.: "Specifying Concurrent Program Modules", ACM TOPLAS, Vol.5, No.2, 1983, pp.190-222.

[Lam85] Lamport, L.: "What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority", Conf. Record of the 12th Annual ACM Symposium on POPL, 1985, pp.78-83.

[Lam89] Lamport, L.: "A Simple Approach to Specifying Concurrent Systems", CACM, Vol.32, No.1, 1989, pp.32-45.

[Lau87] Lautenbach, K.: "Linear Algebraic Techniques for Place Transition Nets", in LNCS 254, Springer-Verlag, 1987, pp.142-167.

[LB79] Liskov, B.H. and V. Berzins: "An Appraisal of Program Specifications", in *Research Directions in Software Technology* (ed. P. Wegner), MIT Press, 1979, pp.276-301.

[LBj80] Lovengreen, H.H. and D. Bjorner: "On A Formal Model Of The Tasking Concept In Ada", SIGPLAN Notices, Vol.15, No.11, Nov. 1980.

[Lee72] Lee, J.A.N.: *Computer Semantics*, Van Nostrand, 1972.

[Les87] Lescanne, P.(ed.): *Rewriting Techniques and Applications*, LNCS 256, Springer-Verlag, 1987.

[LH90] Lee, J.A.N. and X. He: "A Methodology for Test Selection", to appear in Journal of Systems and Software, 1990; also as TR 88-29, Department of Computer Science, VPI&SU.

[Li82] Li, W.: "An Operational Semantics of Ada Multitasking and Exception Handling", Proc. of AdaTEC Conf., Washington, 1982.

[LL85] Li, W. and P.E. Lauer: "Using the Structural Operational Approach to Express True Concurrency", in *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), Elsevier Science Publishers, IFIP , 1985, pp.147-164.

[Llo84] Lloyd, J.W.: *Foundations of Logic Programming*, Springer-Verlag, 1984.

[LS87] Loeckx, J. and K. Sieber: *The Foundations of Program Verification - Second Edition*, John Wiley & Sons, 1987.

[Luc78] Lucas, P.: "On the Formalization of Programming Languages: Early History of Main Approaches", in LNCS 61, Springer-Verlag, 1978.

[LW82] Leszczylowski, J. and M. Wirsing: "A System for Reasoning within and about Algebraic Specifications", in LNCS 137, Springer-Verlag, (1982), 257-282.

[LZ75] Liskov, B.H. and S.N. Zilles: "Specification Techniques for Data Abstraction", IEEE TSE, Vol.SE-1, No.1, 1975, pp.7-19.

[Mai85] Maiocchi, M.: "The Use of Petri Nets in Requirements and Functional Specification", in *System Description Methodologies* (eds. D. Teichroew and G. David), North-Holland, IFIP, 1985, pp.253-272.

[MBT85] Marsan M.A., G. Balbo and K. Trivedi(eds.): *1985 International Workshop on Timed Petri Nets*, Torino, Italy, July 1-3, 1985.

[MG87] Meiling, E. and C.W. George: "The RAISE Language, Method and Tools", RAISE/DDC/EM/27/V1, September, 1987.

[Mil80] Milner, R.: *A Calculus of Communication Systems*, LNCS 92, Springer-Verlag, 1980.

[MM83] Moszkowski, B. and Z. Manna: "Reasoning in Interval Temporal Logic", in LNCS 164, Springer-Verlag, 1983, pp.360-370.

[Mos83] Mosses, P.: "Abstract Semantic Algebras", in *Formal Description of Programming Concepts - II* (ed. D. Bjorner), North-Holland, IFIP, 1983, pp.45-70.

[MP81] Manna, Z. and A. Pnueli: "Verification of Concurrent Programs: the Temporal Framework", in *The Correctness Problem in Computer Science* (eds. R.S. Boyer and J.S. Moore), Academic Press, 1981, pp.215-274.

[MP83] Manna, Z. and A. Pnueli: "How to Cook a Temporal Proof System for Your Pet Language", 10th Annual ACM Symposium on the Principles of Programming Languages, Austin, Texas, 1983, pp.141-154.

[MS76] Milne, R. and C. Strachey: *A Theory of Programming Language Semantics*, John Wiley & Sons, 1976.

[Mus80] Musser, D.R.: "Abstract Data Type Specification in the Affirm System", *IEEE Trans. on SE*, SE-6, (Jan. 1980), 24-32.

[MV87] Memmi, G. and J. Vautherin: "Analysing Nets by the Invariant Method", in LNCS 254, Springer-Verlag, 1987, pp.300-336.

[MW84] Manna, Z. and P. Wolper: "Synthesis of Communicating Processes from Temporal Logic Specifications", ACM Trans. on Programming Languages and Systems, Vol.6, No.1, 1984, pp.68-93.

[MZGT85] Mandrioli, D., R. Zicari, C. Ghezzi and F.Tisato: "Modeling the Ada Task System By Petri Nets", Computer Language, Vol.10, No.1, 1985, pp.43-61.

[NHN78] Nakajima, R., M. Honda and H. Nakahara: "Describing and Verifying Programs with Abstract Data Types", in *Formal Description of Programming Concepts* (ed. E.J. Neuhold), IFIP, North-Holland, 1978, pp.527-556.

[NHK85] Nyberg, K.A., A.A. Hook and J.F. Kramer: "The Status of Verification Technology for the Ada Language", IDA Paper, P-1859, July, 1985.

[NHS83] Nelson, R.A., L.M. Haibt and P.B. Sheridan: "Casting Petri Nets into Programs", IEEE Trans. on SE, SE-9, 5, 1983, pp.590-602.

[Nos85] Nossum, R.: "Automated Theorem Proving Methods", BIT, Vol.25, 1985, pp.51-64.

[Obe87] Oberquelle, H.: "Human-Machine Interaction and Role / Function / Action-Nets", in LNCS 254, Springer-Verlag, 1987, pp.171-190.

[OG76] Owicki, S. and D. Gries: "An Axiomatic Proof Technique for Parallel Programs I", Acta Informatica, Vol.6, 1976, pp.319-340.

[Ohl88] Ohlbach, H.J.: "A Resolution Calculus for Model Logics", LNCS 310, Springer-Verlag, 1988, pp.500-516.

[Oll74] Ollongren, A.: *Definition of Programming Languages by Interpreting Automata*, Academic Press, 1974.

[Ost87] Osterweit, L.: "Software Processes are Software Too", IEEE 1987 9th International Conference on Software Engineering, March 30 - April 2, 1987, Monterey, California, USA, pp.2-13.

[Par77] Parnas, D.L.: "The Use of Precise Specifications in the Development of Software", Proceeding of Information Processing 77, IFIP, North-Holland, 1977, pp.861-868.

[PD82] Pnueli, A. and W.P. DeRoever: "Rendezvous with Ada - A Proof Theoretical View", Proc. of the AdaTEC Conference, Crystal City, 1982, pp.129-136.

[Pe62] Petri, C.A.: "Fundamentals of a Theory of Asynchronous Information Flow", Information Processing 1962, North-Holland, 1962, pp.386-390.

[Pet77] Peterson, J.L.: "Petri Nets", ACM Computing Survey, Vol.9, 1977, pp.223-252.

[Pet81] Peterson, J.L.: *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[Plo76] Plotkin, G.D.: "A Power Domain Construction", SIAM J. on Comp., Vol.5, 1976, pp.452-487.

[Plo82] Plotkin, G.D.: "A Operational Semantics for CSP", Proc. of IFIP Working Conf. on Formal Description of Programming Concepts II, North-Holland, 1982, pp.199-223.

[PN87] PN87: *International Workshop on Petri Nets and Performance Models*, Madison, Wisconsin, August, 1987.

[Pnu77] Pnueli, A.: "The Temporal Logic of Programs", 19th Annual Symposium on Foundations of Computer Science, Nov. 1977, pp.46-57.

[Pnu81] Pnueli, A.: "The Temporal Semantics of Concurrent Programs", Theoretical Computer Science, 13, 1981, pp.45-60.

[Pnu86] Pnueli, A.: "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", in LNCS 224, Springer-Verlag, 1986, pp.510-584.

[Pol81] Polak, W.: "Program Verification Based on Denotational Semantics", Conf. Record of the 8th Annual Symposium on the POPL , 1981, pp.149-158.

[QS82] Queille, J.P. and J. Sifakis: "Specification and Verification of Concurrent Systems in CESAR", in LNCS 137, 1982, pp.337-351.

[Ram86] Ramamoorthy, C.V. (et al): "Software Quality and Requirement Specification", IEEE 1986 International Conf. on Computer Languages , 1986, pp.75-83.

[Rei85a] Reisig, W.: *Petri Nets - An Introduction*, Springer-Verlag, 1985.

[Rei85b] Reisig, W.: "On the Semantics of Petri Nets", in *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North-Holland, 1985, pp.347-372.

[Rei87a] Reisig, W.: "Petri Nets in Software Engineering", in LNCS 255, Springer-Verlag, 1987.

[Rei87b] Reisig, W.: "Embedded System Description Using Petri Nets", in *Embedded Systems* (eds. A. Kundig, R.E. Buhrer and J. Dahler), Springer-Verlag, 1987, pp.18-62.

[Rei87c] Reisig, W.: "Place Transition Systems", in LNCS 254, Springer-Verlag, 1987.

[Roz84] Rozenberg, G.(ed.) *Advances in Petri Nets 1984*, LNCS 188, Springer-Verlag, 1984.

[Roz87] Rozenberg, G.(ed.) *Advances in Petri Nets 1987*, LNCS 266, Springer-Verlag, 1987.

[RS77] Ramamoorthy, C.V. and H.H. So: "Survey of Principles and Techniques of Software Requirements and Specification", in Software Engineering Techniques 2, 1977, pp.265-318.

[RU71] Rescher, N. and A. Urquhart: *Temporal Logic*, Springer-Verlag, 1971.

[Sam86] Sammet, J.E.: "Why Ada Is Not Just Another Programming Language", CACM, Vol.29, No.8, Aug. 1986, pp.722-732.

[SB82] Swartout, W. and R. Balzer: "On the Inevitable Intertwining of Specification and Implementation", CACM, Vol.25, No.7, 1982, pp.438-440.

[Sco76] Scott, D.: "Data Types as Lattices", SIAM J. Comput., Vol.5, 1976, pp.522-587.

[Sco82] Scott, D.S.: "Domains for Denotational Semantics", in LNCS 140, Springer-Verlag, 1982, pp.577-613.

[SM83] Suzuki, I. and T. Murata:: "A Method for Stepwise Refinement and Abstraction of Petri Nets", Journal of Computer and System Sciences, Vol.27, 1983, pp.51-76.

[SMS81] Schwartz, R.L. and P.M. Melliar-Smith: "Temporal Logic Specification of Distributed System", Proc. of the 2nd International Conference on Distributed Computing Systems, Paris, 1981, pp.446-454.

[SMSV83] Schwartz, R.L., P.M. Melliar-Smith and F.H. Vogt: "An Interval Based Temporal Logic ", in LNCS 164, Springer-Verlag, 1983, pp.443-457.

[Smy78] Smyth, M.B.: "Power Domains", JCSS Vol.16, 1978, pp.23-36.

[Sto77] Stoy, J.E.: *Denotational Semantics: the Scott - Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[Ten76] Tennent, R.D.: "The Denotational Semantics of Programming Languages", CACM, Vol.19, 1976, pp.437-453.

[Thi87] Thiagarajan, P.S.: "Elementary Net Systems", in LNCS 254, Springer-Verlag, 1987, pp.26-59.

[Var87] Vardi, M.Y.: "Verification of Concurrent Programs - the Automata-Theoretic Framework", Proc. of IEEE Symposium on Logic in Computer Science, 1987, pp.167-176.

[VDM87] VDM.87: *VDM - A Formal Method at Work*, LNCS 252, Springer-Verlag, 1987.

[VDM88] VDM.88: *VDM - The Way Ahead*, LNCS 328, Springer-Verlag, 1988.

[Vog88] Vogt, F.H.(ed.): *Concurrency 88*, LNCS 335, Springer-Verlag, 1988.

[Vos87a] Voss, K.: "Nets in Data Bases", in LNCS 255, Springer-Verlag, 1987, pp.97-134.

[Vos87b] Voss, K.: "Nets in Office Automation", in LNCS 255, Springer-Verlag, 1987, pp.234-257.

[VW86] Vardi, M.Y. and P. Wolper: "An Automata-Theoretic Approach to Automatic Program Verification", Proc. of IEEE Symposium on Logic in Computer Science, 1986, pp.332-344.

[Wal87] Wallen, L.A.: "Matrix Proof Methods for Modal Logics", in Proc. of 10th IJCAI, 1987, pp.917-923.

[Weg72] Wegner, P.: "The Vienna Definition Language", Computing Surveys, Vol.4, No.1, 1972, pp.5-63.

[Win86] Wing, J.: "Role of Formal Specification" (Discussion Group Summary of NRL Invitational Workshop on Testing and Proving, July, 1986), ACM SIGSOFT Software Engineering Notes, Vol.11, No.5, 1986, pp.65-67.

[Wir71] Wirth, N.: "Program Development by Stepwise Refinement", CACM, Vol.14, No.4, 1971, pp.221-227.

[Wol82] Wolper, P.: "Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic", Proc. of 9th ACM Symposium on POPL, 1982, pp.20-33.

[Zam87] Zamfir, M.: "Initial Algebra Semantics and Concurrency", in LNCS 298, Springer-Verlag, 1987, pp.528-549.

[Zav82] Zave, P.: "An Operational Approach to Requirement Specification for Embedded Systems", IEEE Trans. on SE, SE-8, 3, 1982, pp.250-269.

[Zem85] Zemanek, H.: "Formal Definition: The Hard Way", in *Formal Models in Programming* (eds. E.J. Neuhold and G. Chroust), IFIP, North-Holland, 1985.